X-Secure: Ransomware Detection Model

Design Document

Authors:

Rehan Mumtaz (SE-19036)

Kabeer Ahmed (SE-19028)

Wajahat Ahmed (SE-19038)

Moiz Tariq (SE-19060)

Group: 4

This page was intentionally left blank.

Document Revision History

| Date | Version | Description | Author |
| --- | --- | --- | --- |
| 5/25/2023 | 0.0 | Initial draft |  |
| 7/25/2023 | 0.1 | <ex: Added Class Diagram> | John Q. Doe |

This page intentionally left blank.

Contents

[**1 Introduction**](#) 7

[1.1 Purpose](#) 7

[1.2 System Overview](#) 7

[The system's primary components are:](#_kya5ueepik1x) 7

[· Static Analysis Module](#_ys8d457zo5tb) 7

[· Signature-based Malware Detection](#_cp9pmix39x74) 7

[· Dynamic Analysis](#_e7vjg4y5er8y)

[1.3 Design Objectives](#) 7

[1.4 References](#) 8

[1.5 Definitions, Acronyms, and Abbreviations](#) 8

[**2 Design Overview**](#) 10

[2.1 Introduction](#_1t3h5sf) 10

[2.2 Environment Overview](#) 10

[2.3 System Architecture](#) 10

[2.3.1 Top-level system structure of X-Secure:](#) 11

[2.3.2 Static Analysis Sub-system 1](#)2

[2.3.3 Signature Detection Subsystem 1](#)3

[2.4 Constraints and Assumptions 1](#)4

[**3 Structural Design 1**](#)8

[3.1 Creational Design Patterns 1](#)8

[3.1.1 Singleton 1](#_rcnhu9jdoarx)8

[3.1.2 Factory 1](#_s6ddvklnqu42)9

3.1.3 Builder 24

[3.2 Structural Design Patterns 2](#_he4ulwsel5zu)6

[3.2.1 Facade 2](#_1ss74ml3gkt)6

3.2.2 [Adapter Pattern](#_1hof4ukymazh) 30

3.2.3. [Decorator Pattern](#_lr0bwl6heued) 33

3.2.4 Bridge Pattern36

3.3 [Behavioral Design Patterns](#_j1jdu7y2pkjh) 39

3.3.1 [Observer](#_1x48waehn03o) 39

[**4. Dynamic Model 4**](#)2

[4.1. Scenarios 4](#)2

[4.1.2 Scenario: Facade Scenarios 4](#_df5mxgde46pq)2

[4.1.3 Scenario: Adapter Scenario 4](#_k6ms25ewclsl)3

[4.1.3 Decorator Scenarios](#_1jpxhzebiqx3) 44

[4.1.4 Bridge Scenarios](#_wtjz9jyjjmkr) 45

[4.1.5.Singleton Scenario](#_oqndwdfy286o) 46

[4.1.6.Factory Scenario](#_tfhxmufepquh) 47

4.1.7.Builder Scenario 48

4.1.8.Observer Scenario 49

[**5. Supplementary Documentation**](#) 50

# Introduction

The aim of this final year project is to develop a ransomware detection and prevention system using machine learning algorithms. The project will involve collecting and analyzing data on known ransomware strains, as well as developing and training machine learning models to detect and prevent ransomware attacks. The effectiveness of the system will be evaluated using real-world datasets and metrics such as detection accuracy, false positive rates, and response time. The project will contribute to the ongoing efforts to improve cybersecurity and protect individuals and organizations from ransomware attacks.

## *Purpose*

Following are the aims and objectives behind building a ransomware detection and prevention solution:

* Can be used by anyone from the average user to an expert level user such as an IT administrator.
* Open source solution that is capable of countering ransomwares on Windows machines.
* Uses conventional methods such as signature based as well as newer methods such as through Machine Learning and AI for detection and prevention.

## *System Overview*

Our system contains static and dynamic analysis over any program which enters in the computer then performs a machine learning model on it to detect whether it is ransomware or not, then saving the result in a database to have a record of those ransomwares so that similar ransomware can be detected early.

## The system's primary components are:

## · Static Analysis Module: This module employs a deep learning model to analyze the permissions and intents of any running program on Windows Systems. It assesses the nature of the application, distinguishing between benign and malign apps.

## · Signature-based Malware Detection: This module compares the hash of a process against a database of known malicious hashes obtained from reputable malware sample providers such as VirusShare and VirusTotal. This is done using the ML model.

## · Dynamic Analysis: The dynamic analysis of ransomware involves acquiring and executing ransomware samples within a controlled environment. Through behavior monitoring, network traffic analysis, and system monitoring, the ransomware's actions, communication patterns, and system modifications are observed. Additionally, code analysis and artifact examination provide insights into the ransomware's functionality and impact. The findings are documented in an analysis report, which informs the development of mitigation strategies to enhance defense against ransomware attacks.

## *Design Objectives*

The overall system design aims to develop a robust and efficient ransomware detection system using machine learning. It covers a range of functionalities that focus on accurately identifying and mitigating ransomware threats. The design goals encompass both functional and non-functional requirements to ensure a comprehensive and effective solution.

Functional Goals:

Ransomware Detection: The system should be able to analyze network traffic or system behavior in real-time to detect potential ransomware attacks. It should employ machine learning algorithms to identify patterns and indicators of ransomware activity.

Alert Generation: Upon detecting a ransomware threat, the system should generate timely and informative alerts. These alerts should provide relevant details about the detected activity, enabling security analysts to take appropriate actions.

Evasion Techniques Handling: The system should be designed to adapt and respond to various evasion techniques employed by ransomware attackers. It should employ advanced algorithms and methods to counteract obfuscation, encryption, or other evasion mechanisms.

Scalability and Integration: The system design should allow for easy scalability, enabling it to handle large-scale networks and diverse system architectures. It should be capable of integrating with existing security infrastructures, such as firewalls, intrusion detection systems, or security information and event management (SIEM) systems.

Non-functional Goals:

Performance: The design should prioritize high-performance capabilities, ensuring that the ransomware detection system operates efficiently and delivers real-time results. It should minimize latency and processing overhead to maintain the responsiveness of the system.

Usability: The system should have an intuitive and user-friendly interface that simplifies the management and monitoring of ransomware detection. It should provide clear visualizations, informative dashboards, and easy-to-understand reports to facilitate efficient decision-making.

Accuracy and False Positive Reduction: The design should focus on achieving high accuracy in detecting ransomware attacks while minimizing false positives. The system should undergo rigorous testing and validation to ensure its effectiveness in distinguishing between genuine ransomware threats and benign activities.

Robustness and Adaptability: The system should be designed to be robust and resilient against emerging ransomware threats. It should be capable of adapting and learning from new ransomware variants without requiring extensive retraining or manual updates.

Privacy and Security: The design should adhere to privacy regulations and incorporate mechanisms to protect sensitive data and user privacy during the ransomware detection process. It should employ encryption and secure data handling practices to maintain confidentiality and integrity.

## *References*

FYDP Proposal Link: https://docs.google.com/document/d/11FEpRVv1XS\_lnL\_GgO1-GvZp5Kvq6TU8/edit

## *Definitions, Acronyms, and Abbreviations*

AI Artificial Intelligence

PE Portable Executable

ML Machine Learning

RF Random Forest

UI/UX User Interface / User Experience

RaaS Ransomware-as-a-Service

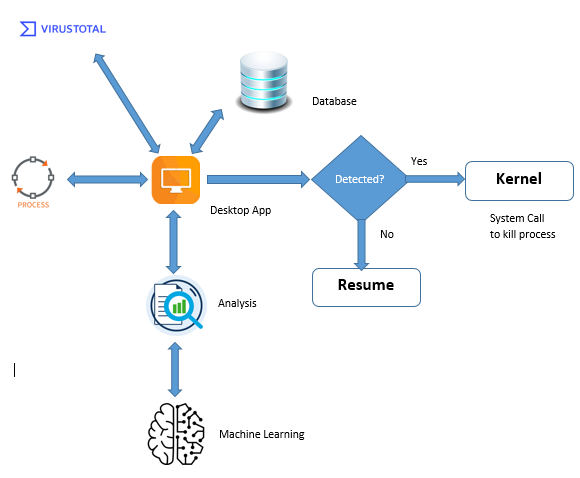
# Design Overview

## *Introduction*

For designing our system we will be using both UML diagrams and design patterns with class diagram to make our system and code more reusable, extensible, maintainable, flexible and fault tolerant, Tools we are using are open source online available like ms word, creatly, genmymodel etc. We will be implementing object oriented design in design patterns, UML diagrams will include, architecture, sequence, activity and use case diagrams.

## *Environment Overview*

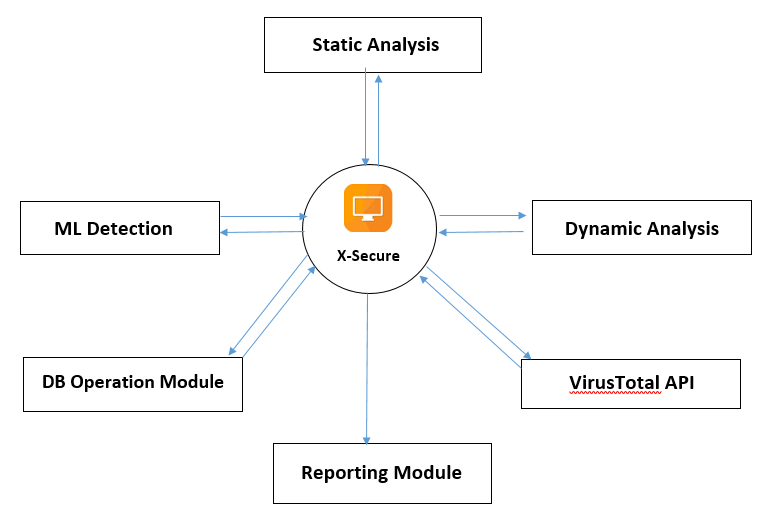
*Our application will reside in windows OS because it is a desktop application, and it require a hardware of atleast 4GB Ram , corei3 processor and windows OS, and how user and system will be interacting with each other, the following environment diagram will show the complete overview of environment and system.*

**

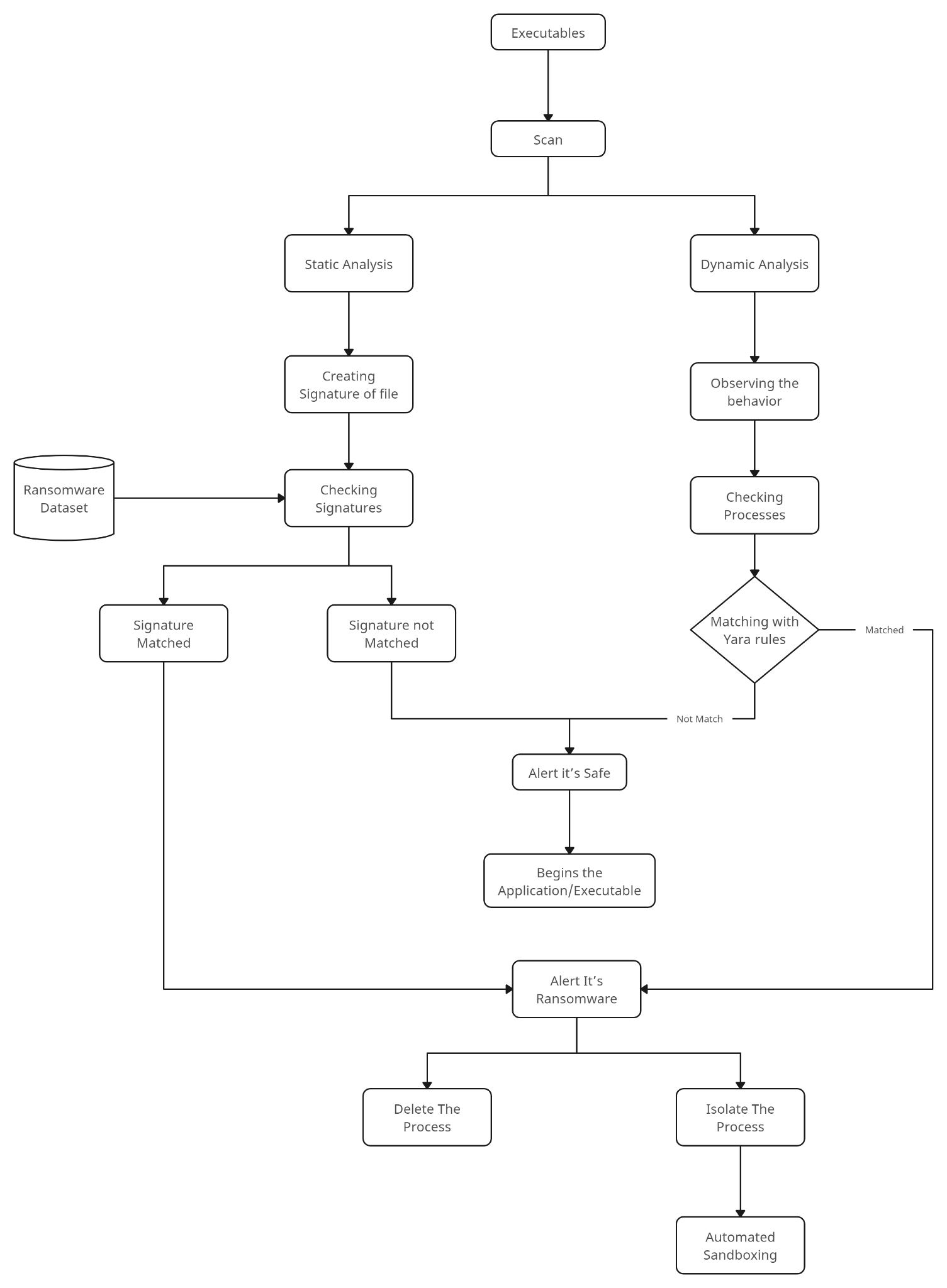
## *System Architecture*

*The X-Secure system architecture is designed as a client-server model, with the client component being the windows based application and the server component being the analysis and ML detection within the client’s system. The desktop application is integrated with ML module and third party analysis tools, enabling seamless communication and data exchange.*

