Michael Nguyen
Shih-Lei Chen
A Priyanka

University of California, Irvine
SWE 261P Software Testing and Debugging

CRYPTOMATOR

Team Members:
Michael Nguyen
Shih-Lei Chen
A Priyanka

Michael Nguyen
Shih-Lei Chen
A Priyanka

Table of Contents:

Michael Nguyen
Shih-Lei Chen
A Priyanka

Section One

Github Repository: https://github.com/michaelnguyen26/cryptomator.git

Introduction:

Cryptomator enables independent and self-protection of cloud data and operates with any cloud storage service that syncs with a local directory, which includes Google Drive, OneDrive, etc. Majority of the cloud service providers maintain the decryption key or encrypt data only during transmission.

Cryptomator on the other hand allows the user and the user alone to hold the key to decrypt their cloud data. Since this is a client side service, data is never shared with any online service. Cryptomator's technology encrypts both files and filenames with AES and 256 bit key length and follows the latest standards.

Language Used: Java

Project Contents (Provided by the IntelliJ Statistics Plugin) :

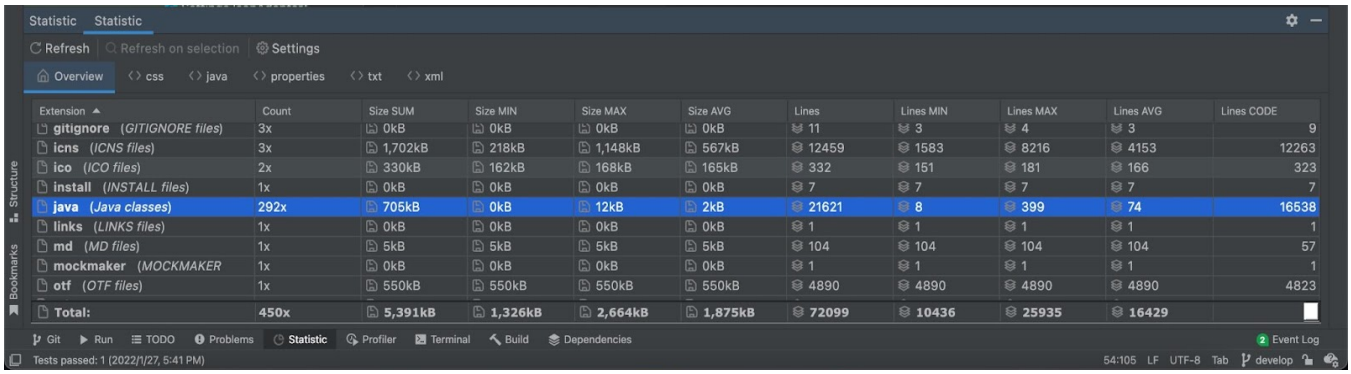

Figure 1: Contents of Cryptomator Project

Michael Nguyen
Shih-Lei Chen
A Priyanka

Setup:

Install Maven

- brew install maven

Run Maven

- mvn clean install (IntelliJ)
- This builds all the jars and bundles them together under the ‘target’ folder. This can now be used to build native packages.

```
[INFO] --- maven-jar-plugin:3.2.0:jar (default-jar) @ cryptomator ---
[INFO] Building jar: /Users/shihlei/Documents/MSWE/261P(debug)/cryptomator/target/cryptomator-1.7.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-install-plugin:2.4:install (default-install) @ cryptomator ---
[INFO] Installing /Users/shihlei/Documents/MSWE/261P(debug)/cryptomator/target/cryptomator-1.7.0-SNAPSHOT.jar to /Users/shih
.7.0-SNAPSHOT/cryptomator-1.7.0-SNAPSHOT.jar
[INFO] Installing /Users/shihlei/Documents/MSWE/261P(debug)/cryptomator/pom.xml to /Users/shihlei/.m2/repository/org/cryptom
0-SNAPSHOT.pom
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  38.328 s
[INFO] Finished at: 2022-01-23T17:43:07-08:00
[INFO] ------------------------------------------------------------------------
shihlei@chenshileideMBP ~/D/M/2/cryptomator (develop)>
```

Figure 2: Successful Completion of the Build

Running Cryptomator

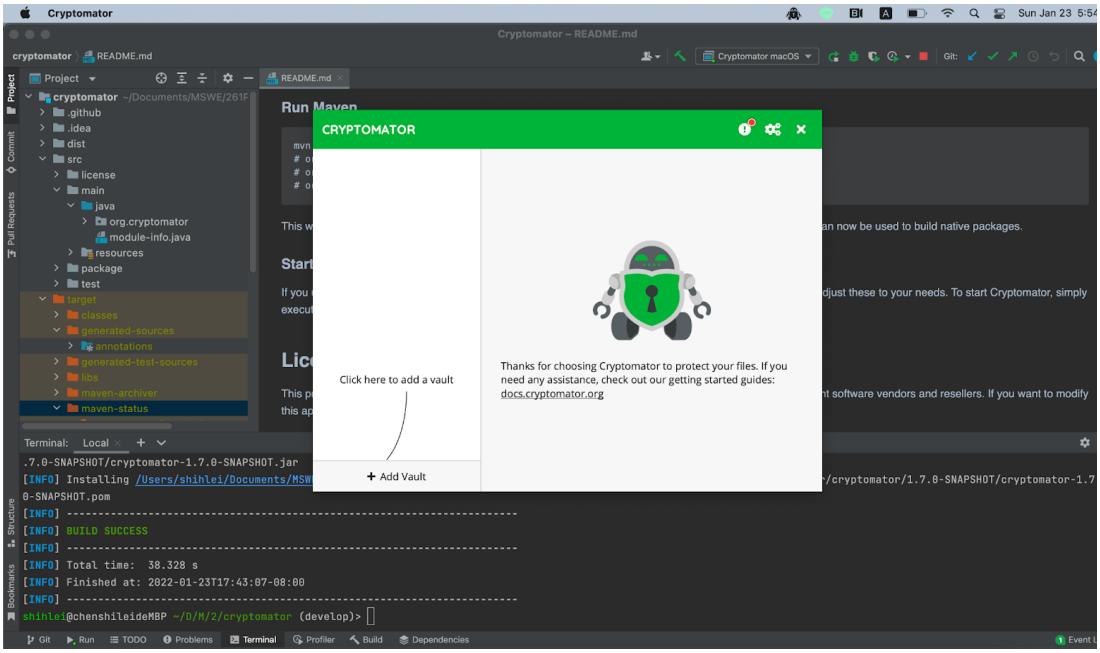

Figure 3: Cryptomator UI

Michael Nguyen
Shih-Lei Chen
A Priyanka

Exploring and Executing the Test Cases

The software includes test cases to test the functionality of the UI, logging system, launcher, environment/settings, and network system. These existing test cases include Junit and Mockito to verify their functionality.

Each of the test cases are divided into multiple folders indicating the specific component that is being tested. These tests could be run individually in their respective folders as shown below.

The ▶ button indicates the specific function that we want to perform our test on. A testing block is indicated by the @Test as shown in line 50 and line 61.

Result of the Test Cases

The test case results are shown in the bottom left corner and the green checkmark indicates if a test case *passes* or a red if the test case *fails*.

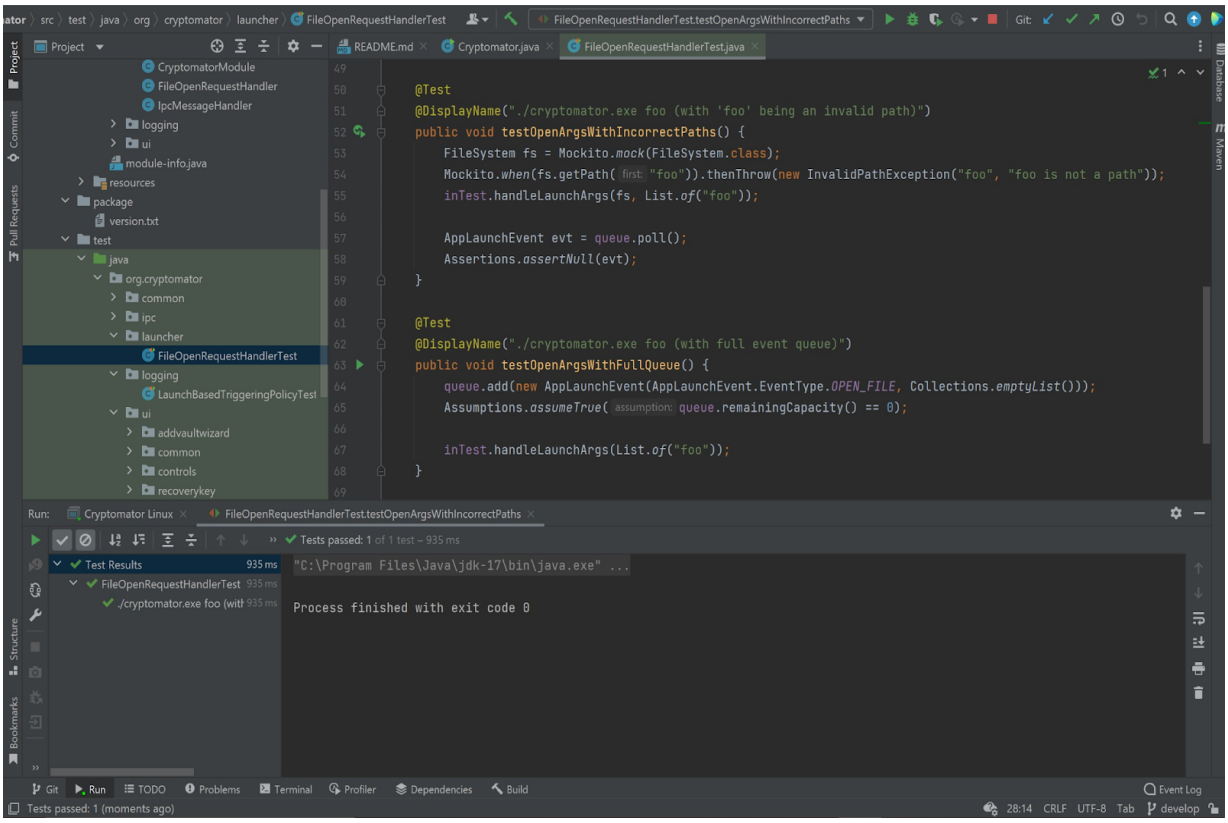

Figure 4: A Successful Test Case

Michael Nguyen
Shih-Lei Chen
A Priyanka

Test Case Functionality - Another Perspective

The UI section for the test cases contains different testing styles that tackle similar concepts of software testing. As shown below, **@ParameterizedTest** and **@CsvSource** are used to test the function **public void testEscapeNonAsciiChars**. This test allows the function to receive a comma separated value list as its parameters and check if the test passes in the *Assertion* step.

This testing technique offers flexibility in software testing since the tester is able to provide various inputs. The benefits include testing a variety of edge cases and tackling problems that are not statically present as a single input.

The result of this test case is the same as the previous example; however, in the bottom left hand corner the tester is able to see how each parameter performs in the test given by the [1], [2], and [3]. For clarity, the IDE also displays the test parameters passed to the function as well.

