5 Testing

5.1 Unit Testing

Our team's software internal logic is comprised of two units, instruction insertion unit (IIU) and attack unit (AU).

Instruction Insertion Unit (IIU)

The IIU takes in an attack type, source code, and detection system and outputs an evasive attack. It's responsible for leveraging a machine learning model and instruction power signature dataset to insert x86 instructions into the provided attack source code to create an undetectable microarchitecture attack.

Early in developing the IIU, we will first test its basic functionality of inserting instructions. We will see if, provided any arbitrary C source code, it can adequately insert the correct x86 instructions in the right location without breaking the program. This will be done using a python script which will call the insertion function with various inputs, receive the source output, compare it with the expected value, and compile and test-run the program for errors. After any changes to the unit, we will test it by running the script.

Later in the project, once the instruction power signatures have been collected and the machine learning model has been developed, we will test the unit's ability to identify the instructions and locations needed to create an evasive power signature. This manual effort will require us to provide the unit with our attack codes, make the changes suggested by the unit, run the evasive attacks, and analyze the power signatures with MATLAB scripts and the detection system. We can confirm that it is working correctly by the evasive attack not getting detected and if its attack speed has not significantly slowed down.

Attack Unit (AU)

The AU is responsible for properly executing the attack and collecting statistics about the attack. The testing for this unit will be fairly simple. Again, it will use a python script to call the unit with various attack inputs and see if it operates as expected. We want to see that the unit runs an attack a given number of times with the given amount of space between them, and it collects the amount of data leaked and how long each attack takes to execute.

5.2 Interface Testing

User Interface

Our team's program will feature a simple and user-friendly graphical user interface (GUI). The interface will allow users to do the following tasks:

1. Define the attack type
2. Include the data sets they used.
3. Upload the application's source code used to train the model
4. Upload attack source code
5. Select between different detection models
6. Run evasive power-mimicking attacks

After running an attack, the GUI must display statistics about the attack, including the data leak rate in bits per second and the security systems malware detection certainty for the selected model as a percentage.

UI Testing

1. *Data Type Errors & Error Logging*
    a. We plan to test each one of the inputs fields and make sure each one of the methods handles data type errors correctly as well as to make sure it is logging any errors it encounters. Each method will be tested by using unit testing and testing different scenarios where different input will be passes on each method and check if the outcome is as expected.
2. *UI Component Testing*
    a. All fields, labels, buttons, and other items on the screen will be tested to make sure they are functioning as desired.
3. *Screen Response Testing*
    a. We will be checking screen controls such as colors, fonts, sizes, icons, and others to see how they responds to user's inputs. Make sure each place is easy on the eye of the user and as well to provide and easy experience to the user while using the UI.
4. *Navigation Elements Testing*
    a. We will be testing navigation tool such as scroll bars, navigation bars to ensure smooth transitions while navigating the page, and refresh rate of the page.
5. *Outputs Testing*
    a. *We will test the output given back to users meets the requirements we set on our UI requirements.*

5.3 Integration Testing

Instruction Insertion Unit / Attack Unit Integration

Integration between the IIU and AU is crucial as they make up the two parts of the internal logic. The output of the IIU directly feeds into AU, and hidden errors with the IIU might not be known until it reaches the AU. Testing this integration is very similar to testing the AU and IIU. A python script will provide the IIU with various attack codes and check for correctness. It will see if:

1. The attack runs the appropriate number of times without any errors
2. The correct statistics are outputted
3. The attack runs undetected
4. The attack speed is not significantly slowed down

UI / Internal Logic Integration

The UI / internal logic integration joins the two main parts of the application together. Testing will consist of navigating the UI, calling UI functions, and testing their behavior and outputs with expected values. Like most of the test, this will be done with a python script, but manual effort will be involved with navigating the UI to ensure everything is connected correctly. We expect to see the following:

1. Importing source files works as expected
2. Selecting a detection model successfully loads the internal machine-learning model and data sets
3. Attacks successfully run with the selected type
4. All statistics are shown correctly
5. Each button pressed successfully calls all the necessary parts of the application.

5.4 System Testing

Describe system level testing strategy. What set of unit tests, interface tests, and integration tests suffice for system level testing? This should be closely tied to the requirements. Tools?

System testing will be very similar to the UI / internal logic integration testing, with the main difference being that we will also check if we meet all the functional requirements. These tests will use previously developed testing python scripts and manual efforts of going through each attack code and seeing if we are meeting the requirements. The system test will check to see if:

1. All attack codes are executed below the detection certainty of 20%
2. Power usage never exceeds 2x normal activity
3. No attack exceeds a 5x slower data leak rate compared to its unaltered version

5.5 Regression Testing

How are you ensuring that any new additions do not break the old functionality? What implemented critical features do you need to ensure do not break? Is it driven by requirements? Tools?

Almost all tests use python scripts to check for correctness and can be efficiently run when changes are made. Before any group member pushes their changes, they will run a master testing script that performs all developed tests to ensure nothing is broken.

5.6 Acceptance Testing

How will you demonstrate that the design requirements, both functional and non-functional are being met? How would you involve your client in the acceptance testing?

User Acceptance Testing (UAT):
Our clients will test the detection tool to determine whether it is working for the users correctly and expected.