*The following diagram represents the overall architecture of our Ransomware Detection system:*

**

*Figure 1 Block Diagram*



*Figure 2 Architecture Diagram*

The diagram shows:

Executables: These are files that can run programs or commands on a device, such as .exe, .bat, .dll, etc. They can be downloaded from the internet, received as email attachments, or copied from removable media. Some executables may contain ransomware or other malware that can harm the device or its data.

Scan: This step involves scanning the executables for any signs of malware, such as suspicious code, behavior, or network activity. The scanning can be done by antivirus software, firewall, or other security tools.

Analyze: This step involves analyzing the scan results to determine if the executables are benign or malicious. The analysis can be done by using various techniques, such as signature-based detection, heuristic-based detection, sandboxing, etc.

Signature Check: This step involves checking the executables against a database of known malware signatures, which are unique identifiers of malware samples. If the executables match any of the signatures, they are classified as malicious and sent to the next step. If not, they are sent back to the analyze step for further inspection.

Match with Known Ransomware Behavior: This step involves checking the executables against a list of known ransomware behavior patterns, such as file encryption, data exfiltration, ransom note creation, etc. If the executables exhibit any of these behaviors, they are classified as ransomware and sent to the next step. If not, they are classified as benign and allowed to run on the device.

Alert User: This step involves notifying the user that ransomware has been detected on their device and advising them not to pay the ransom or click on any links in the ransom note. The user may also be given options to quarantine or delete the ransomware files or restore their data from backups.

Delete Threat Process: This step involves deleting the ransomware process from the device’s memory and preventing it from running again. This may stop the encryption or decryption of files or the communication with the ransomware server.

Isolate Threat Process: This step involves isolating the ransomware process from the rest of the device’s processes and network connections. This may prevent it from spreading to other devices or accessing any data.

*The following is provided as an example:*

### Top-level system structure of X-Secure:

The top-level system structure of X-Secure encompasses multiple components that work together to provide comprehensive Ransomware detection. The three major components are Static Analysis subsystem, Signature-based subsystem, and Dynamic Analysis subsystem.

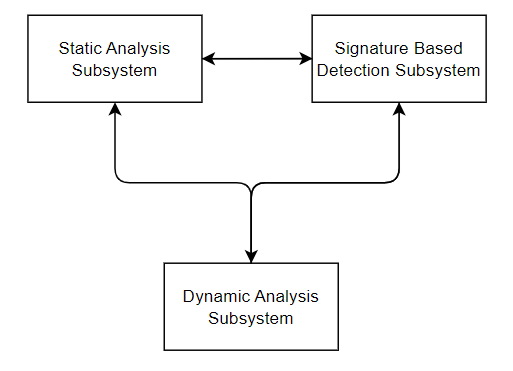


Figure 3 Top Level Structure

*The system consists of three major components, the static analysis subsystem, the signature-based subsystem and the dynamic analysis subsystem. The interaction between all the systems is bi-directional as each subsystem interacts with each other for certain features and provide details to the users.*

### Static Analysis Sub-system

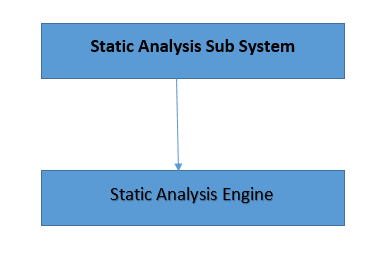
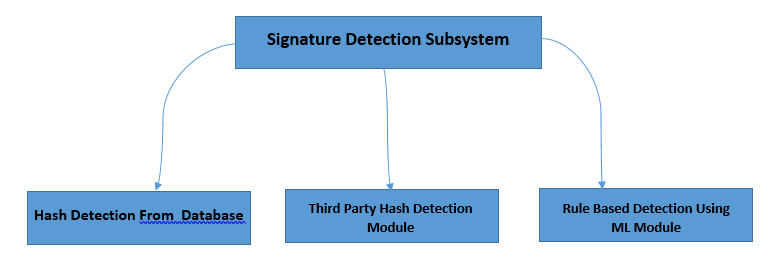


Figure 4 *Static Analysis Sub-system*

The Static Analysis Subsystem is responsible for analyzing the nature of Desktop applications based on static analysis techniques. It comprises the Static analysis engine component:

The Static Analysis Engine is the core component of the Static Analysis Subsystem. It extracts permissions and intents from the programs running on the windows OS and utilizes a Machine Learning Model to classify the applications as either malign or benign. The Static Analysis Engine is integrated into the desktop application, providing real-time analysis results to the user.

### Signature Detection Subsystem



Fig[ure](#figur_emailsubsys)5. Signature Based Subsystem

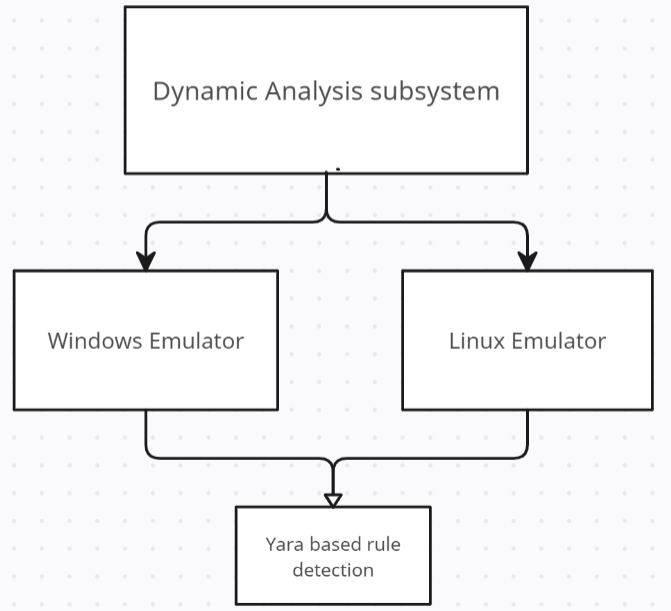
The Signature-based Detection Subsystem focuses on detecting ransomwares based on signatures or hashes. It involves the following components:

Malicious Hash Database: The Malicious Hash Database stores a collection of known malicious hashes obtained from various malware sample providers, such as VirusShare and VirusTotal. The database serves as a reference for comparison during the detection process.

Rule-Based Detection Module: The program is decompiled and the extracted code is analyzed based on some rules and patterns to detect vulnerabilities in the application code, testing is done using Machine Learning Module.

Third Party Hash Detection Module: The Hash Comparison Module compares the hash of a program against the Malicious Hash Database to determine if it matches any known malicious hashes. If a match is found, it indicates the presence of malware. Third Party App like virus total, virus share etc.

2.3.4 Dynamic Analysis Subsystem:



The dynamic analysis of ransomware is a critical process that involves a series of events to understand and analyze the behavior, functionality, and impact of ransomware samples. This analysis helps in developing effective mitigation strategies and enhancing defense mechanisms against ransomware attacks. In this project, the focus is on dynamically analyzing ransomware samples using various techniques and tools.

The scenario begins with the acquisition of ransomware samples from reliable sources or simulated environments. These samples are then executed within a controlled and isolated environment, such as a virtual machine or sandbox. During execution, the behavior of the ransomware is closely monitored using behavior monitoring tools and techniques. This includes tracking file encryption, network communication, and system modifications.

Simultaneously, network traffic analysis is performed to identify communication patterns, command-and-control servers, and any attempts at data exfiltration. System monitoring is also conducted to observe any suspicious or malicious activities, such as unauthorized file access or modifications to the system registry.

Furthermore, the analysis involves examining the artifacts left behind by the ransomware, including dropped files and registry entries. This allows for a deeper understanding of the ransomware's impact on the system. Additionally, dynamic code analysis is performed during execution to uncover the functionality, encryption algorithms, and evasion techniques employed by the ransomware.

Data recovery efforts may be made to retrieve any encrypted files or data using decryption tools or backups. This step helps evaluate the effectiveness of the ransomware's encryption methods.

The findings and observations from the dynamic analysis are documented in a comprehensive analysis report. This report includes details of the ransomware's behavior, network activity, system modifications, and any other relevant findings. Based on the analysis results, mitigation strategies and recommendations are developed to strengthen defense mechanisms against future ransomware attacks.

By following this combined sequence of events in the dynamic analysis of ransomware, valuable insights are gained into the workings of ransomware samples. These insights contribute to the development of effective countermeasures and prevention strategies, bolstering the resilience of systems and networks against ransomware threats.

## *Constraints and Assumptions*

*Constraints for Ransomware Detection using Machine Learning:*

*1. Data Availability: The effectiveness of the machine learning model heavily relies on the availability of relevant and representative data for training. The constraint here is that obtaining a diverse and comprehensive dataset of real-world ransomware samples may be challenging due to the sensitive nature of the data.*

*2. Computational Resources: Training and deploying machine learning models for ransomware detection can be computationally demanding. There may be constraints on the available computational resources, such as processing power, memory, or storage, which can impact the system's performance and scalability.*

*3. Time Constraints: Real-time detection of ransomware requires quick analysis and decision-making. There may be time constraints that need to be considered to ensure timely detection and response, especially in high-traffic environments or critical systems.*

*4. Domain Expertise: Developing an effective ransomware detection system requires expertise in both machine learning and cybersecurity. Constraints may exist in terms of the availability of skilled professionals who can understand the intricacies of ransomware behavior and develop appropriate detection models.*

*Assumptions for Ransomware Detection using Machine Learning:*

*1. Representative Training Data: It is assumed that the available training data sufficiently represents various ransomware attack scenarios, including different variants, techniques, and evasion methods. The assumption is that the training data adequately captures the characteristics and patterns associated with ransomware activities.*

*2. Feature Extraction: The assumption is made that relevant and meaningful features can be extracted from the available data to train the machine learning model effectively. The assumption relies on the availability of experts who can identify and extract features that are indicative of ransomware behavior.*

*3. Continuous Model Updates: It is assumed that the ransomware detection model can be continuously updated to adapt to emerging threats and variations in ransomware techniques. This assumption requires a mechanism for updating the model periodically to ensure its effectiveness over time.*

*4. Network/System Visibility: The assumption is made that the ransomware detection system has sufficient visibility into the network traffic or system behavior to identify potential ransomware activities. This assumption relies on appropriate access permissions and monitoring capabilities within the targeted environment.*

*5. Efficacy of Machine Learning Algorithms: The assumption is that machine learning algorithms, such as deep learning, ensemble methods, or anomaly detection techniques, are effective in capturing and distinguishing ransomware patterns from legitimate activities. The assumption relies on the understanding that these algorithms can learn and generalize from the available data to accurately detect ransomware.*