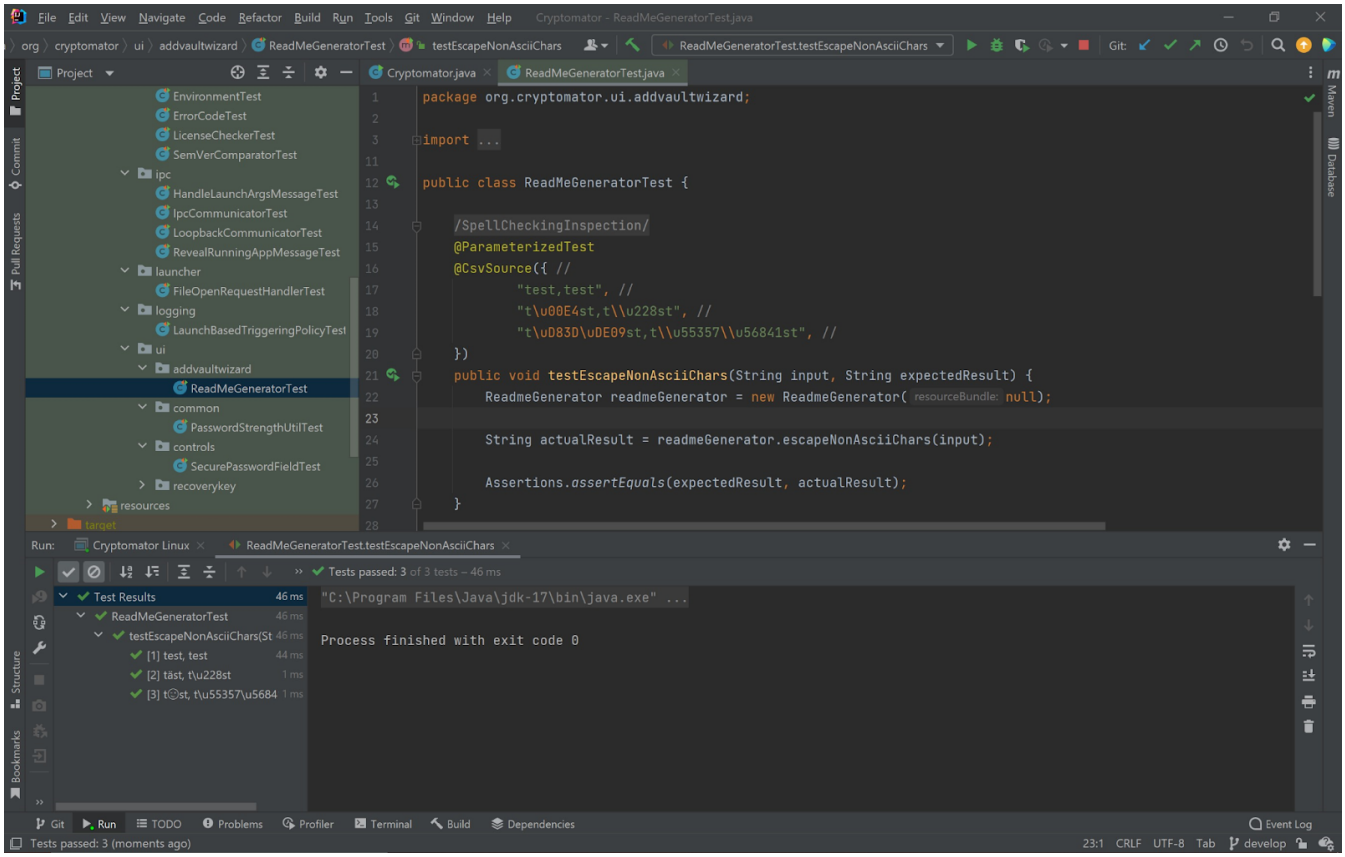

Figure 5: A Successful Parameterized Test Case with Parameters Shown on the Left

Michael Nguyen
Shih-Lei Chen
A Priyanka

Partitioning:

Systematic Functional Testing vs. Partition Testing

These two testing paradigms offer a variety of testing strengths and weaknesses. Partition testing separates an input space into classes whose *union* is the entire space. The general principle of partition testing is to choose samples that are more likely to include "failure zones" of the input space. This is because testing failures are *sparse* in the overall input space; however, there are regions in which these failures are more prominent.

On the other hand, functional testing utilizes the *specification*, which is either formal or informal in order to partition the input space. In other words, functional testing utilizes test cases from program specifications. A functional specification could be described as the intended program behavior.

Cryptomator Partitioning Testing

Partition One:

```
@Test
@DisplayName("\"1\".append(\"test\")")
public void append1() {
  pwField.setPassword(1 + "");
  pwField.appendText("test");

  Assertions.assertEquals("1test", pwField.getCharacters().toString());
}
```

We decided to partition the input space for the *pwField* of the *SecurePasswordField* class which only accepts characters and strings. We tested the type casting capabilities for the integer value in the password field. Since only strings are allowed, the value of "1" was set as the password and a blank string was appended at the end.

The result of this test case passed, which tells us that integer values are able to be set as a password *if and only if* a blank string can be appended at the end of the value. At any point a password is set with an actual integer value, the test case will fail.

Michael Nguyen
Shih-Lei Chen
A Priyanka

Partition Two:

```
AutoCompleter autoCompleter = new AutoCompleter(Set.of("tame", "teach",
"teacher", "irvine" + 2022));


@ParameterizedTest
@DisplayName("find 'irvine'")
@ValueSource(strings = {"i", "ir", "irv", "irvi","irvin", "irvine",
"irvine2022"})
public void testFindTest(String prefix) {
  Optional<String> result = autoCompleter.autocomplete(prefix);
  Assertions.assertTrue(result.isPresent());
  Assertions.assertEquals("irvine2022", result.get());
}
```

In the second partition, we decided to partition the input space for the *autocomplete* method from the *autoCompleter* class which only tests for strings as an argument. In the *autoCompleter* object declared at the top, we added to the dictionary the string “irvine” along with the integer “2022.” This allows us to test integers and strings that are type casted together and determine if the *autoCompleter* object will automatically auto complete the test provided in the *assertEquals* method which is given a string and values appended together.

The result of this test case passed, which shows that the *autoCompleter* is able to discern if both characters and values are contained in the string by type casts. Additionally, comparing the list of words provided, “irvine2022” has a longer sequence of characters than the other words. Therefore, there is no limit to the number of characters that the *autoCompleter* could handle.

Michael Nguyen
Shih-Lei Chen
A Priyanka

Section Two

Github Repository: Test Cases and Repo

Finite Models and Testing:

A finite state machine or FSM describes a model's state-based behavior. They contain a set of states and a set of transitions. This can easily be described using nodes and edges, where a node represents a program state and an edge represents a transition operation that modifies the program state to another.[2] All states in consideration within an FSM are within a finite list and the system takes a single state at a time. This enables each input and output scenario to be analyzed and tested.[4]

When a finite state machine transitions to another state, we can test the quality of a system by verifying each state and its transition and also examine all of the possible inputs that could possibly be entered. As a model-based test, a finite model partitions the entire system into smaller manageable components.[1] These components form a group of models to generate tests that represent the entire system. An example of a finite state machine within the Cryptomator software is shown below in Figure 1.

FSM enables:

- Reduced testing time
- Improved quality of test cases
- Test coverage
- Traceability
- Requirements changes

Michael Nguyen
Shih-Lei Chen
A Priyanka

Cryptomator Functional Model:

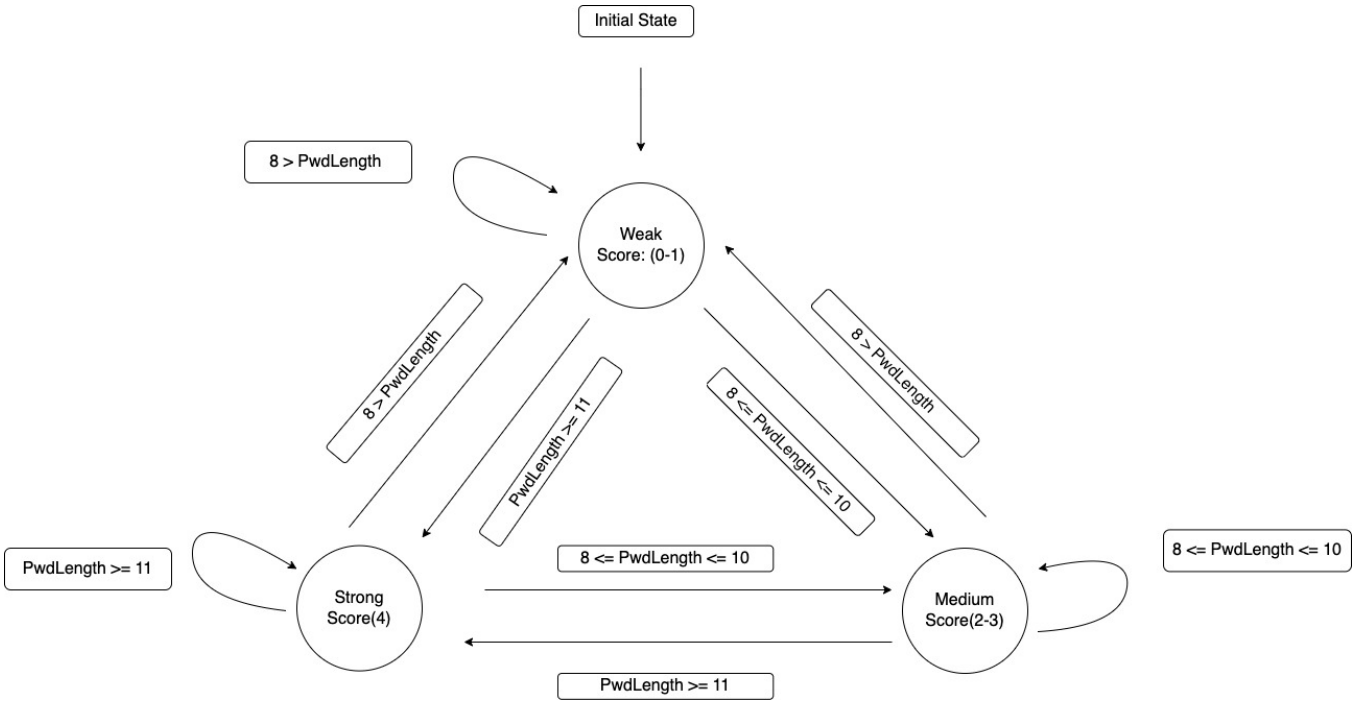

Figure 1: Finite State Model of the Password Strength Functionality

Password Scoring:

Based on the scoring values, we grouped scores 0 & 1 as weak, 2 & 3 as medium, and 4 as strong. Refer to Figures 2-5 for a demonstration of the finite state machine using the password strength functionality.