Expected:

- Client able to download the tool and run the tool on their devices.
- All the buttons on the tool should works on client's site.
- All files can be uploaded.
- When finishing the task, the results should display on the tool.
- All the functionalities are fulfilled the clients' needs.
- Client able to run the tool multiple times without any unexpected errors.
- The power usage should not exceed 2x normal activity, and the attack should not surpass a 5x slower data leak rate on the user's side

Test procedure:

- Client will download the tool on their devices.
- Client will upload different sources files to run the tool.
- Client will use different attack types to run.
- Client will evaluate the correctness of the tool.

5.7 Results

What are the results of your testing? How do they ensure compliance with the requirements? Include figures and tables to explain your testing process better. A summary narrative concluding that your design is as intended is useful.

Figure 1 maps areas of our design to each test and Table 1 shows what area each test evaluates. Testing will be crucial in ensuring that our design works and our attack modifications help to evade the machine learning model. By testing the various attacks, we know that our testing works modularly. When we eventually combine these together into the machine learning model, we know that any issues that occur are due to machine learning model, not the separate attacks. This concludes that our design is useful.

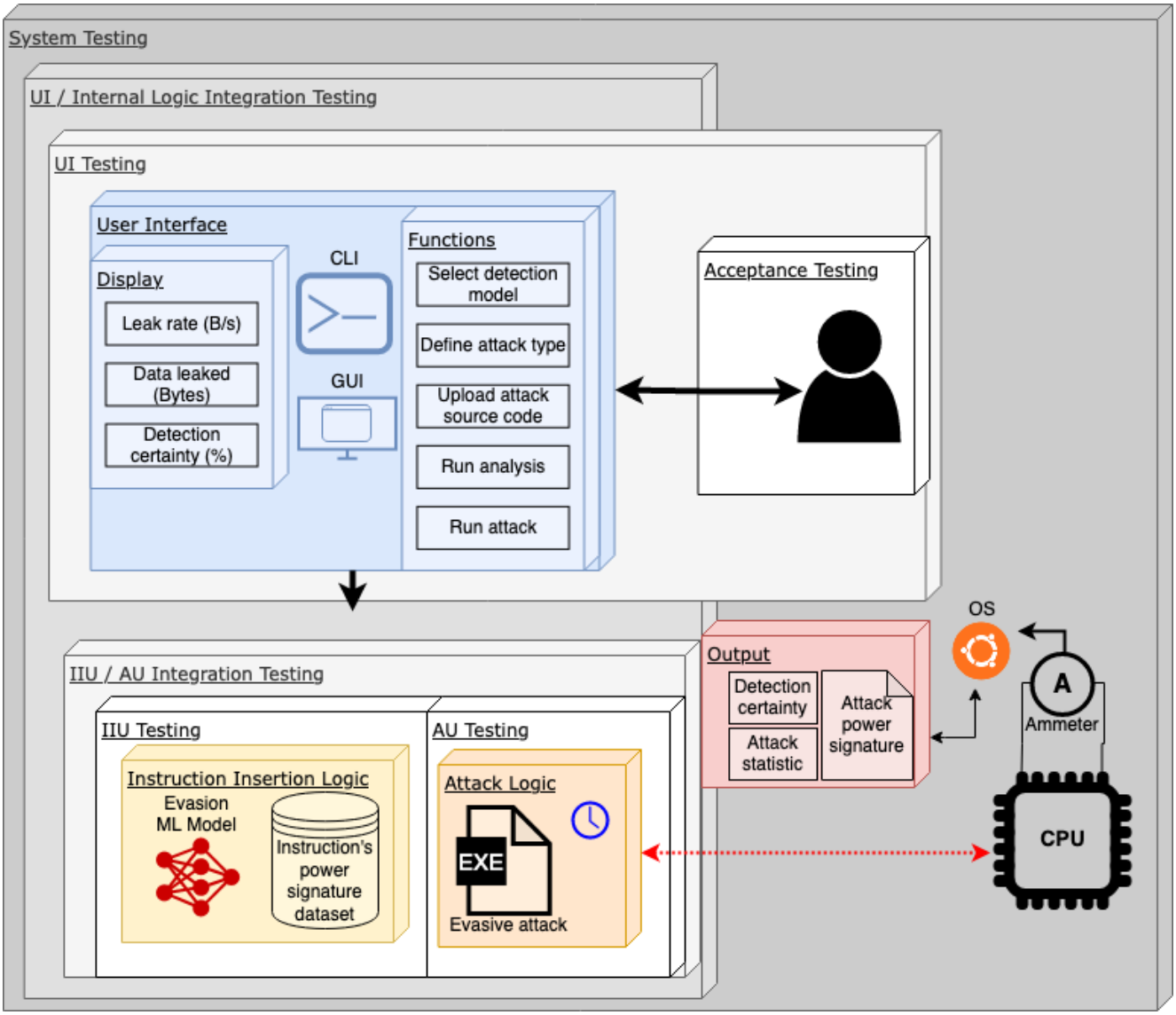

Figure 1: Scope of Testing

Test	UI	Get Statistics	Execute Attack	Correct Profiling	Insert Instructions	ML Model	Power Dataset	Evade Detection
Instruction Insertion Unit (IIU)	x	x	x	✓	✓	✓	✓	✓
Attack Unit (AU)	x	✓	✓	x	x	x	x	x
User Interface (UI)	✓	x	x	x	x	x	x	x
IIU / AU Integration	x	✓	✓	✓	✓	✓	✓	x
UI / Internal Logic Integration	✓	✓	✓	✓	✓	✓	✓	x
System Testing	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Functionality Tested