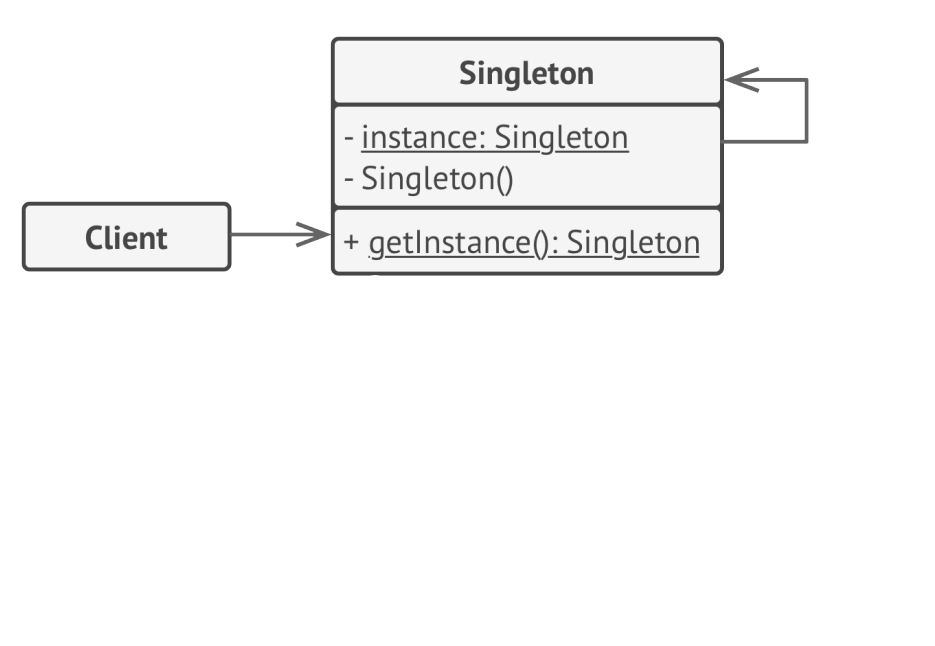
# Structural Design

## Creational Design Patterns

### **Singleton**

#### Design Explanation and Rationale

* + - * 1. Structure



1. Singleton Class: The Singleton class declares the static method getInstance

that returns the same instance of its own class

1. Client: Any point in application from where singleton is called. The Singleton’s constructor is hidden from the client code and the only way it can access the singleton object is by calling the getInstance method.
   * + - 1. Communication and Control Flow

The control flow in the system involves the client code requesting static analysis by invoking methods on the Singleton instance. The Singleton instance handles these requests, performs the necessary analysis, and provides the results back to the client. Communication between the client code and the Singleton instance is direct, without the need for intermediaries.

* + - * 1. Justification

Why Singleton?  
The Singleton pattern is chosen in the Static Analysis Engine to ensure that there is only one instance of the engine throughout the application. This design decision is justified by the need for centralized and consistent behavior of the Static Analysis Engine. By having a single instance, we avoid potential issues related to resource management and ensure that the analysis results are reliable and consistent across different parts of the application.

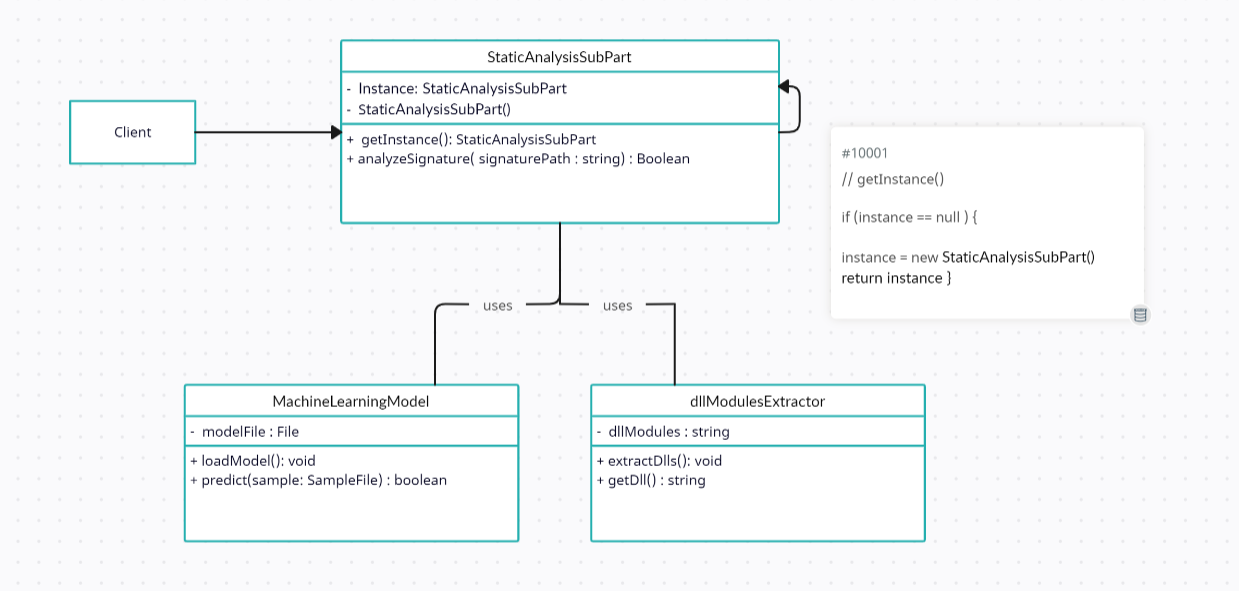
In this class diagram, the StaticAnalysisSubsystem class represents the Singleton class. It contains a private static instance variable instance of type StaticAnalysisSubsystem. The getInstance() method is responsible for returning the single instance of the class and analyzeSignature() to perform Signatureanalysis. The constructor of StaticAnalysisSubsystem is kept private to prevent direct instantiation.

The DeepLearningModel class represents the trained deep learning model used for malware classification. It has an attribute modelFile that stores the file containing the trained model. The loadModel() method is responsible for loading the model, and the predict() method uses the loaded model to predict whether an Signature file is benign or malign.

The dllModulesExtractor class is responsible for extracting permissions from Signature files. It has a method extractdllModules() that performs the extraction process.

The relationships between classes are depicted using associations. The StaticAnalysisSubsystem class uses (has a reference to) the MachineLearningModel and dllModulesExtractor classes.

#### Class Diagram



[Figure 3.](https://docs.google.com/document/d/1XLS578MAeV3A6RLvG8-tsT3ITogO1LN8piT81_gJ9GQ/edit#figur_singleton_design_pattern) Singleton Design Pattern

#### Class Descriptions

The MachineLearningmodel class shows the instance that will access the StaticAnalysissubPart class throughout the analysis. The StaticAnalysissubPart creates only one instance of itself and returns that instance on every call. The analyzeSignature method of StaticAnalysissubPart are used by different requests for analysis code.

#### Classes in the StaticAnalysisSubPart

##### Class: StaticAnalysisSubPart

* Purpose: To ensure that only a single instance of the class can be created and provide global access to that instance.
* Constraints: Only a single instance of the class should exist.
* Persistent: No (created at runtime)

###### Attribute Descriptions

1. Attribute: instance

* Type: Singleton (class reference)
* Description: Stores the single instance of the Singleton class.
* Constraints: None

###### Method Descriptions

1. Method: *getInstance()*

• Return Type: *Singleton (class reference)*

• Parameters: None

• Purpose: Returns the single instance of the Singleton class.

• Algorithm: If the instance is null, create a new instance of the Singleton class. Otherwise, return the existing instance.

• Pre-conditions: None

• Post-conditions: The returned instance is not null.

• Attributes read/used: instance

• Methods called: None

2. Method: analyzeSignature(String signaturePath)

• Return Type: boolean

• Parameters: signaturePath - the path of the signature file to be analyzed

• Return Value: true if the file is classified as benign, false if it is classified as malign

• Pre-condition: The machine learning model and dllmodules extractors are initialized

• Post-condition: None

• Attributes Read/Used: machineLearningModel, dllModulesExtractor

• Methods Called: machineLearningModel.predict(), dllModulesExtractor.extractdlls()

3.1.1.4.2. Class: machineLearningModel

* Purpose: Represents the machine learning model used for malware classification.
* Constraints: None
* Persistent: No

3.1.1.4.2.1. Attribute Descriptions:

1. Attribute: modelFile

• Type: File

• Description: Stores the file containing the trained model.

• Constraints: None

3.1.1.4.2.2. Method Descriptions:

1. Method: loadModel()

• Return Type: void

• Parameters: None

• Purpose: Loads the trained model from the modelFile attribute.

• Algorithm: Loads the model file into memory for future predictions.

2. Method: predict(sample: SampleFile)

• Return Type: boolean

• Parameters: sampleFile - the sample file to be classified

• Return Value: true if the sample is classified as benign, false if it is classified as malign

• Pre-condition: The model is loaded

• Post-condition: None

• Purpose: Predicts whether the given sample file is benign or malign based on the loaded model.

• Algorithm: Uses the loaded model to make a prediction on the given sample file.

3.1.1.4.3. Class: dllModulesExtractor

• Purpose: To extract dll module which are being run by the application and interact with the kernel.

• Constraints: Only a single instance of the dllModulesExtractor class should exist.

• Persistent: No (created at runtime)

3.1.1.4.3.1. Attribute Descriptions

1. Attribute: permissions

• Type: List of strings

• Description: Stores the extracted dllModules required by the application.

• Constraints: None

3.1.1.4.3.2 Method Descriptions

1. Method: extractDlls()

• Return Type: void

• Parameters: None

• Purpose: Extracts the required dll from the application's code.

• Algorithm: Scan the application's code and extract the necessary dll based on predefined rules or patterns. Store the extracted dll in the dll attribute.

• Pre-conditions: None

• Post-conditions: The dll attribute is populated with the extracted dlls.

• Attributes read/used: dll

• Methods called: None

2. Method: getDlls()

• Return Type: List of strings

• Parameters: None

• Purpose: Returns the extracted dlls.

• Algorithm: Return the dlls attribute.

• Pre-conditions: The dlls attribute has been populated with the extracted dlls.

• Post-conditions: The returned list of dlls is not null.

• Attributes read/used: dlls

• Methods called: None

### Factory

#### Design Explanation and Rationale

* + - * 1. Structure

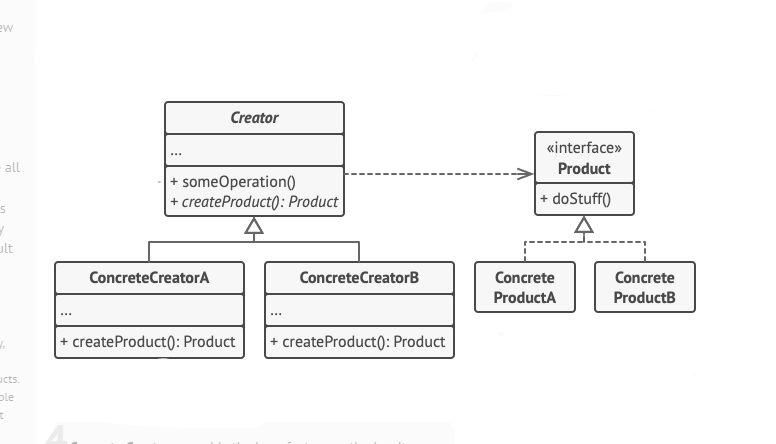


Figure \*. Factory Design Pattern Structure

The Factory Method pattern is implemented in the Dynamic Analysis Subsystem of XSecure. This subsystem is responsible for performing dynamic analysis on desktop Applications using emulators. The Factory Method pattern is employed to create instances of different types of emulators, providing a flexible way to create emulators based on runtime requirements

* + - * 1. Communication and Control Flow

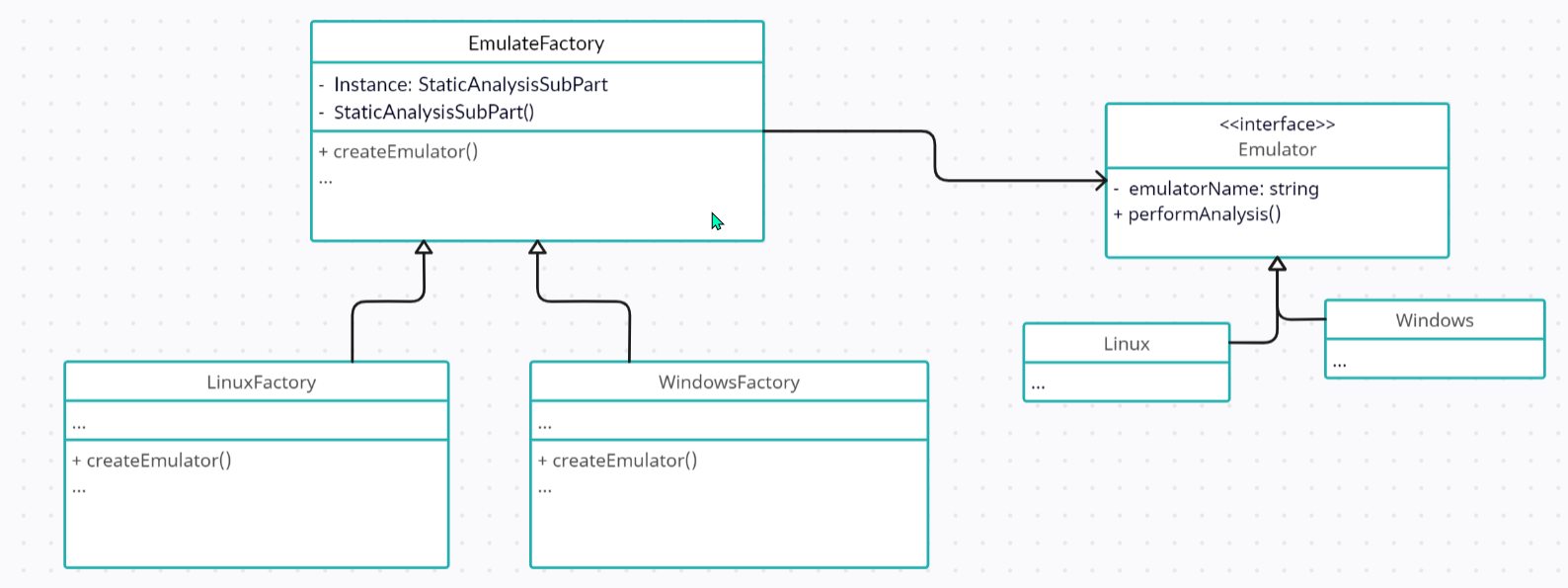
The control flow in the system involves the client code requesting an emulator from the EmulatorFactory by invoking the appropriate factory method. The EmulatorFactory internally determines the concrete emulator type to create based on the runtime requirements or configuration. It then creates and returns an instance of the requested emulator type to the client code.