- 0 - Too guessable: risky password.[5]
- 1 - Very guessable: protection from throttled online attacks. [5]
- 2 - Somewhat guessable: protection from unthrottled online attacks. [5]
- 3 - Safely unguessable: moderate protection from offline slow-hash scenario. [5]
- 4 - Very unguessable: strong protection from offline slow-hash scenarios. [5]

Michael Nguyen
Shih-Lei Chen
A Priyanka

Demo:

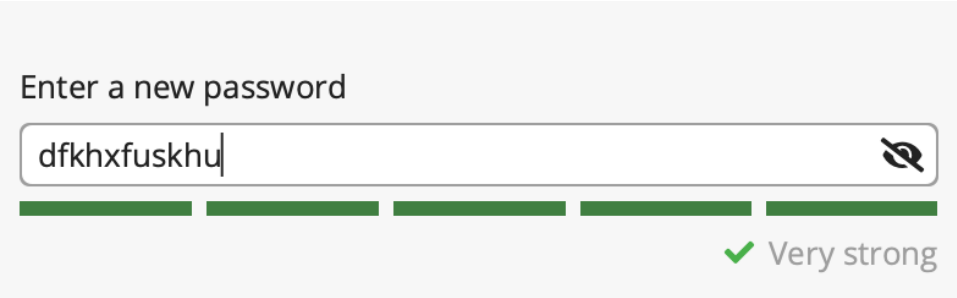

Figure 2: Password with a Character Length of 11

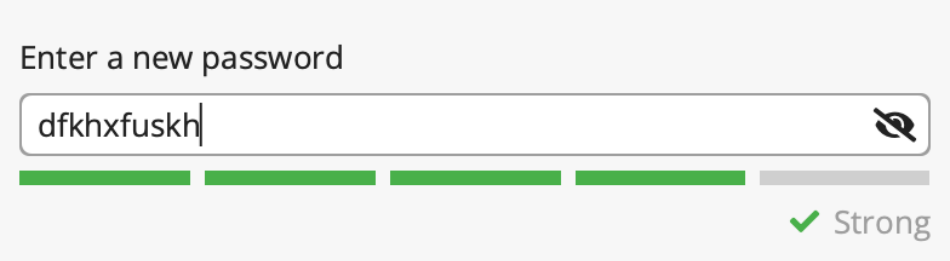

Figure 3: Password with a Character Length of 10

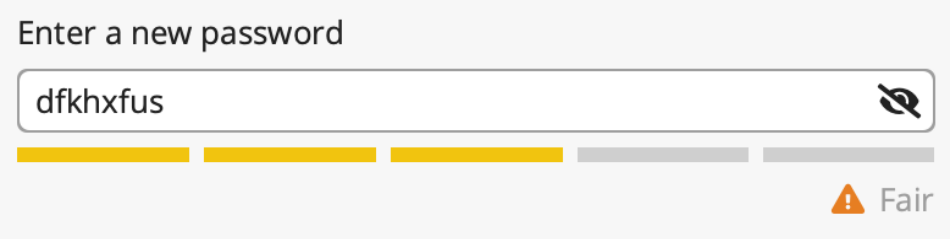

Figure 4: Password with a Character Length of 8

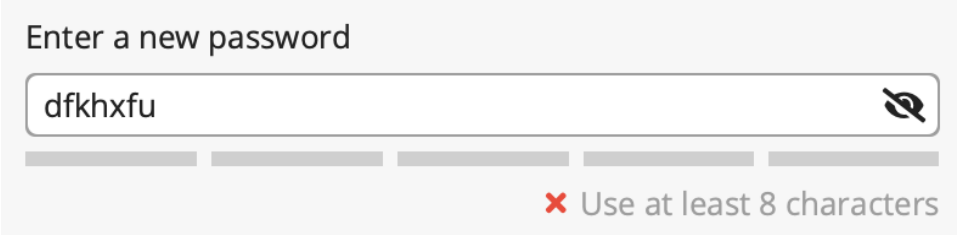

Figure 5: Password with a Character Length Less than 8

Michael Nguyen
Shih-Lei Chen
A Priyanka

Test Cases:

Each test case below contains a set amount of characters that determines the strength of the password. As shown below, increasing the number of characters will strengthen the password. We begin with three characters and end with eleven characters.

The *Mockito* library determines the strength of the password based solely on the number of characters. Therefore, a password with eleven characters or more achieves the highest possible value of four, which determines its strength. Any subsequent password that is less than eleven characters is considered medium or weak with values ranging between 0 - 3.

According to Cryptomator's software, their implementation determines that eight characters is considered medium, ten characters is strong, and eleven characters or more as very strong using the *Mockito* library. Any password below eight characters is not accepted, therefore Cryptomator only accepts password scores with values between 2 - 4. This is described in Figures 2 - 5 above.

```
@Test
public void testForWeak() {
  PasswordStrengthUtil util = new
PasswordStrengthUtil(Mockito.mock(ResourceBundle.class),
Mockito.mock(Environment.class));
  int result1 = util.computeRate("uci");
  Assertions.assertEquals(0, result1);
}

@Test
public void testForWeak2() {
  PasswordStrengthUtil util = new
PasswordStrengthUtil(Mockito.mock(ResourceBundle.class),
Mockito.mock(Environment.class));
  int result1 = util.computeRate("irvine");
  Assertions.assertEquals(1, result1);
}

@Test
public void testForMedium() {
  PasswordStrengthUtil util = new
PasswordStrengthUtil(Mockito.mock(ResourceBundle.class),
Mockito.mock(Environment.class));
  int result1 = util.computeRate("mswe2022");
  Assertions.assertEquals(2, result1);
}
```

Michael Nguyen
Shih-Lei Chen
A Priyanka

```
@Test
public void testForMedium2() {
  PasswordStrengthUtil util = new
PasswordStrengthUtil(Mockito.mock(ResourceBundle.class),
Mockito.mock(Environment.class));
  int result1 = util.computeRate("IrvineMswe");
  Assertions.assertEquals(3, result1);
}

@Test
public void testForStrong() {
  PasswordStrengthUtil util = new
PasswordStrengthUtil(Mockito.mock(ResourceBundle.class),
Mockito.mock(Environment.class));
  int result1 = util.computeRate("MsweIrvine-2022!");
  Assertions.assertEquals(4, result1);
}
```

Michael Nguyen
Shih-Lei Chen
A Priyanka

Section Three

Github Repository: Vault Test Case and Repo, KeyChain Test Case and Repo, SecurePasswordField Test Case and Repo

Structural Testing:

Structural testing (also known as whitebox testing, logic driven testing and open box testing) is a type of testing methodology where the test cases are based on the structure of the software system.[6][7] In other words, structural testing tests the implementation of the software or the structure of the code itself. They also come in many forms which includes control flow testing, data flow testing, slice-based testing and mutation testing.

Structural testing is important from a software testing perspective because it will be able to reveal what is missing from the test suite. This can uncover faults that are not typically exposed, such as control flow elements, statements, and branches. The main idea is to measure and increase the structural *coverage* in order to pinpoint what is missing in the codebase, such as unexecuted elements.

Benefits of Structural Testing:

- Enables exhaustive or meticulous testing of the system[6][7]
- Enables early detection of software defects and errors[6][7]
- Identifies dead codes within the implementation[6][7]
- Detects programming issues with respect to best programming practices[6][7]

Test Suite 1: *SecurePasswordFieldTest*

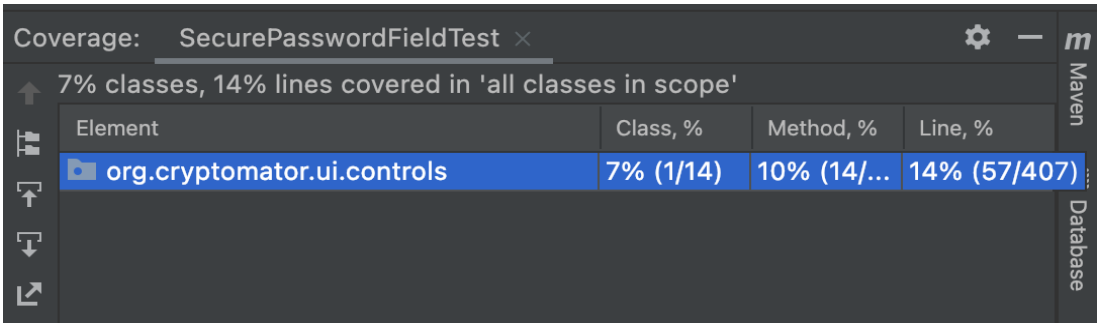

Figure 1: Before Coverage

```
// New Test Cases for Structural (Coverage) - Section 3

@Test
public void testRevealPasswordProperty(){
  pwField.setRevealPassword(true);
//    pwField.revealPasswordProperty();
  Assertions.assertEquals(true, pwField.isRevealPassword());
  pwField.setRevealPassword(false);
//    pwField.revealPasswordProperty();
  Assertions.assertEquals(false, pwField.isRevealPassword());
}

@Test
public void testGetters(){
  Assertions.assertEquals(false, pwField.isContainingNonPrintableChars());
  Assertions.assertEquals(false, pwField.isCapsLocked());
  Assertions.assertEquals(false, pwField.containsNonPrintableCharacters());
  Assertions.assertEquals(new SimpleBooleanProperty().toString(),
pwField.containingNonPrintableCharsProperty().toString());
  Assertions.assertEquals(new SimpleBooleanProperty().toString(),
pwField.capsLockedProperty().toString());
}

@Test
public void testAccessibleAttribute(){
  Assertions.assertEquals(true,
pwField.queryAccessibleAttribute(AccessibleAttribute.EDITABLE));
  Assertions.assertEquals(null,
pwField.queryAccessibleAttribute(AccessibleAttribute.TEXT));
}

// End of Structural Testing (Coverage)
```

Class: SecurePasswordFieldTest (After) - Total Structural Coverage is <u>15 lines</u>

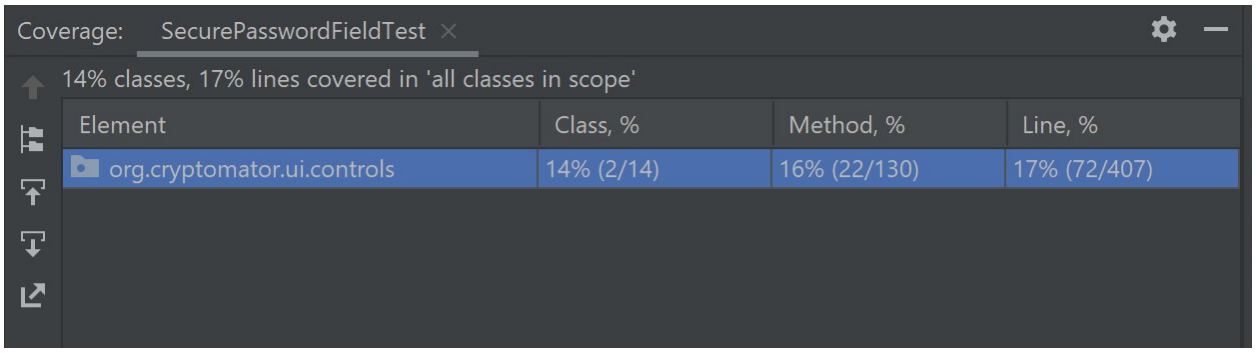

Figure 2: After Coverage

Michael Nguyen
Shih-Lei Chen
A Priyanka

Analysis of Test Suite 1: *SecurePasswordFieldTest*

Originally, Cryptomator's SecurePasswordFieldTest maintained coverage of 7% classes, 10% methods, and 14% lines as shown in Figure 1 above. There are currently undocumented coverage of event handlers, and getters and setters that are important in executing different branches within the class.

After we developed new test cases that targeted uncovered methods, branches, and statements, the total coverage of classes increased to 14%, method coverage increased to 16%, line coverage increased to 17% as described in Figure 2 above. We tested the `revealPasswordChanged()` method by invoking the `isRevealPassword()` boolean that executes this branch of code. The purpose of this test is to determine whether the password revealing functionality is operational within Cryptomator. We also added tests for getters and setters to verify that their functionalities are valid since their values are used to execute other *branches* within the class itself.