* + - * 1. Justification

3.1.2.1.3.1 Why Factory?

The Factory Method pattern is chosen in the Dynamic Analysis Subsystem to provide a flexible way to create different types of emulators based on runtime requirements. This design decision is justified by the need for extensibility and modularity. By employing the Factory Method pattern, we can easily introduce new emulator types in the future without modifying the existing subsystem code. The subsystem remains decoupled from the specific emulator implementations, allowing for easy maintenance and enhancement.

#### Class Diagram

Figure . Factory Design Pattern

#### Class Descriptions

#### Classes in the Emulator

##### Creator class:EmulateFactory

* Purpose: The EmulateFactory class is an abstract class that defines the interface for creating Emulator objects. It encapsulates the creation logic and provides a common method for clients to request the creation of specific emulator types.
* Constraints: None
* Persistent: *None*

###### Attribute Descriptions

1. Attribute:factoryName

Type: String

Description: *Represents the name of the factory.*

Constraints: *None*

###### Method Descriptions

1. Method: createEmulator()

Return Type: Emulator

Parameters: None

Purpose: Creates and returns an instance of an emulator.

Algorithm: This method is abstract and should be implemented by the concrete subclasses of EmulateFactory.

Pre-conditions: None

Post-conditions: The returned emulator instance is not null.

Attributes read/used: None

Methods called: None

##### ConcreteCreator class:LinuxFactory

* Purpose: The LinuxFactory class is a concrete subclass of EmulateFactory. It implements the createEmulator() method to create LinuxFactory objects.
* Constraints: None
* Persistent: *None*

Attribute Descriptions

Attribute: *None*

Method Descriptions

1. Method:createEmulator*()*

Return Type: Emulator

Parameters: None

Purpose: Creates and returns an instance of a Linux emulator.

Algorithm: Create and configure a LinuxFactory object according to specific requirements and return it.

Pre-conditions: None

Post-conditions: The returned LinuxFactory instance is not null.

Attributes read/used: None

Methods called: None

##### ConcreteCreator class:WindowsFactory

* Purpose: The WindowsFactory class is a concrete subclass of EmulateFactory. It implements the createEmulator() method to create Windows objects
* Constraints: None
* Persistent: *None*

Attribute Descriptions

Attribute: *None*

Method Descriptions

1. Method:createEmulator*()*

Return Type: Emulator

Parameters: None

Purpose: Creates and returns an instance of a Windows emulator.

Algorithm: Create and configure a WindowsFactory object according to specific requirements and return it.

Pre-conditions: None

Post-conditions: The returned WindowsFactory instance is not null.

Attributes read/used: None

Methods called: None

##### Interface:Emulator

* Purpose: The Emulator interface defines the common interface for different types of emulators.
* Constraints: *None*
* Persistent: *None*

Attribute Descriptions

Attribute: *None*

Method Descriptions

Method: *None*

##### Concrete Product:Linux

* Purpose: The Linux class is a concrete subclass of Emulator. It represents a specific type of emulator used in the dynamic analysis process
* Constraints: *None*
* Persistent: *None*

Attribute Descriptions: None

Method Descriptions:

1. Method: performAnalysis()

• Return Type: void

• Parameters: None

• Purpose: Performs the analysis using the Linux emulator.

• Algorithm: Execute the necessary steps to perform the dynamic analysis using the Linux emulator. This may include installing the application, simulating user interactions, monitoring system behavior, etc.

• Pre-conditions: The Linux emulator instance has been properly configured and initialized.

• Post-conditions: The dynamic analysis process using the Linux emulator has been completed.

• Attributes read/used: None

• Methods called: None

##### Concrete Product:Windows

* Purpose: The Windows class is a concrete subclass of Emulator. It represents a specific type of emulator used in the dynamic analysis process.
* Constraints: *None*
* Persistent: *None*

Attribute Descriptions: None

Method Descriptions:

1. Method: performAnalysis()

• Return Type: void

• Parameters: None

• Purpose: Performs the analysis using the Windows emulator.

• Algorithm: Execute the necessary steps to perform the dynamic analysis using the Windows emulator. This may include installing the application, simulating user interactions, monitoring system behavior, etc.

• Pre-conditions: The Linux emulator instance has been properly configured and initialized.

• Post-conditions: The dynamic analysis process using the Windows emulator has been completed.

• Attributes read/used: None

• Methods called: None

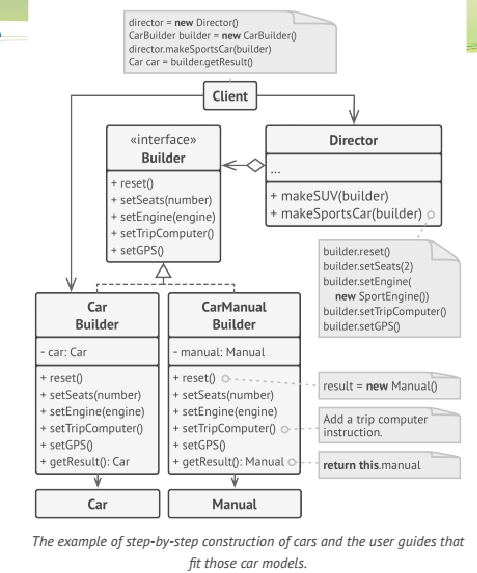
3.1.3 Builder

3.1.3.1 Design Explanation and Rationale

3.1.3.1.1 Structure

The Builder pattern allows you to produce different types and representations of an object using the same construction code.

The given system is structured as:



3.1.3.1.2 Communication and Control Flow

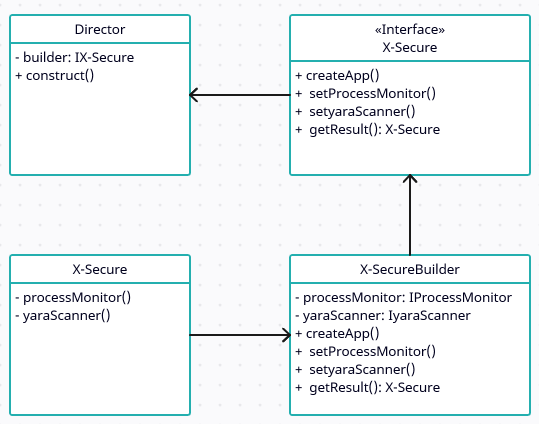
In the system, the Builder pattern facilitates the construction of the communication and control flow components. The Director class manages the construction process by using the interface. The Concrete Builder implements this interface and defines methods to build the communication and control flow. By invoking the Director's construct() method, the components are constructed based on specific requirements, allowing for a flexible and configurable approach to managing ransomware threats.

3.1.3.1.3 Justification

3.1.3.1.3.1 Why Builder?

The Builder pattern is used in software development to simplify the construction of complex objects. It separates the construction logic from the object's representation, allowing for more flexible and controlled object creation. Builders provide step-by-step guidance for constructing objects, enabling the creation of different variations without modifying the underlying code. This pattern is particularly useful when dealing with objects that have multiple configurations or when the construction process is intricate. Overall, the Builder pattern promotes code reusability, maintainability, and enhances the flexibility of object creation.

3.1.3.2 Class Diagram



## Figure Builder Design Pattern

3.1.3.3 Class Descriptions

In the class diagram, the Director class represents the director responsible for constructing the X-Secure object using the interface. The Director takes a concrete builder, and calls the builder's methods to set the required components (monitor, YARA scanner) for the X-Secure object.

3.1.3.4 Classes in the Diagram

3.1.3.4.1 Class: X-Secure

Purpose: To separate the construction of complex objects from their representation.

Constraint: Ensures the same construction process can create different representations. Persistence: The builder pattern does not have inherent persistence, as it focuses on the construction process rather than storage or retrieval of objects.

3.1.3.4.1.1 Attribute Descriptions

1. Attribute: instance

* Type: Builder(class reference)
* Description: Separates the construction of an object from its representation, allowing the same construction process to create different representations.
* Constraints: None

3.1.3.4.1.2 Method Descriptions

No methods.

## Structural Design Patterns

### **Facade**

#### 

* + - 1. Structure:

#### 

Figure \*. Facade Design Pattern Structure

The Facade Method pattern is implemented in the overall system architecture of XSecure. The Facade Method pattern is employed to simplify the design and all the classes are independent of client.

* + - 1. Communication and Control Flow

The control flow in the system involves the client code requesting an facade class for invoking the detection process. The Facade internally determines the concrete objects of other classes to run required functions in different class..Client just need to call facade class to handle all detection operations..

* + - 1. Justification
         1. Why Facade?

When any process starts, our software (client) wants to execute analysis on ransomware, It performs analysis and then it uses ML model to detect and save the results in database. These all three functions are separate and client have to execute them separately so to simplify the task, we bring all those functions together into one facade class, so client will only have to communicate to facade class and all the functions and classes will be implemented by our facade class, client (our software) will only have to interact with one class.

#### 3.2.1.4 Class Diagram

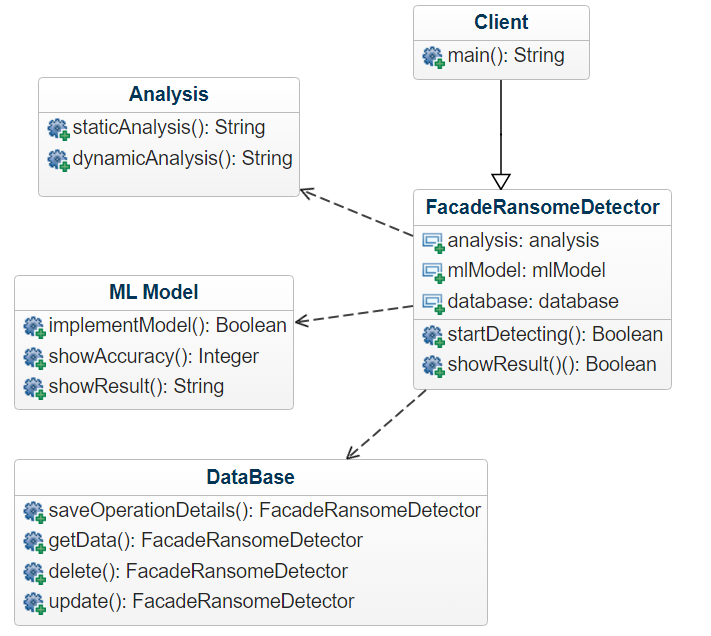


Figure . Facade Design Pattern

#### 3.2.1.5 Class Descriptions:

Class will take the required 3 objects as parameters when we will call the constructor of Facade class, then we have two functions, startDetecting and showResult, which will perform their actions inorder to detect ransomware.

#### 3.2.1.6 Classes in the System

##### 3.2.1.6.1 Creator class:FacadeRansomDetector:

* Purpose: The EmulateFactory class is a concrete class that calls the methods of objects of different class which are passed to it by client, to simplify the connection between client and system classes, It encapsulates the detection logic and provides a common method for clients to request the detection mechanism for specific programs.
* Constraints: None
* Persistent: *None*

###### 3.2.1.6.1.1 Attribute Descriptions

Attribute:analysis

Type: analysis

Description: *object of class analysis*

Constraints: *None*

Attribute:mlModel

Type: mlModel

Description: *object of class analysis*

Constraints: *None*

Attribute:database

Type: database

Description: *object of class analysis*

Constraints: *None*

###### 3.2.1.6.1.2 Method Descriptions

* Method: startDetecting()

Return Type: boolean

Parameters: None

Purpose: Automate the detection task.

Pre-conditions: None

Post-conditions: None

Attributes read/used: all

Methods called: None

* Method: showResult()

Return Type: boolean

Parameters: None

Purpose: To check the status.

Pre-conditions: startDetecting() completed

Post-conditions: None

Attributes read/used: None

Methods called: None

##### 3.2.1.6.2 ConcreteCreator class:Analysis

* Purpose: The analysis a concrete class, to perform static and dynamic analysis.
* Constraints: None
* Persistent: *None*

3.2.1.6.2.1 Attribute Descriptions

Attribute: *None*

3.2.1.6.2.2 Method Descriptions

* Method:staticAnalysis*()*

Return Type: array of object(json)

Parameters: None

Purpose: Perform analysis on program.

Algorithm: Create and configure a analysis object according to specific requirements and return it.

Pre-conditions: None

Post-conditions: The returned json data is not null.

Attributes read/used: None

Methods called: None

* Method:dynamicAnalysis*()*

Return Type: array of object(json)

Parameters: None

Purpose: Perform analysis on program.

Algorithm: Create and configure a analysis object according to specific requirements and return it.

Pre-conditions: None

Post-conditions: The returned json data is not null.

Attributes read/used: None

Methods called: None

##### 3.2.1.6.3 ConcreteCreator class:mlModel

* Purpose: The mlModel class is a concrete class, to give the result based on analysis data
* Constraints: None
* Persistent: *None*

3.2.1.6.3.1 Attribute Descriptions

Attribute: *None*

3.2.1.6.3.2 Method Descriptions

1. Method:startDetection*()*

Return Type: boolean

Parameters: None

Purpose: Check the if the program is legitimate or not.

Algorithm: Create and configure a mlModel object according to specific requirements and return it.

Pre-conditions: None

Post-conditions: None.

Attributes read/used: None

Methods called: None

##### 3.2.1.6.4 ConcreteCreator class:database

* Purpose: The database operations to be performed is done using this.
* Constraints: *None*
* Persistent: *None*

3.2.1.6.4.1 Attribute Descriptions

Attribute: *None*

3.2.1.6.4.2 Method Descriptions

Method:saveOperation*()*

Return Type: boolean

Parameters: None

Purpose: To add analysis details in database.

Algorithm: Create and configure a database object according to specific requirements and return it.

Pre-conditions: None

Post-conditions: None.

Attributes read/used: None

Methods called: None

Explanation And Justification:

When any process starts, our software (client) wants to execute analysis on ransomware, It performs analysis and then it uses ML model to detect and save the results in database. These all three functions are separate and client have to execute them separately so to simplify the task, we bring all those functions together into one facade class, so client will only have to communicate to facade class and all the functions and classes will be implemented by our facade class, client (our software) will only have to interact with one class.

Client (our software) whenever wants to start analyzing first it will create objects of Analysis, ML Model and Database, then it will pass those objects to our FacadeRansomeDetector Class to handle all the tasks itself, client just need to call only one function and all the operations will be handled by Facade class.

Facade has 2 functions, 1) startDetecting() 2)showResult(), client will call startDetecting() method it will handle all the other class objects and call the specific functions that are needed for detecting ransomwares.

Other function showResult() will show the result of ML model, analysis and database and whether the program was detected or not, again the relevant functions of the passed objects will be called here.

### Adapter Pattern

* + - 1. Structure

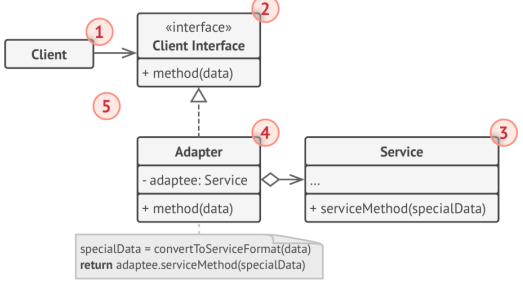


Figure \*. Adapter Design Pattern Structure

The Adapter Method pattern is implemented in the Signature Based Detection Subsystem of XSecure. This subsystem is responsible for performing ML model on analysis dataset . The adapter Method pattern is employed to make both analysis and ml Model interface compatible so that the data from analysis in json format can be converted into excel format which is required by ML model.

* + - 1. Communication and Control Flow

The control flow in the system involves the client code requesting a tabular data from the analysis tool by invoking the appropriate adapter method. The interface of mlModel class will implement the adapter class’s convert method, this method will call data from analysis tool and then will give result back to mlModel;

* + - 1. Justification

3.2.2.3.1 Why Adapter?

Adapter is the only method which is able to make two incompatible interfaces into compatible one by converting data into required format.

#### Class Diagram



Figure . Adapter Design Pattern

#### Class Descriptions

#### Classes in the System

##### Creator class:adapter

* Purpose: The adapter class is an abstract class that defines the interface for converting json objects. It encapsulates the creation logic and provides a common method for clients to request the creation of specific data types into tabular form..
* Constraints: None
* Persistent: *None*

###### Attribute Descriptions

Attribute:adaptee

Type: analysis

Description: *obj of class analysis*

Constraints: *None*

###### Method Descriptions

Method: convertJsonToExcel()

Return Type: null

Parameters: None

Purpose: Creates excelsheet of data,.

Algorithm: This method is abstract and should be implemented by the concrete subclasses of analysis class.

Pre-conditions: None

Post-conditions: Excel sheet should be generated

Attributes read/used: None

Methods called: None

##### Interface:mlModel

* Purpose: The mlModel interface defines the method to implement the convert function of adapter class.
* Constraints: *None*
* Persistent: *None*

Attribute Descriptions

Attribute: *None*

Method Descriptions

Method: convertJsonToExcel()

Return Type: null

Parameters: None

Purpose: Creates excelsheet of data,.

Algorithm: This method is abstract and should be implemented by the concrete subclasses of analysis class.

Pre-conditions: None

Post-conditions: Excel sheet should be generated

Attributes read/used: None

Methods called: None

Explanation And Justification:

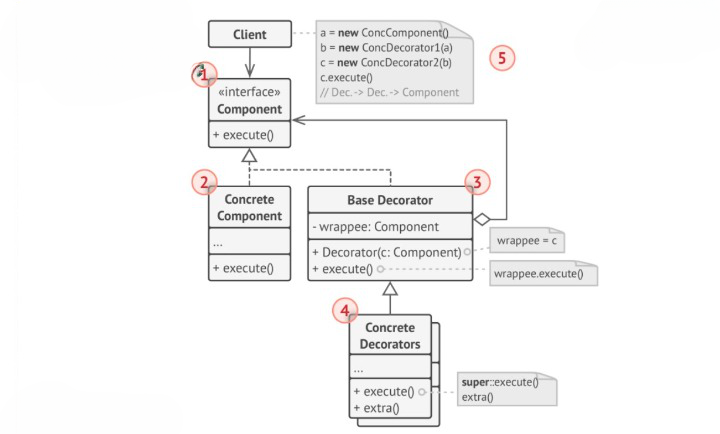
ML model needs data in excelsheet or csv format but the analysis of ransomware from third party tool to extract features give result in json format so for this we use adapter pattern to make both interfaces compatible.

So the static analysis using third party tool will return data using giveAnalysisResultinJson() function, here we have created adapter class which will have composition relationship with analysis class, adapter class will take the result from analysis class and pass the data in convertJsonToExcel(d) function, this will convert data into excel or csv format which is the requirement of our ML Model Class.

ML Model interface will implement the adapter class whenever it needs the analysis data in excel format. ML model class will implement the interface whenever it need data in tabular form.

### Decorator Pattern

* + - 1. Structure of Decorator Pattern



The Decorator design pattern is a structural design pattern that allows behavior to be added dynamically to an object without changing its original class. It provides a flexible alternative to subclassing for extending the functionality of objects at runtime. The pattern involves creating a decorator class that wraps the original object and provides additional functionality by implementing the same interface as the object being decorated. This allows multiple decorators to be stacked on top of each other, each adding its own behavior incrementally. The Decorator pattern promotes the Open-Closed Principle by allowing new functionality to be added without modifying existing code. It also ensures that the client code remains unaware of the decorators and treats the decorated object the same way as the original object. The Decorator pattern is useful in situations where there is a need for dynamic or incremental behavior modification and when subclassing would result in a large number of subclasses. By applying the Decorator pattern, developers can easily enhance the functionality of objects without introducing complexity or breaking existing code.

* + - * 1. Communication and Control Flow

The client sends a request to the DataAnalyzer interface with the data to be analyzed.The DataAnalyzer interface delegates the request to the DataAnalyzerImpl class.The DataAnalyzerImpl class creates and uses the StaticAnalyzer and DynamicAnalyzerDecorator classes to analyze data based on static and dynamic analysis respectively.The StaticAnalyzer class checks the data for syntax errors and potential vulnerabilities.The DynamicAnalyzerDecorator class executes the data and monitors its behavior and performance.The DataAnalyzerImpl class applies the three Yara rules to the data to detect any malicious or suspicious patterns.The DataAnalyzerImpl class sends a response back to the client with the result of the analysis, which includes any errors, warnings, or alerts found by the system.The client receives the final result and decides what to do next based on the final result.

* + - * 1. Justification

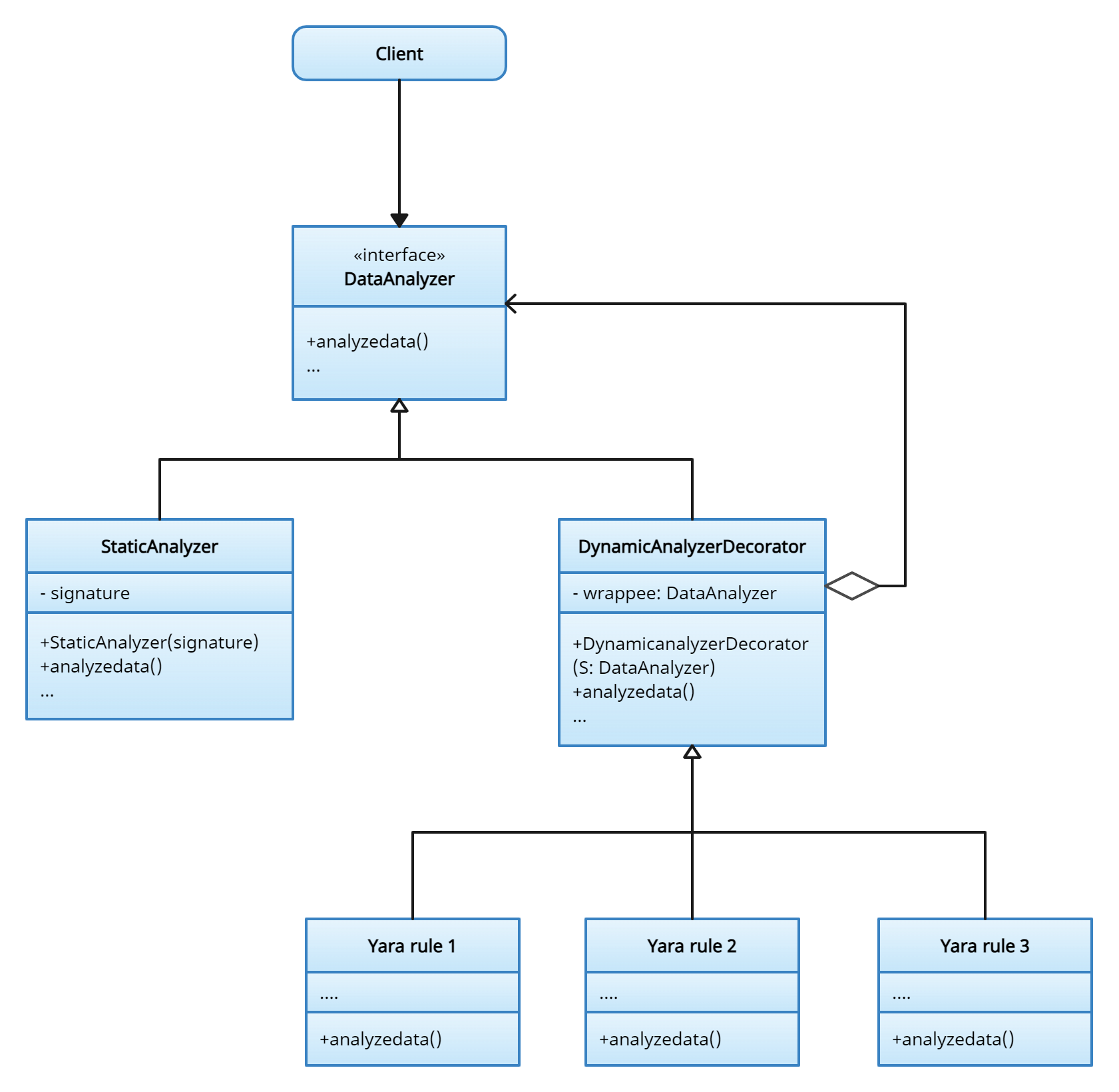
Why Decorator?  
The decorator design pattern is a type of structural design pattern that is used to add new functionality to an existing object without changing its structure. It is also known as the wrapper pattern, as it wraps the original object with a new layer of functionality. It is often useful for adhering to the Single Responsibility Principle, as it allows functionality to be divided between classes with unique areas of concern.