Test Suite 2: *VaultSettingsTest*

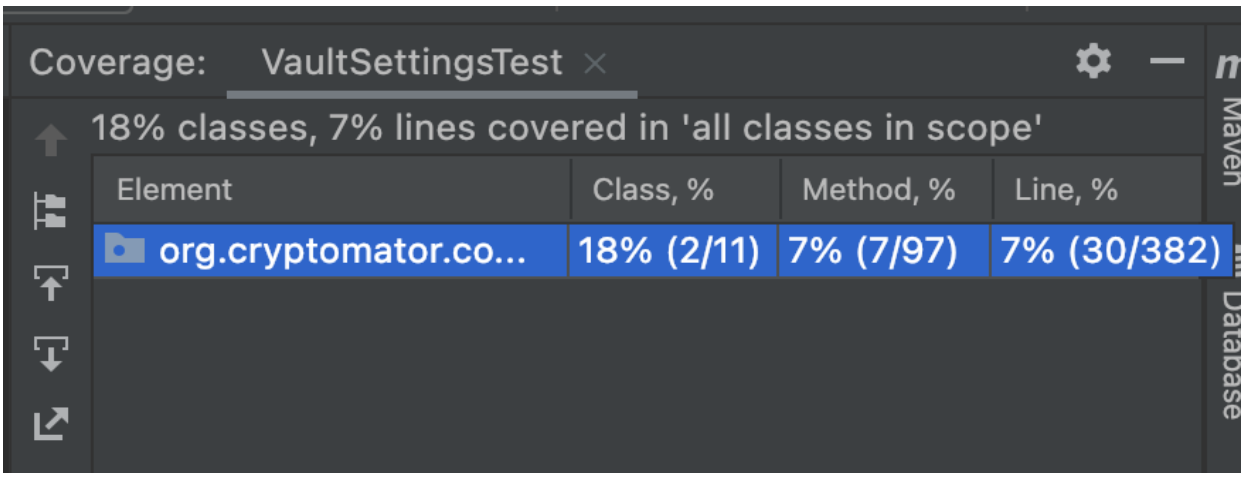

Figure 3: Before Coverage

```
// New Test Cases for Structural (Coverage) - Section 3

@Test
public void testGenerateRandomID(){
  // Test method invoking of subsequent methods within the call
  VaultSettings.withRandomId();
}
```

Michael Nguyen
Shih-Lei Chen
A Priyanka

```
@Test
public void testIdAndPath(){
  VaultSettings settings = new VaultSettings("test_id");
  settings.path();
  Assertions.assertEquals("test_id", settings.getId());
}

@Test
public void testMountFlags(){
  VaultSettings settings = new VaultSettings("test_id");
  Assertions.assertEquals(new SimpleStringProperty("").toString() ,
settings.mountFlags().toString());
}

@Test
public void testRevealAfterMount(){
  VaultSettings settings = new VaultSettings("test_id");
  Assertions.assertEquals(new SimpleBooleanProperty(true).toString(),
settings.revealAfterMount().toString());
}

@Test
public void testCustomMount(){
  VaultSettings settings = new VaultSettings("test_id");
  //settings.mountName();
  Assertions.assertEquals(new SimpleStringProperty(null).toString(),
settings.customMountPath().toString().toString());
}

@Test
public void testGetCustomMount(){
  VaultSettings settings = new VaultSettings("test_id");
  Assertions.assertEquals(Optional.empty(), settings.getCustomMountPath());
}

@Test
public void testUsingCustomMountMath(){
  VaultSettings settings = new VaultSettings("test_id");
  Assertions.assertEquals(new SimpleBooleanProperty(false).toString(),
settings.useCustomMountPath().toString());
}

@Test
public void testAutoLockAndReadOnly(){
  VaultSettings settings = new VaultSettings("test_id");
  Assertions.assertEquals(new SimpleBooleanProperty(false).toString(),
settings.autoLockWhenIdle().toString());
```

```
  Assertions.assertEquals(new SimpleBooleanProperty(false).toString(),
settings.usesReadOnlyMode().toString());
}

@Test
public void testMaxClearTextFilenameLength(){
  VaultSettings settings = new VaultSettings("test_id");
  IntegerProperty maxCleartextFilenameLength = new SimpleIntegerProperty(-1);
  Assertions.assertEquals(maxCleartextFilenameLength.toString(),
settings.maxCleartextFilenameLength().toString());
}

@Test
public void testWinDriveLetter(){
  VaultSettings settings = new VaultSettings("test_id");
  settings.winDriveLetter();
  Assertions.assertEquals(Optional.empty(), settings.getWinDriveLetter());
}

@Test
public void testEquals(){
  VaultSettings settings = new VaultSettings("test_id");
  Assertions.assertEquals(true, settings.equals(settings));
  Assertions.assertEquals(false, settings.equals(null));
}

@Test
public void testUnlockAfterStartup(){
  VaultSettings settings = new VaultSettings("test_id");
  Assertions.assertEquals(new SimpleBooleanProperty().toString(),
settings.unlockAfterStartup().toString());
}

@Test
public void testAutoLockIdle(){
  VaultSettings settings = new VaultSettings("test_id");
  IntegerProperty autoLockIdleSeconds = new SimpleIntegerProperty(30*60);
  Assertions.assertEquals(autoLockIdleSeconds.toString(),
settings.autoLockIdleSeconds().toString());
}

// End of Structural Testing (Coverage)
```

Michael Nguyen
Shih-Lei Chen
A Priyanka

Class: VaultSettingsTest (After) - Total Structural Coverage is 25 lines

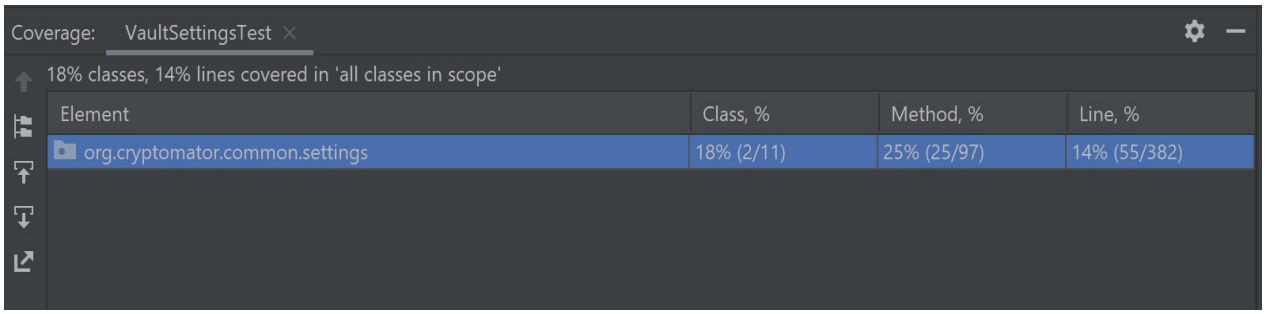

Figure 4: After Coverage

Analysis of Test Suite 2: *VaultSettingsTest*

This test suite dealt with Cryptomator's Vault settings, which is the main security feature. There were only a few test cases that catered to testing different vault settings in their documentation. Cryptomator's VaultSettingsTest maintained coverage of 18% classes, 7% methods, and 7% lines as shown in Figure 3 above. There are currently undocumented coverage of ID generation, auto-locking features, drive letters, mounting, and getters and setters that are important in executing different branches within the class.

After we developed new test cases that targeted uncovered methods, branches, and statements, the total coverage of classes remained stagnant at 18%; however, method coverage increased to 25% and line coverage increased to 14% as described in Figure 4 above. We targeted our tests towards most of the uncovered methods and branches in the class that were important to the functionality of Cryptomator's vault.

This included its auto-lock feature, mounting, drive letters, setters/getters, and ID functionalities. To accomplish this task, we added code that triggered responses in each of the methods by invoking setters that provided functionalities to other methods within the class, as well as testing whether the return values are valid from each statement and branch. For instance, when we added testing for the `getCustomMountPath()` method, the branch can go one of two ways `Optional.ofNullable` or `Optional.empty()`. We verified that statements and branches were executed properly when this test was executed and the result matched our expectations from the assertions.

We also tested setters and getters to determine whether their functionality is valid, such as the `autoLockIdleSeconds()` method. Clearly, an auto-lock must initiate after 30 • 60 seconds. We verified that a valid data type and object is able to effectively pass the test and execute an

auto-lock. Most of the getters and setters were able to effectively execute other *statements and branches* within the class, so that we are able to test other functionalities. As a result of the test cases that we implemented, we improved Cryptomator's VaultSettings and verified that not only their functionality is valid, but their coverage of test cases have been fulfilled.

Test Suite 3: *KeychainManagerTest*

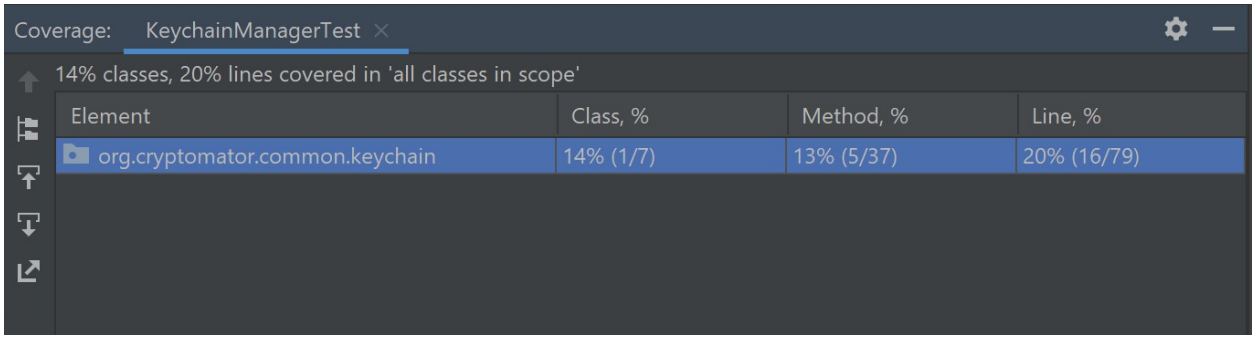

Figure 5: Before Coverage

```
// New Test Cases for Structural (Coverage) - Section 3

@Test
public void testCreateStoredPassphraseProperty() throws
KeychainAccessException {
  KeychainManager keychainManager = new KeychainManager(new
SimpleObjectProperty<>(new MapKeychainAccess()));
  keychainManager.storePassphrase("irvine", "asd");

  // delete the added passphrase
  keychainManager.deletePassphrase("irvine");
  keychainManager.displayName();

  // check if passphrase is stored now
  Assertions.assertEquals(false,
keychainManager.isPassphraseStored("irvine"));
}

@Test
public void testChangePassphrase() throws KeychainAccessException {
  KeychainManager keychainManager = new KeychainManager(new
SimpleObjectProperty<>(new MapKeychainAccess()));
  keychainManager.storePassphrase("irvineMSWE", "remove");

  // change the added passphrase
  keychainManager.changePassphrase("irvineMSWE", "test_phrase");
```

```
  keychainManager.changePassphrase("irvineMSWE",
keychainManager.displayName(), "test_phrase");

  // check if passphrase is still stored (and changed)
  Assertions.assertEquals(true,
keychainManager.isPassphraseStored("irvineMSWE"));
}

@Test
public void testIsSupportAndIsLock() throws KeychainAccessException {
  KeychainManager keychainManager = new KeychainManager(new
SimpleObjectProperty<>(new MapKeychainAccess()));

  Assertions.assertEquals(false, keychainManager.isLocked());
  Assertions.assertEquals(true, keychainManager.isSupported());
}

// End of Structural Testing (Coverage)
```