It can add new functionality to the DataAnalyzerImpl class without modifying its existing code or structure. For example, if we want to add a new analysis technique, such as a NetworkAnalyzer, we can create a new decorator class that implements the DataAnalyzer interface and wraps the DataAnalyzerImpl object.

It can avoid creating a large number of subclasses for each combination of DataAnalyzer and its sub-analyzers. For example, if we want to add a new sub-analyzer, such as a NetworkAnalyzer, we do not need to create a new subclass of DataAnalyzerImpl for each existing sub-analyzer combination.

It can increase the modularity, extensibility, and reusability of the software architecture. By separating the analysis logic into different classes, we can easily modify or replace one class without affecting the others. We can also introduce new analysis techniques or support new data types by adding new classes that implement the same interface. Furthermore, we can reuse the existing classes in other contexts or systems that require similar functionality.

* + - 1. Data Analyzer Class



[Figure](#figur_facade) 4. Decorator Design Pattern

#### Class Descriptions

#### Classes in the System

##### Class: Client

* Purpose: The Client class represents the user or system that uses the RansomwareDetector interface to analyze a program for ransomware threats.
* Constraints: None
* Persistent: *None*

###### Attribute Descriptions

Attribute: *None*

###### Method Descriptions

Method: requestAnalysis(data)

Return Type: Result

Parameters: data - a Data object that represents the data to be analyzed.

Purpose: Sends a request to the DataAnalyzer interface with the data to be analyzed and receives the final result.

Algorithm: Invoke the analyzeData() method of the DataAnalyzer interface with the data as an argument and return its result.

Pre-conditions: The data is not null.

Post-conditions: The returned result is not null.

Attributes read/used: None

Methods called: analyzeData() of DataAnalyzer interface

Explanation And Justification:

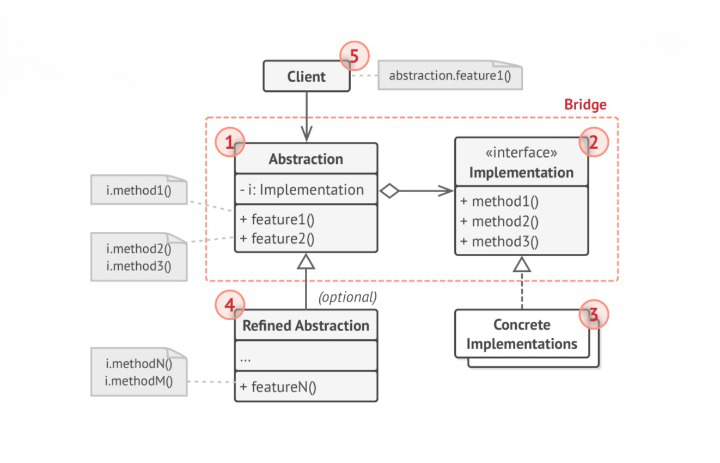
In Dynamic Analysis the file or process are analyzed on the basis of different yara rules.In this design, DataAnalyzer serves as the base class that contains common functionality and methods for dynamic analysis. The StaticAnalyzer class inherits from DataAnalyzer to specialize the analysis techniques for ransomware detection and prevention. The DynamicAnalyzer class further extends DataAnalyzer to incorporate YARA rule application during dynamic analysis.

so the decorator extends the dynamic analysis by adding more yara rules in the analysis so that in future if any more yara rules are created it can easily be added in dynamic analysis.

This design allows for flexibility in combining and extending different Yara rules while adhering to the principles of composition and inheritance.

### Bridge Pattern

* + - 1. Structure of Bridge Pattern



The Bridge design pattern is a powerful structural design pattern that enables the separation of an abstraction from its implementation. By decoupling the abstraction and implementation into separate class hierarchies, the Bridge pattern promotes loose coupling and flexibility in the system's design. The pattern consists of an Abstraction class that defines the abstract interface and maintains a reference to the implementation object through a common interface. Subclasses of the Abstraction, known as RefinedAbstractions, can extend or add additional functionality to the abstraction. On the other hand, the Implementation interface declares the operations that concrete implementation classes must provide. Concrete implementations provide specific implementations of the operations defined by the Implementation interface. The Bridge pattern allows the abstraction and implementation to vary independently, enabling easy extension or modification of either part of the system. This pattern is particularly useful when there is a need to switch or add different implementations without impacting the existing codebase. By employing the Bridge pattern, software designers can achieve a more flexible, maintainable, and modular system architecture.

* + - * 1. Communication and Control Flow

The client sends a request to the RansomwareDetector interface with the program to be analyzed.The RansomwareDetector interface delegates the request to the RansomwareDetectorImpl class.The RansomwareDetectorImpl class creates and uses the SignatureDetector and BehaviorDetector classes to detect ransomware based on static and dynamic analysis respectively.The SignatureDetector class checks the program for known ransomware signatures, such as file extensions, encryption keys, or ransom notes.The BehaviorDetector class monitors the program’s actions, such as file access, network activity, or registry changes.The RansomwareDetectorImpl class combines the results of the SignatureDetector and BehaviorDetector classes and sends them back to the client.The client also receives the results from the WindowsDetector and LinuxDetector classes, which are created and used by the RansomwareDetectorImpl class.The WindowsDetector and LinuxDetector classes scan and match the program against a database of known ransomware hashes for each operating system.The client receives the final result, which indicates whether the program is safe, suspicious, or malicious.

* + - * 1. Justification

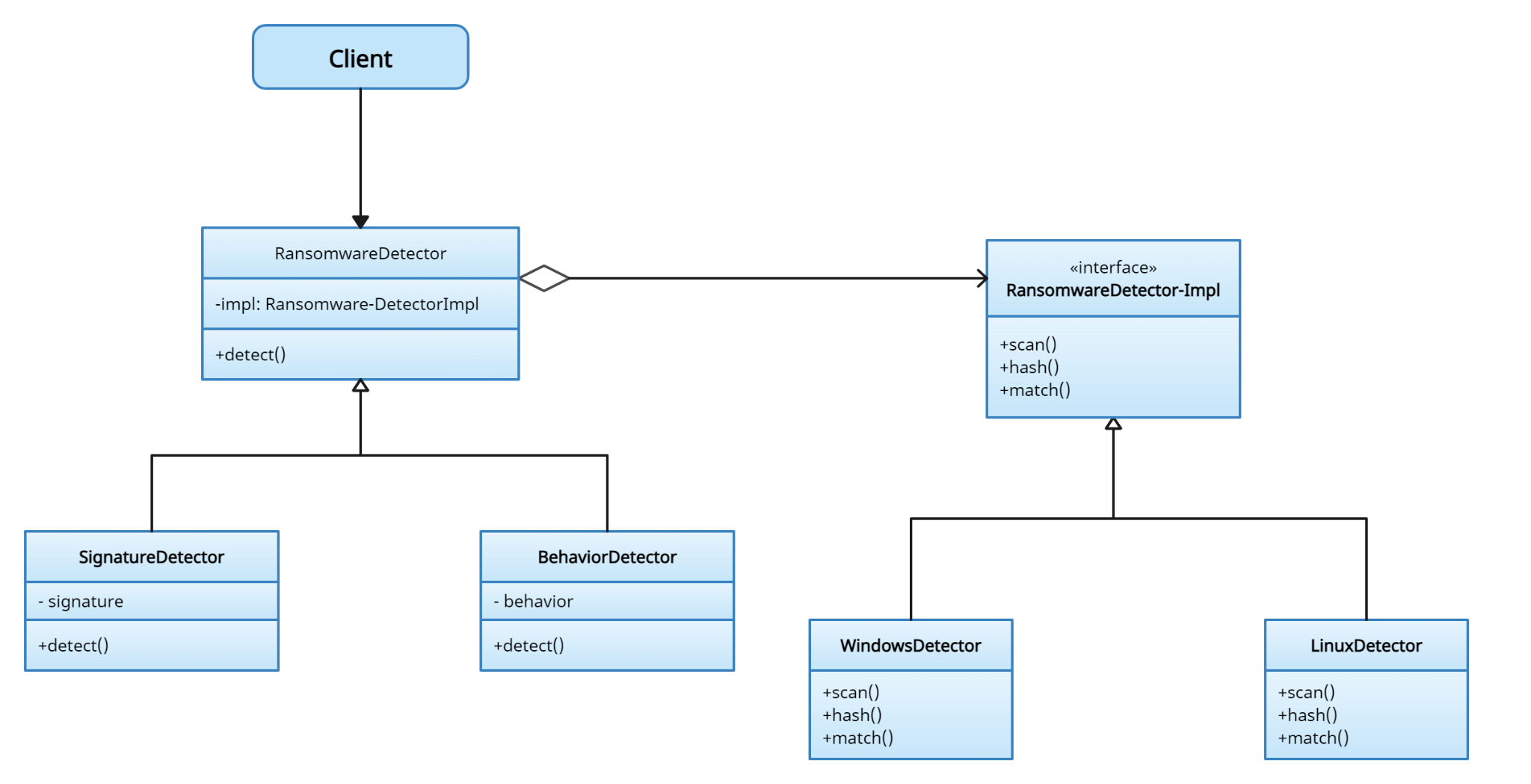
Why Bridge?  
The bridge design pattern is a type of structural design pattern which is used to split a large class into two separate inheritance hierarchies one for the implementations and one for the abstractions. It allows you to decouple an abstraction from its implementation so that the two can vary independently. It also helps you to avoid creating a large number of subclasses for each combination of abstraction and implementation.The bridge design pattern is useful when both the class and what it does vary often. The class itself can be thought of as the abstraction and what the class can do as the implementation. The bridge design pattern can also be thought of as two layers of abstraction

It can separate the abstraction (RansomwareDetector) from its implementation (RansomwareDetectorImpl) so that they can vary independently. For example, if we want to change the implementation of RansomwareDetectorImpl, we do not need to modify the RansomwareDetector interface or the Client class.

It can avoid creating a large number of subclasses for each combination of RansomwareDetector and its sub-detectors. For example, if we want to add a new sub-detector, such as a NetworkDetector, we do not need to create a new subclass of RansomwareDetector for each existing sub-detector combination.

It can increase the modularity, extensibility, and reusability of the software architecture. By separating the detection logic into different classes, we can easily modify or replace one class without affecting the others. We can also introduce new detection techniques or support new operating systems by adding new classes that implement the same interface. Furthermore, we can reuse the existing classes in other contexts or systems that require similar functionality.

* + - 1. Data Analyzer Class



[Figure](#figur_facade) 4. Decorator Design Pattern

#### Class Descriptions

#### Classes in the System

##### Class: Client

* Purpose: The Client class represents the user or system that uses the RansomwareDetector interface to analyze a program for ransomware threats.
* Constraints: None
* Persistent: *None*

###### Attribute Descriptions

Attribute: *None*

###### Method Descriptions

Method: requestAnalysis(program)

Return Type: Result

Parameters: program - a Program object that represents the program to be analyzed.

Purpose: Sends a request to the RansomwareDetector interface with the program to be analyzed and receives the final result.

Algorithm: Invoke the detectRansomware() method of the RansomwareDetector interface with the program as an argument and return its result.

Pre-conditions: The program is not null.

Post-conditions: The returned result is not null.