Class: KeychainManagerTest(After) - Total Structural Coverage is <u>16 lines</u>

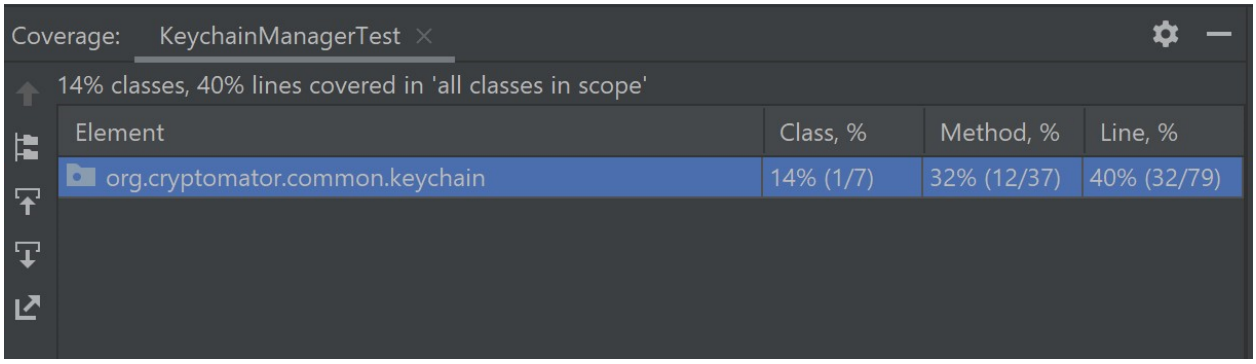

Figure 6: After Coverage

Analysis of Test Suite 3: *KeychainManagerTest*

This test suite dealt with Cryptomator's KeychainManager, which handles the passphrases stored for a given key. Cryptomator's KeychainManagerTest originally had a coverage of 14% classes, 13% methods, and 20% lines as shown in Figure 5 above. There are currently undocumented coverage of passphrase manipulations such as deletions, updates, and support and lock indicators of the key manager.

After we developed new test cases that targeted uncovered methods, branches, and statements, the total coverage of classes remained stagnant at 14% ; however, method coverage

doubled over to 32% and the line coverage doubled to exactly 40% as shown in Figure 6 above. We targeted our tests towards most of the uncovered methods and branches in the class that were important to the functionality of Cryptomator's keychain manager.

In our `testCreateStoredPassphraseProperty()` we test the deletion and passphrase storage check. In order to test these functionalities, we created a new keychain object and stored a new passphrase. We subsequently used the `deletePassphrase()` function to delete the passphrase and used the `isPassphraseStored()` to verify that the passphrase is no longer stored. We also tested methods that change the passphrase in `testChangePassphrase()`. We changed the passphrase using the two different overloaded methods called `changePassphrase()` and used the `isPassphraseStored()` function to assert and verify that the value returned was as expected.

As a result, our established test coverage verified that the statements and branches were executed properly when these tests were executed. The results matched our expectations from the assertion tests, which proves that our test functionalities and class functionalities are working as intended.

Conclusion:

Structural testing is a critical step in the software testing phase. Throughout the structural testing of the three test suites, we achieved a total structural coverage of 56 lines. We developed an understanding of how to improve the coverage of tests by using the coverage testing tool within the IntelliJ IDE. The tool provided us with information on how to target specific sections of classes within Cryptomator that have not been thoroughly covered. Ultimately, our developed test cases provided useful insight on how we exhaustively covered the important classes, methods, and lines that Cryptomator originally excluded.

Michael Nguyen
Shih-Lei Chen
A Priyanka

Section Four

Github Repository: Continuous Integration

Continuous Integration:

Continuous integration or CI is a software development practice where developers merge their codes frequently to a ***main*** central repository. The purpose is to work independently on different features in parallel and commit code to build the main repository every day. [8][9][10] Developers that commit their changes are held accountable for them until the changes they made successfully build and pass tests in the integration machine, which conducts the automated build and test process.

Feedback is a critical component in software development. Therefore, there are monitors in CI that can display what the integration machine is performing. This includes the process of building, running tests, who committed changes, and which tests or builds are passing or failing. The downside of CI is that a slow build and test will be the most challenging obstacle for continuous integration. This type of testing practice is a haste process, the ideal time to build and test is usually around 10 minutes since CI is performed each time there is a new commit or change to the main central repository. This helps in ensuring that the new changes do not impact or affect the existing functionality and the system as a whole is operating and functioning as intended.

Advantages and Purpose of Testing with CI:

- It enables easier predictions of development time
- Transparent communication between developers
- Tests are more reliable due to smaller changes being integrated into the system thus allowing for more positive or negative tests to be performed[2]
- Bugs and defects are detected easier and faster
- It enables fault isolations and thus bugs in the application are limited in scope and are isolated before they affect or impact the entire system[2]

Michael Nguyen
Shih-Lei Chen
A Priyanka

Continuous Integration with Cryptomator:

Environment Tool - Github Actions:

We utilized Github Actions to perform our CI tests on Cryptomator. This CI environment allows us to automate software workflows and makes it easy to build, test, and deploy our code directly from GitHub.

For our project, we created a maven.yml file as shown in Figure 1 below. When we execute the workflow, GitHub Actions builds and runs the test cases of the project. The test cases are able to be executed in the workflow because we list the main branch of our project, which is located in the maven.yml file as *branches: [develop]*.

```yaml
# This workflow will build a Java project with Maven, and cache/restore any dependencies to improve the workflow execution time
# For more information see: https://help.github.com/actions/language-and-framework-guides/building-and-testing-java-with-maven

name: Java CI with Maven

on:
  push:
    branches: [ develop ]
  pull_request:
    branches: [ develop ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2
    - name: Set up JDK 17
      uses: actions/setup-java@v2
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven
    - name: Build with Maven
      run: mvn -B package --file pom.xml
```

Figure 1: maven.yml

Results:

The log snippet provided below describes how GitHub Actions executes and displays the different test cases and their respective outcomes whether they are successful or if there are any indications of failures and warnings.

```
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.032 s - in
org.cryptomator.common.EnvironmentTest$SettingsPath
331[INFO] Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.09 s - in
org.cryptomator.common.EnvironmentTest
332[INFO] Running org.cryptomator.common.SemVerComparatorTest
333[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.044 s - in
org.cryptomator.common.SemVerComparatorTest
334[INFO] Running org.cryptomator.common.settings.VaultSettingsJsonAdapterTest
33501:06:55.197 [main] WARN org.cryptomator.common.settings.VaultSettingsJsonAdapter -
Unsupported vault setting found in JSON: shouldBeIgnored
336[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.087 s - in
org.cryptomator.common.settings.VaultSettingsJsonAdapterTest
337[INFO] Running org.cryptomator.common.settings.SettingsJsonAdapterTest
338[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.051 s - in
org.cryptomator.common.settings.SettingsJsonAdapterTest
339[INFO] Running org.cryptomator.common.settings.VaultSettingsTest
340[INFO] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.045 s -
in org.cryptomator.common.settings.VaultSettingsTest
341[INFO] Running org.cryptomator.common.settings.SettingsTest
342[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.074 s - in
org.cryptomator.common.settings.SettingsTest
343[INFO] Running org.cryptomator.common.vaults.VaultModuleTest
```

The results of running the .yml file is shown in Figure 2 and Figure 3 below. As displayed, GitHub Actions returns a “Build Success” notifying that there were no errors encountered during the process and the test cases successfully passed.

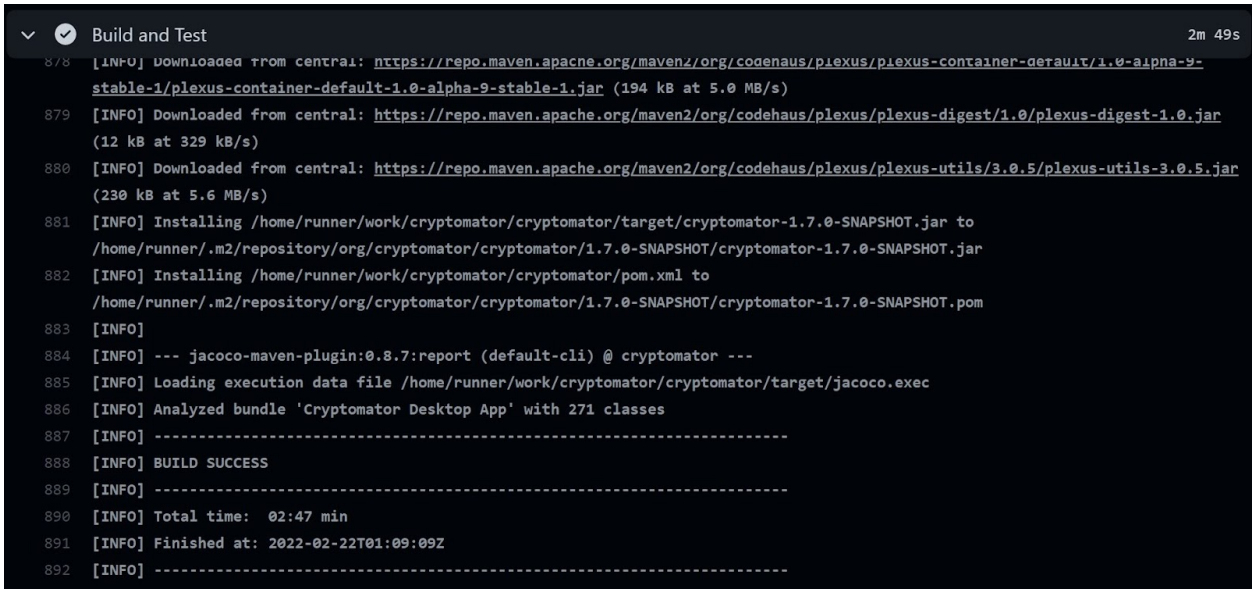

Figure 2: Build Success

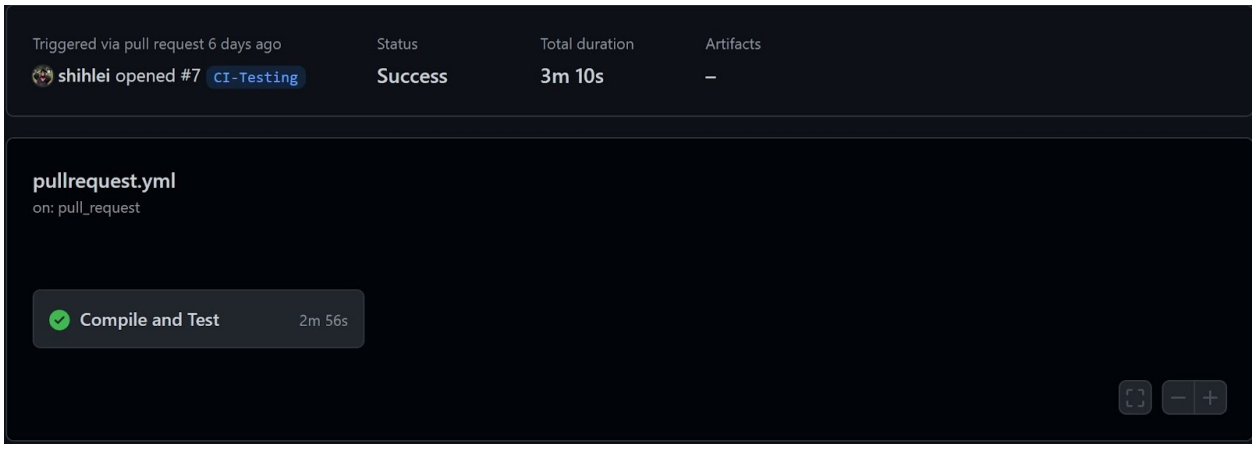

Figure 3: Successful .yml Execution

Continuous Integration Error:

Since we are not the main developers of Cryptomator, the build.yml and release.yml files are blocked through a secret token. In order for these files to successfully build, we require authorization in the form of a SONAR_TOKEN as shown in Figure 4 below. Aside from this error, we were able to write our own .yml file for GitHub Actions to execute, build, and run the test cases in our project.