Attributes read/used: None

Methods called: detectRansomware() of RansomwareDetector interface

Explanation And Justification:

This Diagram shows a flowchart diagram of how a Ransomware Detector system works. It has different components that perform different functions, such as:

The Client is the user or device that uses the Ransomware Detector system to protect their files.The Ransomware Detector is the main component that detects and blocks ransomware attacks. It has an interface that communicates with the Client and an implementation that performs the detection and remediation tasks.The Signature Detector is a sub-component that scans files for known ransomware signatures, which are patterns of code or behavior that indicate a ransomware infection.The Behavior Detector is a sub-component that monitors files for suspicious or abnormal activities, such as encryption, deletion, or modification. It uses heuristic analysis and machine learning to detect unknown or new ransomware variants.The Windows Detector and Linux Detector are sub-components that specialize in detecting ransomware attacks on Windows and Linux operating systems, respectively.They use platform-specific techniques and features to identify and stop ransomware threats.

This diagram also shows the flow of data between the different components. For example:

The Client sends files to the Ransomware Detector for scanning and protection.

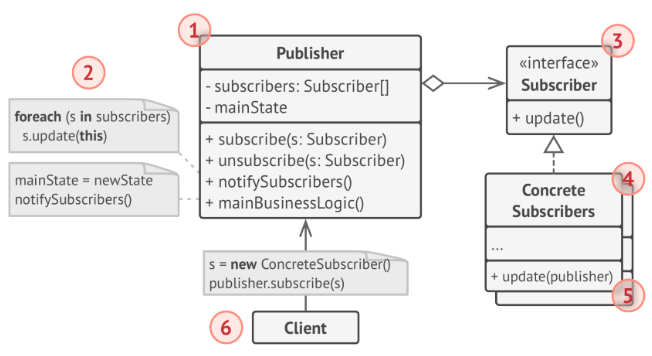
The Ransomware Detector sends files to the Signature Detector and Behavior Detector for analysis.The Behavior Detector sends files to the Windows Detector or Linux Detector depending on the operating system of the Client.The Signature Detector, Behavior Detector, Windows Detector, and Linux Detector send alerts to the Ransomware Detector if they detect any ransomware activity.The Ransomware Detector sends alerts and actions to the Client if it detects any ransomware attack. Actions may include blocking, quarantining, deleting, or restoring infected files.

## Behavioral Design Patterns

### **Observer**

#### 3.3.3.1 Design Explanation and Rationale

3.3.3.1.1 Structure



By implementing the Observer pattern, the application can establish a reliable notification mechanism where components can subscribe to receive updates and alerts about ransomware activities. This ensures that the application can efficiently distribute information, trigger necessary actions, and maintain a proactive defense against ransomware threats.

3.3.3.1.2 Communication and Control Flow

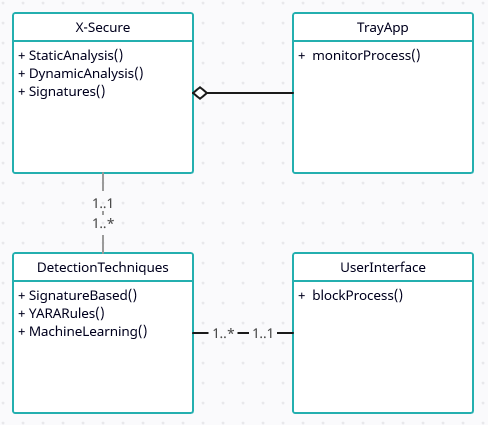
1. The observer components, such as the real-time scanning module and logging module, subscribe to receive notifications from the observable component. 2.
2. When a ransomware threat is detected, the observable component notifies the registered observers, enabling them to take appropriate actions like quarantining or blocking the malicious files and generating log entries.

3.3.3.1.3 Justification

3.3.3.1.3.1 Why Observer?

By incorporating the Observer pattern, the anti-ransomware application establishes a dependable system for exchanging notifications among its components. This enables different modules to subscribe and receive timely updates and alerts regarding ransomware activities. Consequently, the application efficiently disseminates crucial information, triggers appropriate actions, and upholds an active defense against ransomware threats.

#### 3.3.3.2 **Class Diagram**



#### Figure . Observer Design Pattern

#### 3.3.3.3 Class Descriptions

The Observer pattern in the X-Secure application involves two main classes which serve as the central component that maintains a list of subscribed Observers and manages the notification process. The class represents the entities that receive updates and alerts about ransomware activities. It defines an interface that allows it to notify them of any relevant changes. This decoupled design ensures flexibility and extensibility, allowing new Observers to be easily added and ensuring a seamless flow of information within the application.

#### 

3.3.3.4 Classes in the Observer Design

3.3.3.4.1 Class: TrayApp

* Purpose: The detection class in the Observer pattern of the anti-ransomware Windows app provides real-time monitoring and analysis of ransomware activities.
* Constraint: The detection class must adhere to the predefined rules and patterns to identify potential ransomware threats accurately.
* Persistence: The detection class maintains persistent state information to track ongoing ransomware activities and facilitate threat detection and response.

3.3.3.4.1.1 Attribute Descriptions

No attributes.

3.3.3.4.1.2 Method Descriptions

* Return Type: *void*
* Parameters: No Parameters
* Return value: None
* Pre-condition: The app is running and actively monitoring the system processes.
* Post-condition: Any suspicious or malicious processes detected are logged or handled accordingly.
* Attributes read/used: The monitoring process function reads and uses the list of running processes, their properties, and their associated behaviors.
* Methods called: The monitoring process function calls methods to analyze and evaluate the behavior of each running process.
* Processing logic: The monitoring process function applies specific logic and rules to identify any processes exhibiting ransomware-like behavior.

# 4. Dynamic Model

## 4.1. Scenarios

## 4.1.1 Scenario: Facade Scenarios

* Scenario Name: Simplify the structure of overall system
* Scenario Description:

Here we have applied facade design pattern to simplify the architecture of our system.

When any process starts, our software (client) wants to execute analysis on ransomware, It performs analysis and then it uses ML model to detect and save the results in database. These all three functions are separate and client have to execute them separately so to simplify the task, we bring all those functions together into one facade class, so client will only have to communicate to facade class and all the functions and classes will be implemented by our facade class, client (our software) will only have to interact with one class.

Client (our software) whenever wants to start analyzing first it will create objects of Analysis, ML Model and Database, then it will pass those objects to our FacadeRansomeDetector Class to handle all the tasks itself, client just need to call only one function and all the operations will be handled by Facade class.

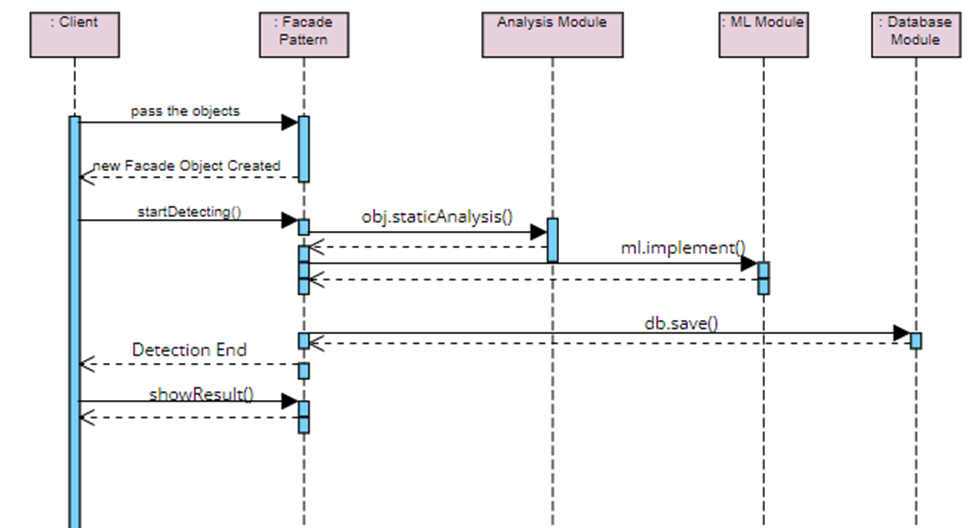
Facade has 2 functions, 1) startDetecting() 2)showResult(), client will call startDetecting() method it will handle all the other class objects and call the specific functions that are needed for detecting ransomwares.

Other function showResult() will show the result of ML model, analysis and database and whether the program was detected or not, again the relevant functions of the passed objects will be called here.

* Sequence Of Events:

1. The client (your software) wants to start the analysis process for ransomware detection.
2. The client creates objects of the Analysis, ML Model, and Database classes.
3. The client creates an object of the FacadeRansomDetector class, which will handle all the tasks.
4. The client passes the objects of Analysis, ML Model, and Database to the FacadeRansomDetector class.
5. The client calls the startDetecting() method of the FacadeRansomDetector class.
6. The FacadeRansomDetector class handles all the necessary operations internally.
7. The FacadeRansomDetector class invokes specific functions of the Analysis, ML Model, and Database objects to perform the ransomware detection process.
8. The analysis is performed by the Analysis object.
9. The ML Model is used to detect ransomware and save the results.
10. The results are stored in the Database.
11. The startDetecting() method completes the analysis process.
12. The client can call the showResult() method of the FacadeRansomDetector class to display the results.
13. The FacadeRansomDetector class retrieves the relevant information from the Analysis, ML Model, and Database objects.
14. The showResult() method displays the results, including whether the ransomware was detected or not.
15. The sequence of events related to the analysis, ML model, and database is simplified for the client, who only needs to interact with the FacadeRansomDetector class.

Sequence Diagram:



## 4.1.2 Scenario: Adapter Scenario

* Scenario Name: JSON to Excel data Conversion Scenario
* Scenario Description: ML model needs data in excelsheet or csv format but the analysis of ransomware from third party tool to extract features give result in json format so for this we use adapter pattern to make both interfaces compatible.

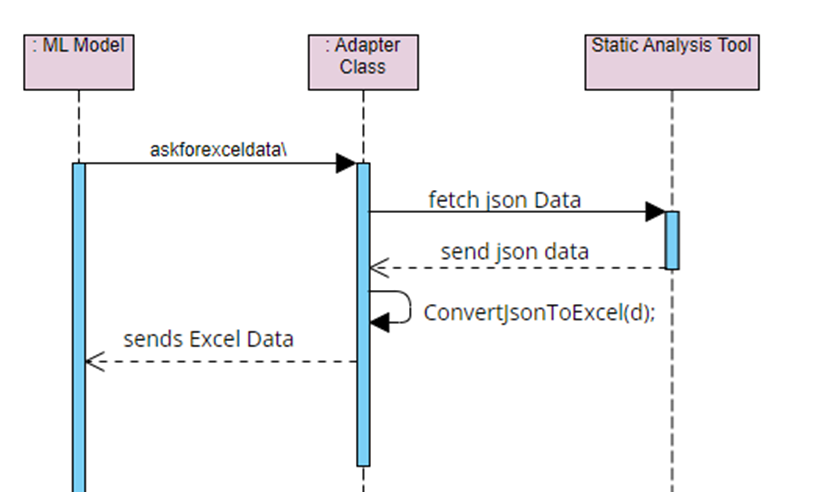
So the static analysis using third party tool will return data using giveAnalysisResultinJson() function, here we have created adapter class which will have composition relationship with analysis class, adapter class will take the result from analysis class and pass the data in convertJsonToExcel(d) function, this will convert data into excel or csv format which is the requirement of our ML Model Class.

ML Model interface will implement the adapter class whenever it needs the analysis data in excel format. ML model class will implement the interface whenever it need data in tabular form.

* Sequence of Events:

1. The static analysis using a third-party tool is performed.
2. The analysis tool generates the result in JSON format using the giveAnalysisResultinJson() function.
3. An adapter class is created to make the JSON data compatible with the ML model, which requires data in Excel or CSV format.
4. The adapter class has a composition relationship with the analysis class, meaning it contains an instance of the analysis class.
5. The adapter class receives the JSON result from the analysis class.
6. The adapter class calls the convertJsonToExcel(d) function, passing the JSON data as an argument.
7. The convertJsonToExcel(d) function converts the JSON data into Excel or CSV format, which is the required format for the ML model.
8. The adapter class returns the converted data to the ML model.
9. The ML model implements the interface provided by the adapter class whenever it needs the analysis data in Excel format.
10. The ML model requests the analysis data from the adapter class.
11. The adapter class retrieves the data from the analysis class and converts it into Excel or CSV format.
12. The adapter class returns the converted data to the ML model.
13. The ML model uses the converted data for further analysis or training.