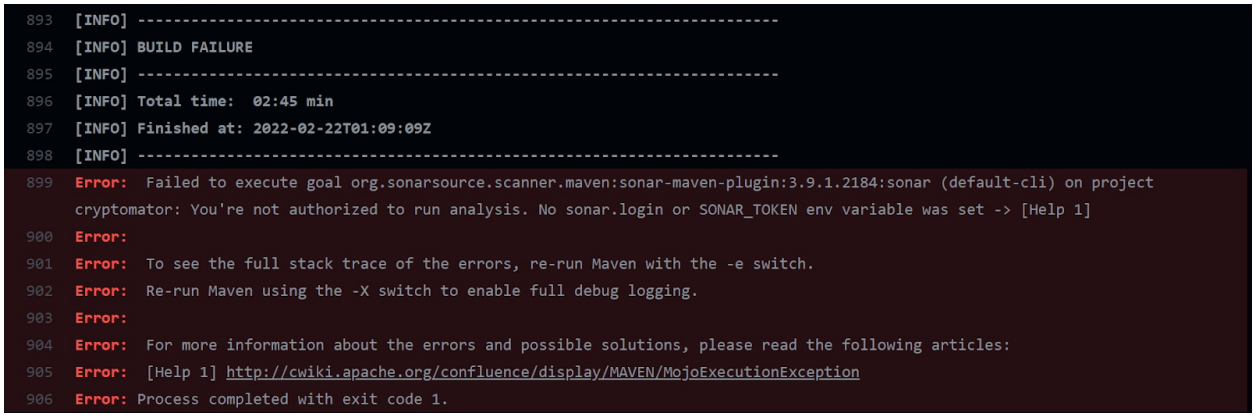

Figure 4: Permission Error

Michael Nguyen
Shih-Lei Chen
A Priyanka

Section Five

Github Repository: Testable Design Test, Testable Design Method, and Mockito Test

Testable Design:

In testable design, each logical piece of code should be quick and simple in order to write test cases. If the system is large, then it would be more difficult to test and maintain. Systems that cannot be changed will be difficult to deliver in Agile development, especially in legacy systems.[11][12] Testable design requires various strategies. The first strategy is to avoid complex private methods. The reason behind this is because of the strict visibility overhead. This results in private methods that can never be tested. The inclusion of complex logic in private methods will produce unknown bugs that cannot be found by testing directly. However, simple private methods are okay to use because testing would be required less frequently. An example are getters and setters.

A second strategic testable design pattern is to avoid static methods since static methods function on the class itself rather than the object. This is because there are scenarios where some functionality may have side effects or require randomness, and including static methods will make it difficult or impossible to test. A third testable design strategy is to caution against using "new" as a hard coded feature. Implementing "new" as hard coded will prevent the object from being stubbed. Fixing this problem requires the object reference to be constructed outside the method and then passed, which is a concept of *dependency injection*.

The fourth strategy of testable design is to avoid developing logic in the constructors since constructors are tough to maneuver around. This is because the constructor of a subclass will always execute at least one of the superclass's constructors. A work around would be to design a simple constructor and have the functionality included in another method. In a nutshell, the goal is to verify that any code that is executed in a constructor is not something that should be included in a test case. Thus, if the code were to be moved to a method, then it can be overridden. The final strategic approach to testable design is to avoid the singleton pattern. The singleton pattern verifies that at any given point, there is only a single object instance of a class. This pattern is useful in some cases; however, the goal is to verify that the functionality being developed does not need to be exchanged out for testing.

Michael Nguyen
Shih-Lei Chen
A Priyanka

Cryptomator Testable Design:

The KeychainManager class includes a *private* method that makes testing impossible since the visibility of the method is within the class itself. This can be proven because the existing test class doest include the method under the coverage as shown from the red bar on the left in Figure 1.

Figure 1: Testable Design (**Private Method**)

This method prevents the BooleanProperty from being returned indicating if a stored passphrase property was created or not. The solution to the problem will require the alteration of the *private* visibility field declared in `createStoredPassphraseProperty()` by changing the visibility to *public*. We implemented a new version of the code as a "dummy" method within the KeychainManager class by making it *public*, so that the original code is not tampered with and we avoid the risk of the functionality being broken. The new method was declared directly under the original *private* method that Cryptomator was designed with. The new implementation is seen below.

Source Code of Updated Implementation:

```
// Section 5: Testable Design

// Original
private BooleanProperty createStoredPassphraseProperty(String key) {
  try {
     return new SimpleBooleanProperty(isPassphraseStored(key));
  } catch (KeychainAccessException e) {
     return new SimpleBooleanProperty(false);
  }
}
```

Michael Nguyen
Shih-Lei Chen
A Priyanka

```
// New implementation
public BooleanProperty createStoredPassphrasePropertyPublic(String key) {
  try {
     return new SimpleBooleanProperty(isPassphraseStored(key));
  } catch (KeychainAccessException e) {
     return new SimpleBooleanProperty(false);
  }
}

// End of Section 5: Testable Design
```

Once we constructed the new testable feature, we added the test cases to the existing testing class KeychainManagerTest. The process is as follows, first we instantiate a new KeychainManager object and store a passphrase called “MSWE.” Once this object is created, we construct a new BooleanProperty that will hold our *expected* result from the *public* method that we designed earlier. Since “MSWE” is stored, the first test case will return *true* when we invoke the public method. However, in the second test case, we create a stored passphrase property called “Irvine” in which the method will search for this stored key. Since this key does not exist initially from the instantiated object, the expected result is *false* and the entire test case passes for both assertions. The test case source code and the result is shown in Figure 2 below.

Source Code for Test Case with Result:

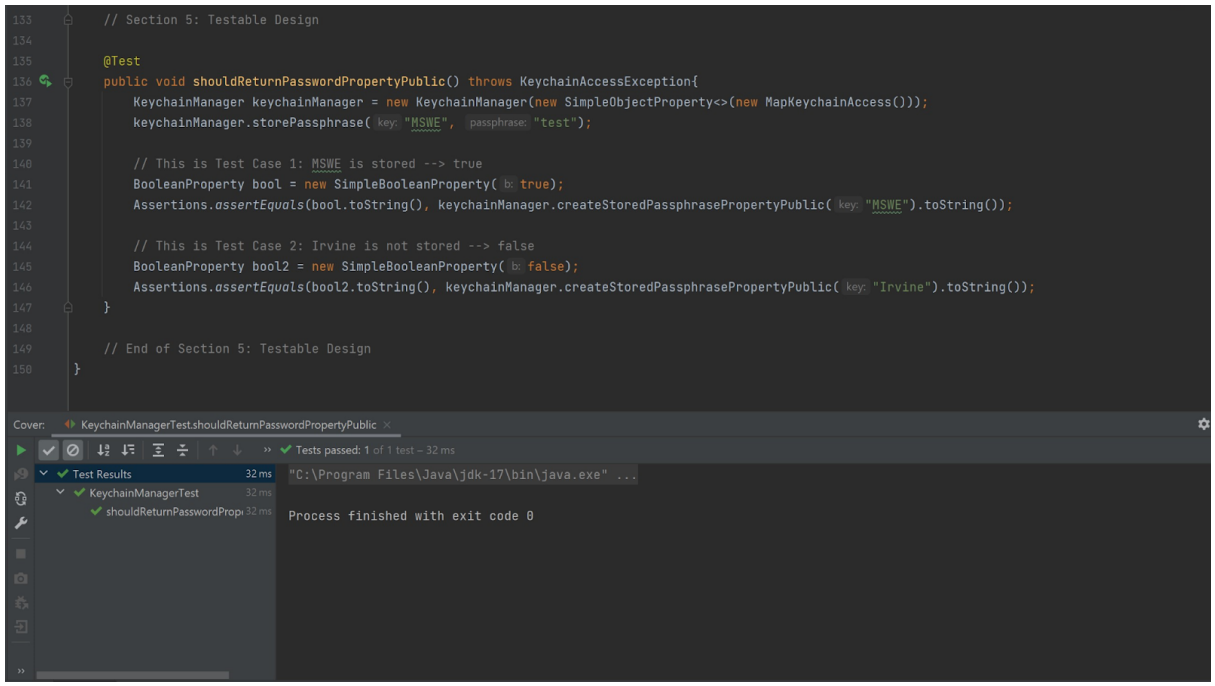

Figure 2: Source Code for Test Case and Result

Michael Nguyen
Shih-Lei Chen
A Priyanka

Mocking:

Mocking is a testing process where a *fake* object is designed over the *real* object and can make a decision whether a unit test passes or fails. This is accomplished by observing the interactions between objects, which is an important property for interaction testing too. In other words, mocking is useful for processing test cases quickly and reliably.[13] From an implementation standpoint, mocking interacts with an object's properties where the main purpose is to section out and focus on the code being tested, rather than the behavior or state of external dependencies.[14]

Furthermore, since external dependencies are no longer a constraint to the unit test, this allows mocking to become much more useful for testing. Without mocking, when a test case fails, it is difficult to pinpoint if the problem or failure is caused by dependencies or source code. Therefore, as a result of mocking, this process can hasten development and testing by isolating failures. Mock tests could also avoid code or test duplication, where the task of verifying methods from our module can be delegated to the mock.[14]

Cryptomator Mocking:

We constructed our test case using Mockito and began with a mock object of the KeychainManager class and developed the respective test cases in the KeychainManagerTest class. The goal of our test case is to target and verify that the *inorder* invocations of the methods are being executed properly along with the assertions. First, we created an object called *keychainManagerMock*, which is a mock object of the KeychainManager class. Next, we invoke the *storePassphrase()* method to store a new passphrase. Using the *when()* function of Mockito, we return *true* when calling the *isPassphraseStored()* function on the *keychainManagerMock.* Afterwards, we assert that the value returned by the function is *true*.

Subsequently, we utilized the *InOrder* class of mockito to verify that methods are executed properly in the order that is intended. To accomplish this task, we constructed an *InOrder* object and invoked the *verify* method that cross-checks that each method from the KeychainManager class is called in the specific order that we designed. Upon executing, the test case passed successfully, which indicates that the methods were properly functioning in the correct order as the source code intended. On a side note, if we swapped the order of the methods being called, the result of the test case would fail. This further proves that we properly designed the unit test to execute the structure of code from the KeychainManager class properly. The test case and successful results are described in Figure 3 below.

Michael Nguyen
Shih-Lei Chen
A Priyanka

Source Code for Test Case with Result:

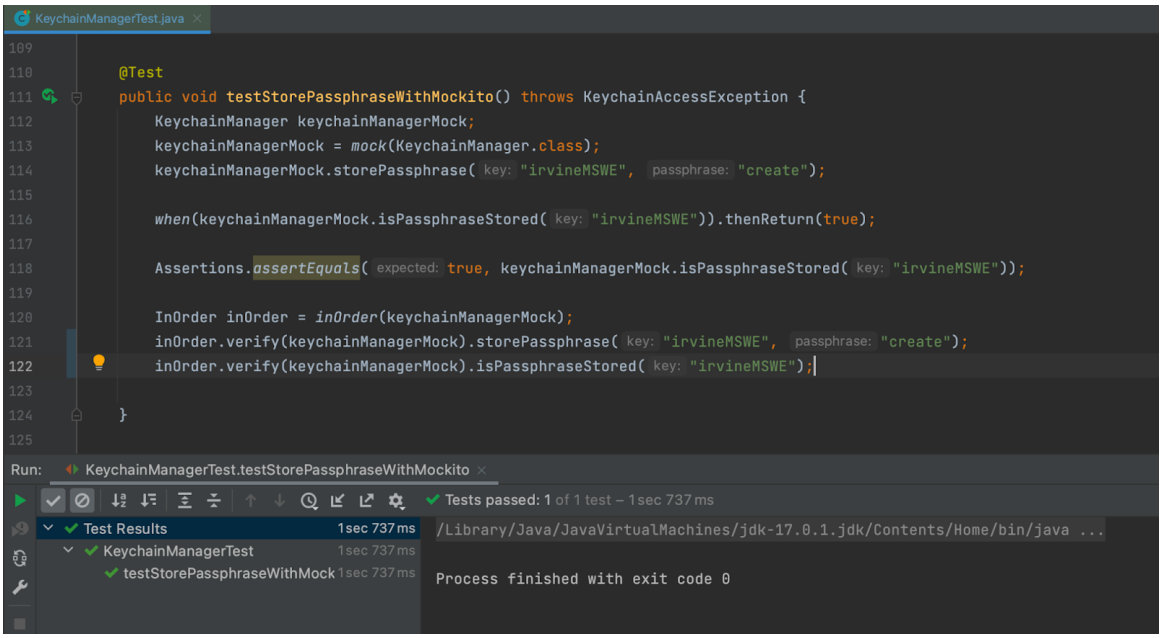

Figure 3: Mockito Test Case with Source Code

Michael Nguyen
Shih-Lei Chen
A Priyanka

Section Six

Github: https://github.com/michaelnguyen26/cryptomator

Static Analyzers:

Static analyzers are a type of debugging tool that accomplishes the task of examining source code before a program is executed.[15] It performs this operation by analyzing groups of code against a set or even multiple sets of coding rules.[15] The main goal and purpose of static analyzers are essentially to perform ***automated*** code reviews instead of ***manual*** code reviews. Feedback is received from the static analyzer which identifies specific coding patterns and displays warnings or other information associated with them if there are any issues.[16]

Uses of Static Analysis:

- Weakness in security.[16]
- Problems with performance[16]
- Standards are not met[16]
- Use of programming constructs are outdated[16]

Plugin Installation:

SpotBugs: *Refer to Figure 1 Below*

1. Preferences → Plugins → MarketPlace and search for the SpotBugs Plugin
2. Install the plugin and restart the IntelliJ IDE.

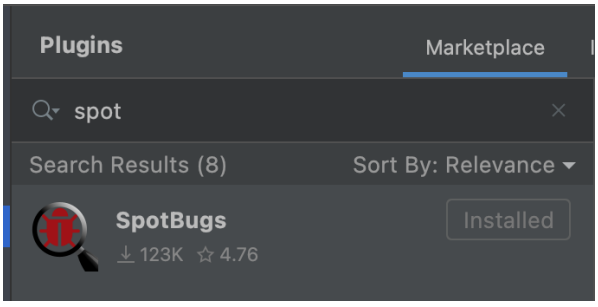

Figure 1: Downloading/Installing the SpotBugs Plugin

Michael Nguyen
Shih-Lei Chen
A Priyanka

CheckStyle: *Refer to Figure 2 & 3 Below*

1. Preferences → Plugins → MarketPlace and search for CheckStyle Plugin

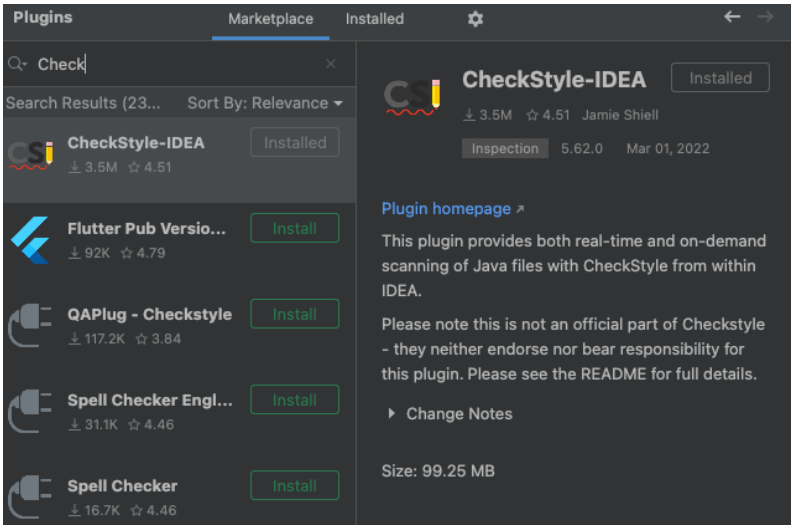

Figure 2: Downloading/Installing the CheckStyle Plugin

2. Install the plugin and restart the IntelliJ IDE.
3. Go to Preferences→ Tools → CheckStyle and click on + to add a configuration file.
4. Add a suitable description and select '*Use a Checkstyle file accessible via HTTP*' option and add the URL to the checkStyle XML file (*https://raw.githubusercontent.com/wso2/code-quality-tools/master/checkstyle/checkstyle.xml*).
5. Select the newly added configuration file and click on '*Apply*'.

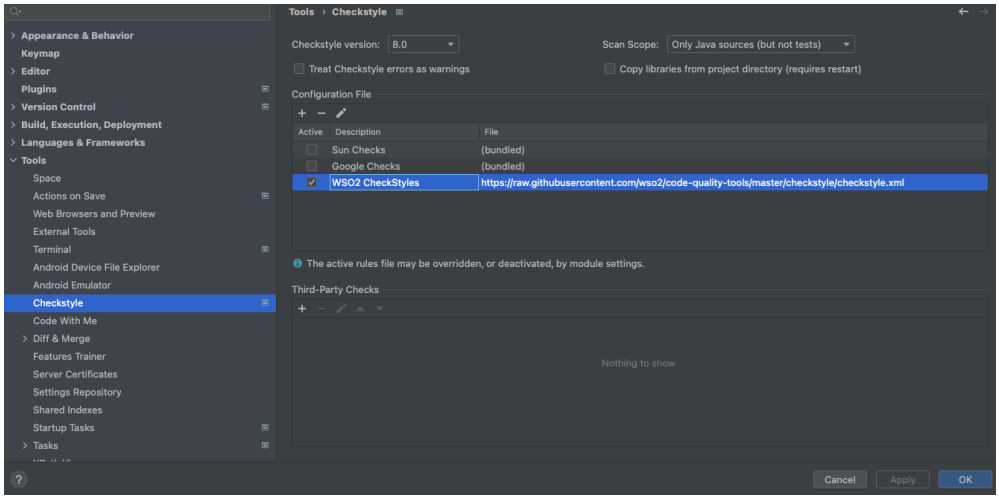

Figure 3: Downloading/Installing the CheckStyle Plugin

Michael Nguyen
Shih-Lei Chen
A Priyanka

Cryptomator Static Analyzer Results:

SpotBugs:

We analyzed the subset of code under the *Settings* folder of Cryptomator by using the SpotBugs tool, the total errors that were displayed are 5 bug items that were contained in the 9 classes.

Inside the *Settings* folder, there are two kinds of errors that were displayed. The first one is *dodgy* code. The error indicated that there might be a null pointer dereference because of the return value of the invoked method that we can see in Figure 5 below. The second error is the *malicious* code vulnerability, this error describes that the code may expose internal representations by incorporating the reference to a mutable object in the SettingsJsonAdapter class and SettingsProvider class by `this.env = env;`. This can be seen in Figure 6 below.

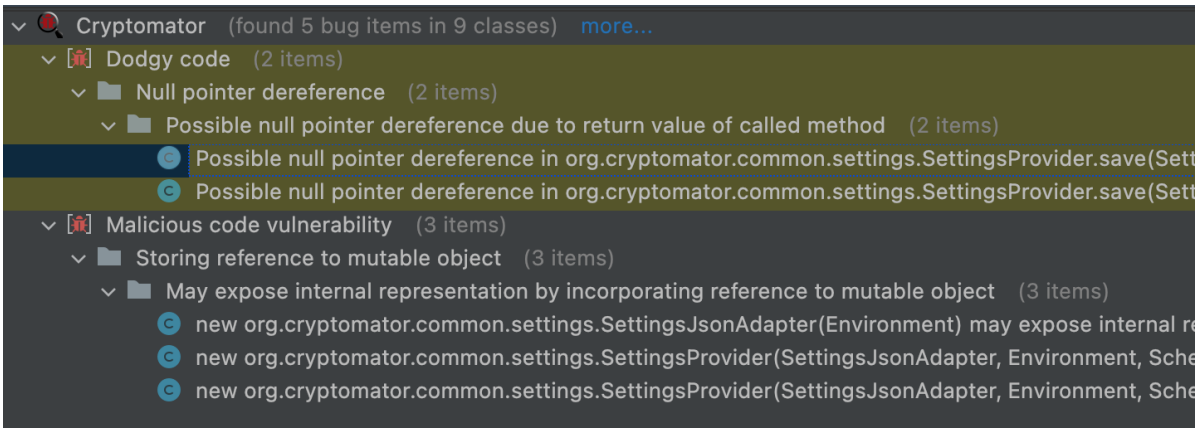

Figure 4: SpotBugs Result for the Settings Folder

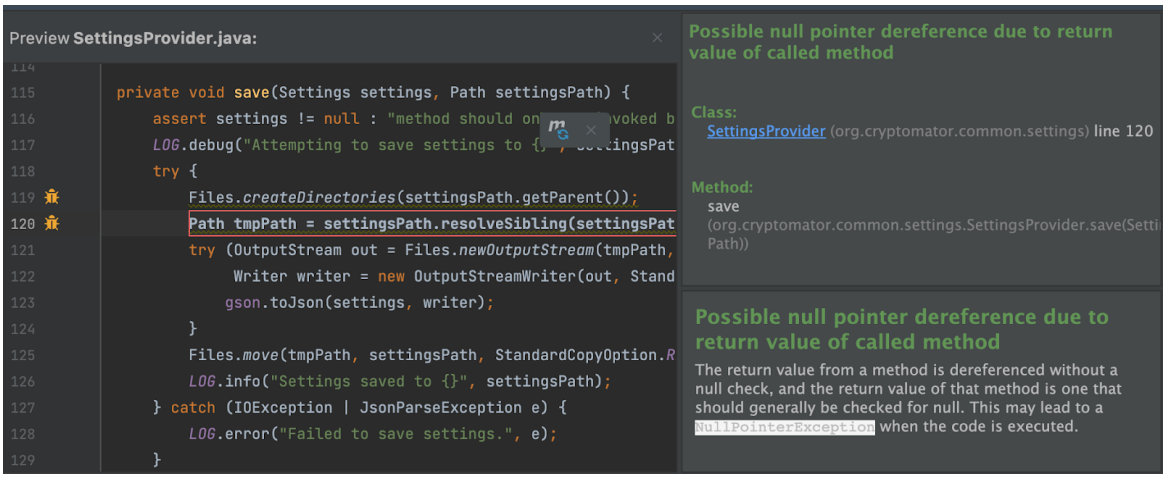

Figure 5: SpotBugs Result for the SettingsProvider.java (dodgy code)

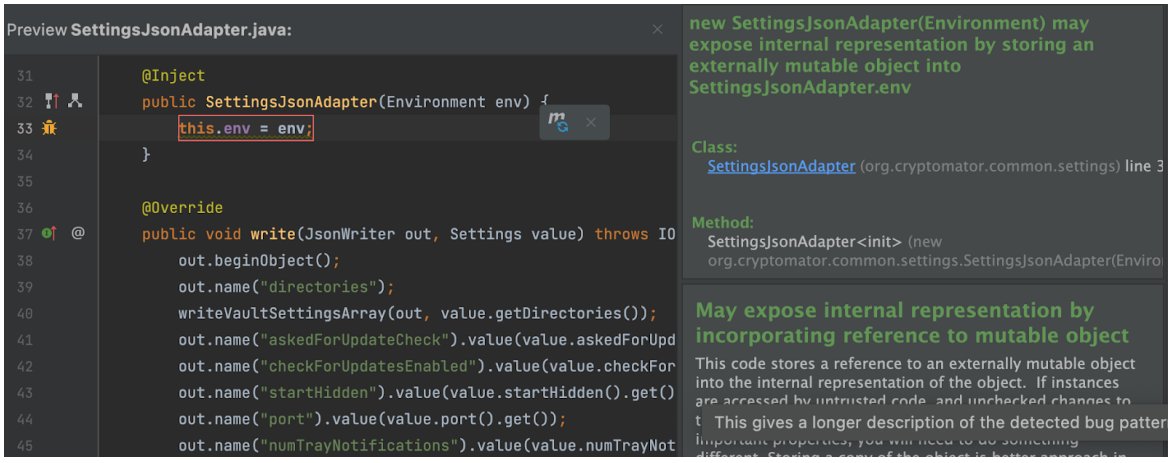

Figure 6: SpotBugs Result for the SettingsJsonAdapter.java (malicious code)

CheckStyle:

We analyzed the subset of code under the *Settings* folder of Cryptomator by using the CheckStyle tool, the total errors that were displayed are 28 error items and 11 warnings that were contained in the 9 classes.

When we take a closer look at the errors in Figure 7, most of the bugs analyzed by CheckStyle pertains to formatting. This includes

- Checkstyle: File contains tab characters (this is the first instance).
- Checkstyle: Line is longer than 120 characters (found 124).
- Checkstyle: 'return' is not preceded with whitespace.

- Checkstyle: '{' is not followed by whitespace.
- Checkstyle: Wrong order for 'javafx.beans.Observable' import.

Most of the above errors detected by the CheckStyle plugin, do not really affect the functionality of the system and are rather trivial in nature.

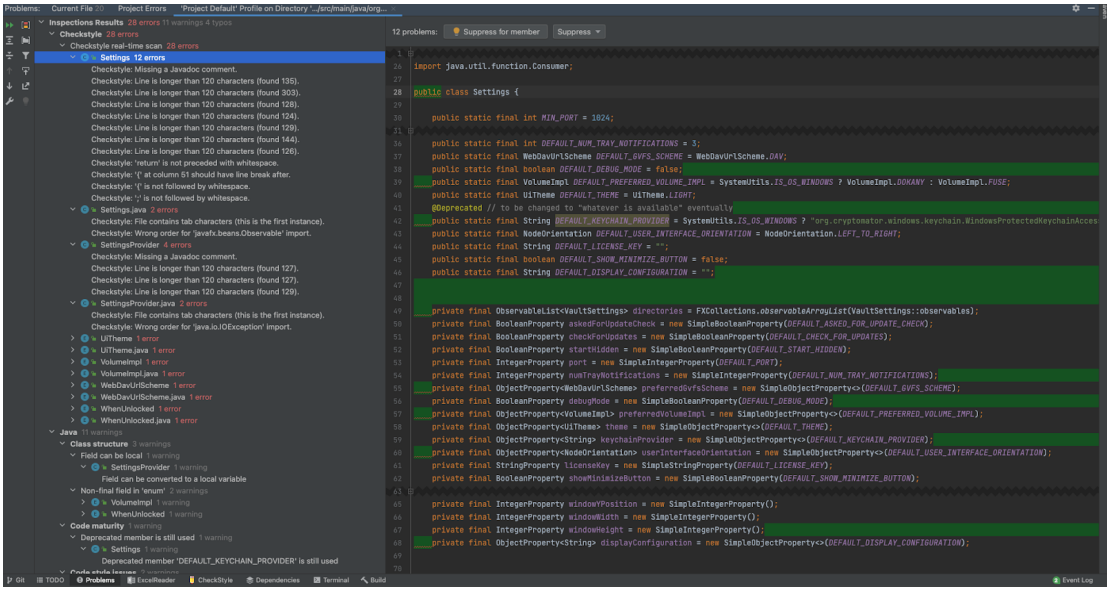

Figure 7: CheckStyle Result for the Settings Folder

The warnings on the other hand, as seen in Figure 8, provide more information about possible bugs and errors. For example, CheckStyle identified unused declarations of methods and variables. In the SettingsProvider.java file, variable *settingsJsonAdapter* has been declared and assigned but never used after assignment. Additionally, the getActionAfterUnlock() method is declared in VaultSettings.java but is never used. Unused declared methods and variables often result in dead code. Although it does not affect the functionality of the software system it is advisable to not have dead codes as it adds unnecessary complexity and impacts the performance.

In addition to the above, CheckStyle also identified probable bugs. For example, in the VaultSettingsJsonAdapter.java file, the *vaultSettings* object is used to set the path of the vault but the path could be null and this could result in an exception.Therefore, this warning provides information that is indicative of a possible error.

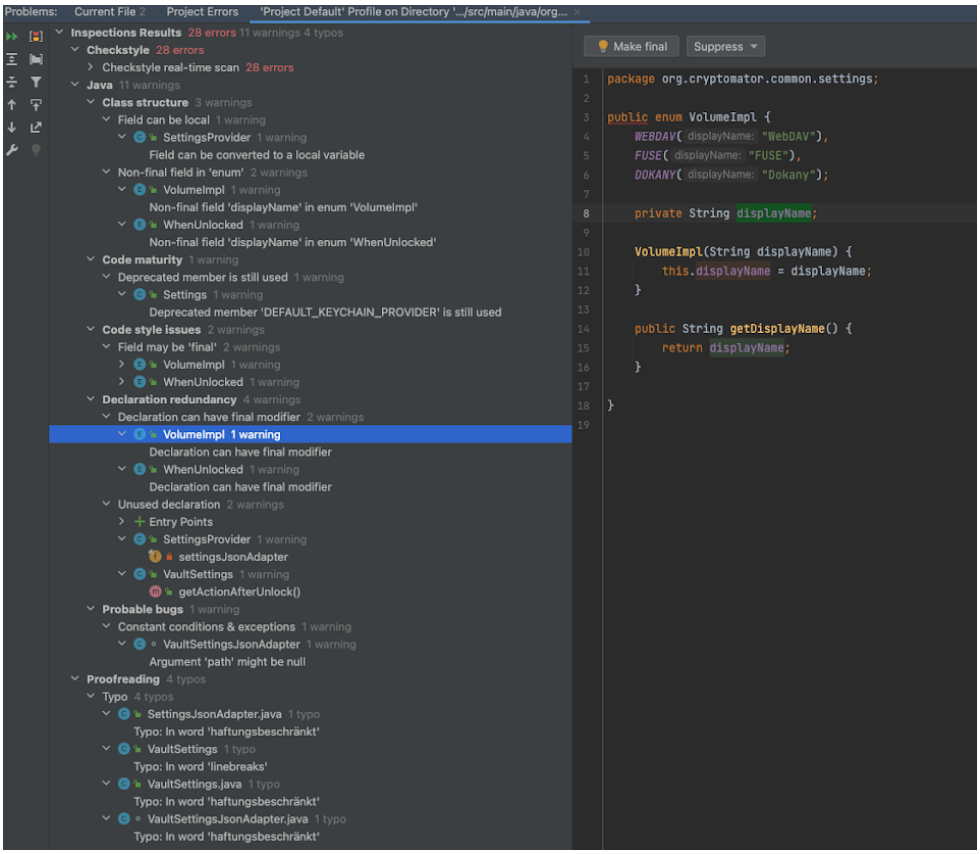

Figure 8: CheckStyle Result for the Settings Folder

Conclusion and Comparison:

We discovered that SpotBugs detects actual problems for Cryptomator. The errors that were displayed using SpotBugs provided information that was useful to analyze, such as the null pointer dereference. On the other hand, CheckStyle contains a lot of redundant errors such as ordering the imports incorrectly and checking the line character count. We agreed that these are not actual errors as it provides no information on whether the program will fail or not, but rather more of a stylistic error. The error information provided in SpotBugs will be useful to discover possible bugs or errors in our code, since a null pointer dereference can actually cause unwanted results. Therefore, the errors provided by SpotBugs are insightful and would be worth considering during the development and testing phase as a static analyzer tool.

We discovered that the error information gathered from both tools did *not* overlap because there is a sharp difference in the tools when determining what is an error and what is not. The errors indicated by CheckStyle are mostly stylistic and only provide warnings and errors that

are related to typos. It is worth noting that CheckStyle indicates some errors related to dead code. SpotBugs targets errors found in each line of code and categorizes them based on the importance such as, *dodgy* or *malicious*. Although they both offer warnings, the tools provide different results and each of their respective warnings differ. The most useful error messages are from SpotBugs since we are able to target where and how the code is behaving and less of the stylistic choices provided from CheckStyle.

Michael Nguyen
Shih-Lei Chen
A Priyanka

References:

Finite State Machines:

1. https://www.einfochips.com/blog/model-based-test-automation-your-key-to-ensuring-the-best-software-quality-quicker/

2. https://www.cambridge.org/core/journals/ai-edam/article/case-similarity-metric-for-software-reuse-and-design/4D205ED5F183E50BA51EA145BDDFD2DE

3. https://www.cs.purdue.edu/homes/apm/foundationsBook/samples/fsm-chapter.pdf

4. https://whatis.techtarget.com/definition/finite-state-machine

Scoring info:

5. https://github.com/dropbox/zxcvbn

Structural Testing:

6. https://www.geeksforgeeks.org/structural-software-testing/

7. https://www.tutorialspoint.com/software_testing_dictionary/structural_testing.htm

Continuous Integration:

8. https://www.atlassian.com/continuous-delivery/continuous-integration#:~:text=Continuous%20integration%20(CI)%20is%20the,into%20a%20single%20software%20project.&text=The%20version%20control%20system%20is,style%20review%20tools%2C%20and%20more

9. https://www.katalon.com/resources-center/blog/benefits-continuous-integration-delivery/

10. https://aws.amazon.com/devops/continuous-integration/

Testable Design:

11. https://livebook.manning.com/book/the-art-of-unit-testing-second-edition/chapter-11/6

12. https://www.scaledagileframework.com/design-for-testability-a-vital-aspect-of-the-system-architect-role-in-safe/

Mocking:

13. https://circleci.com/blog/how-to-test-software-part-i-mocking-stubbing-and-contract-testing/

14. https://devopedia.org/mock-testing

Static Analyzers:

15. https://www.perforce.com/blog/sca/what-static-analysis

16. https://www.securecodewarrior.com/blog/what-is-static-analysis