* Sequence Diagram:

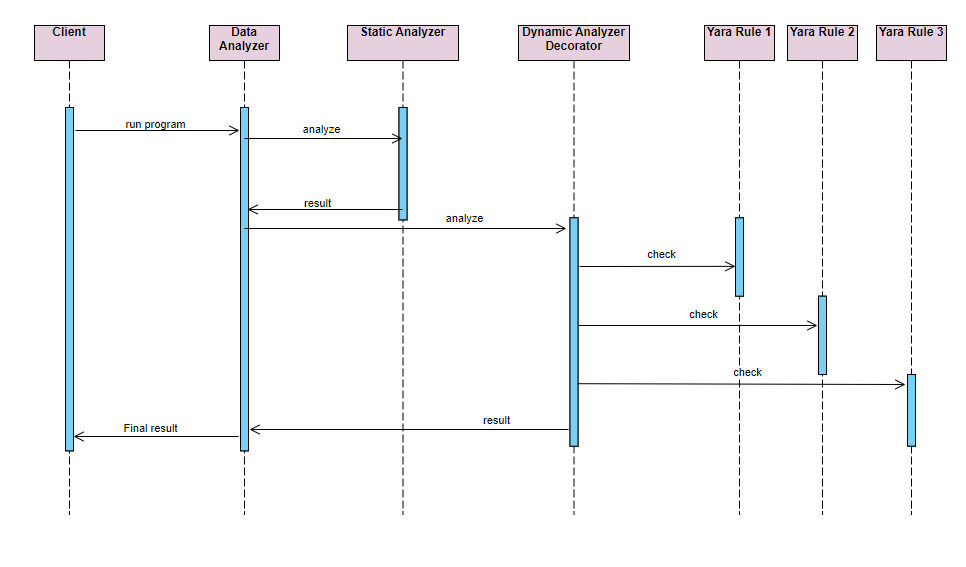


## 4.1.3 Decorator Scenarios

* *Scenario Name: Decorator Data Analysis scenario*
* *Scenario Description: This scenario shows how a client uses a data analysis system to run a program and get the final result. The data analysis system consists of a data analyzer, a static analyzer, a dynamic analyzer, and three Yara rules. The client sends a request to the data analyzer with the program to be run.*
* *Sequence of Events:*

1. *The data analyzer analyzes the program using the static analyzer and the dynamic analyzer.*
2. *The static analyzer checks the program for syntax errors and potential vulnerabilities.*
3. *The dynamic analyzer executes the program and monitors its behavior and performance.*
4. *The data analyzer applies the three Yara rules to the program to detect any malicious or suspicious patterns.*
5. *The data analyzer sends a response back to the client with the result of the analysis, which includes any errors, warnings, or alerts found by the system.*
6. *The client receives the final result and decides what to do next.*

* *Sequence Diagram:*

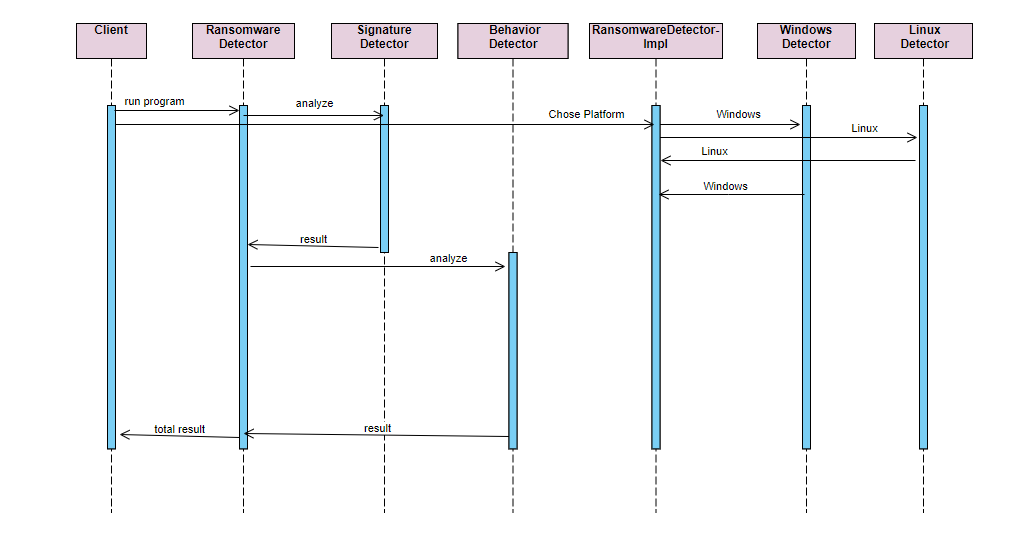
**

## 4.1.4 Bridge Scenarios

* *Scenario Name: Bridge Program Analysis System for Ransomware Prevention*
* *Scenario Description: This scenario shows how a client uses a ransomware detection system to analyze a program for potential ransomware threats. The ransomware detection system consists of several detectors that use different techniques to identify ransomware patterns.*
* *Sequence of Events:*

1. *The client runs the program, which is then sent to the ransomware detector for analysis.*
2. *The ransomware detector uses two sub-detectors: the signature detector and the behavior detector. The signature detector checks the program for known ransomware signatures, such as file extensions, encryption keys, or ransom notes. The behavior detector monitors the program’s actions, such as file access, network activity, or registry changes.*
3. *The ransomware detector combines the results of the sub-detectors and sends them back to the client.*
4. *The client also receives the results from two other detectors: the Windows detector and the Linux detector. These detectors are specific to the operating system of the client and check for any system-level changes or vulnerabilities caused by the program.*
5. *The client receives the final result, which is the total result of all the detectors. The final result indicates whether the program is safe, suspicious, or malicious. The client can then decide what to do next based on the final result.*

* *Sequence Diagram:*

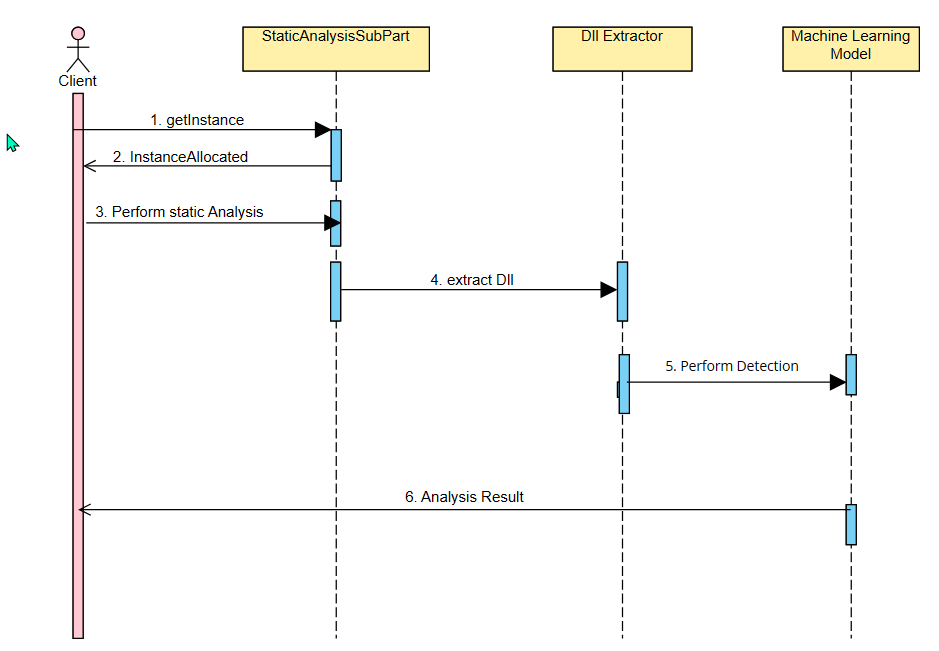
**

## 4.1.5. Singleton Scenario

* *Scenario Name: Performing Static Analysis using Singleton Pattern*
* *Scenario Description: This scenario illustrates the process of performing static analysis on an desktop application using the Static Analysis SubPart, which utilizes the Singleton pattern. The sequence of actions includes the initialization of the subsystem, the retrieval of the singleton instance, and the invocation of the static analysis method.*
* *Sequence of Events:*

1. *The client application initiates the process by calling the getInstance() method of the StaticAnalysisSubPart class, which is implemented using the Singleton pattern.*
2. *The StaticAnalysisSubPart class retrieves the singleton instance, ensuring that only one instance is created throughout the system.*
3. *The singleton instance of the StaticAnalysisSubPart then invokes the performStaticAnalysis() method, which triggers the static analysis process.*
4. *Within the static analysis process, the StaticAnalysisSubPart interacts with the dllModulesExtractor class to extract the required permissions from the application's code.*
5. *The dllModulesExtractor class, responsible for extracting dll modules, receives the request and invokes the dllExtractor() method.*
6. *The dllExtractor() method scans the application's code and extracts the necessary dlls based on predefined rules or patterns.*
7. *As part of the static analysis process, the StaticAnalysisSubPart also interacts with the MachineLearningModel class, which may perform additional analysis or processing on the extracted dlls.*
8. *Finally, the analysis results are then presented to the client.*

* *Sequence Diagram:*



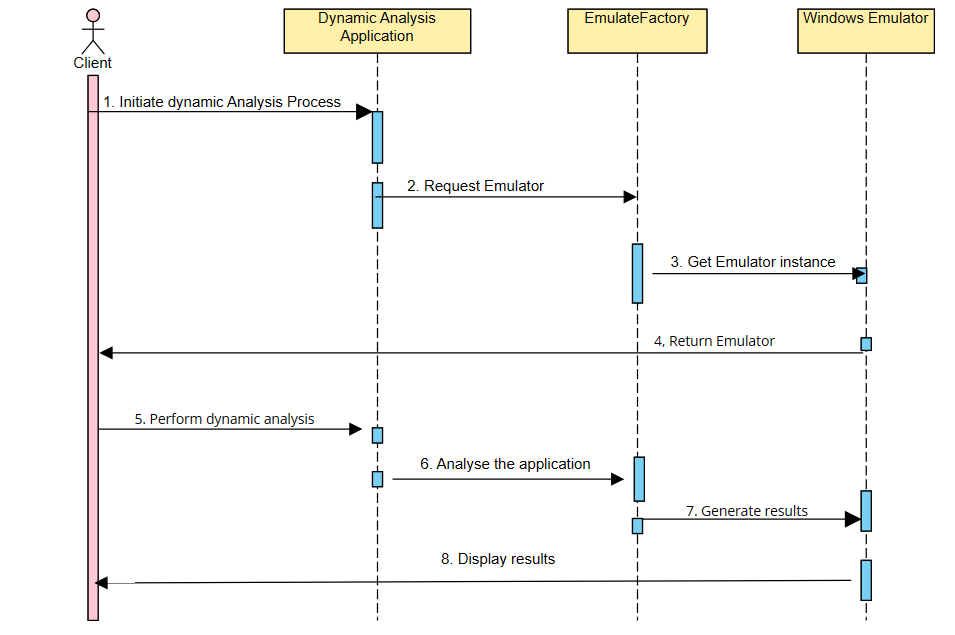
## 4.1.6. Factory Scenario

* *Scenario Name: Emulator Selection and Dynamic Analysis*
* *Scenario Description: The scenario involves the selection of a specific emulator type using the EmulatorFactory and performing dynamic analysis on an desktop application using the chosen emulator.*

* *Sequence of Events:*

1. *The client initiates the dynamic analysis process.*
2. *The Application requests the specified emulator from the EmulateFactory.*
3. *The EmulateFactory then gets the instance for the Dynamic Analysis*
4. *The specified instance returns to the client.*
5. *The client asks the application to perform the dynamic analysis.*
6. *The application then performs the dynamic analysis on the same instance of the emulator.*
7. *The analysis is performed on the emulator and the results are generated.*
8. *The results are then displayed to the client.*

*Sequence Diagram:*

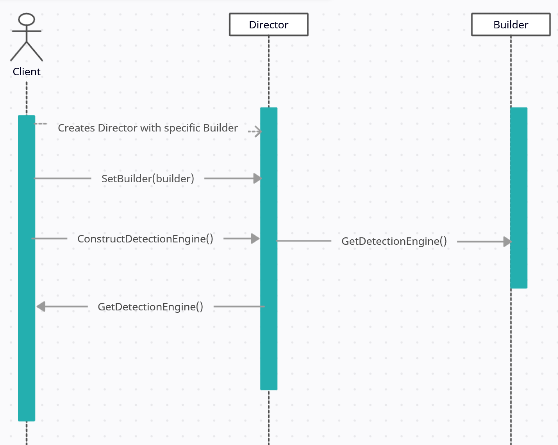


## 4.1.7. Builder Scenario

* *Scenario Name: Ransomware Detection Engine Construction*
* *Scenario Description: The scenario involves the X-Secure application as the builder pattern can be used to create a complex rule-based detection engine.*
* *Sequence of Events:*

1. *The client code initializes the Director with a specific builder implementation.*
2. *The Director invokes the builder methods to specify the steps for building the detection engine.*
3. *The builder constructs and configures the detection engine step by step.*
4. *Once the construction is complete, the client retrieves the fully built detection engine from the builder.*
5. *The client can then use the detection engine to scan and detect ransomware threats.*

*Sequence Diagram:*

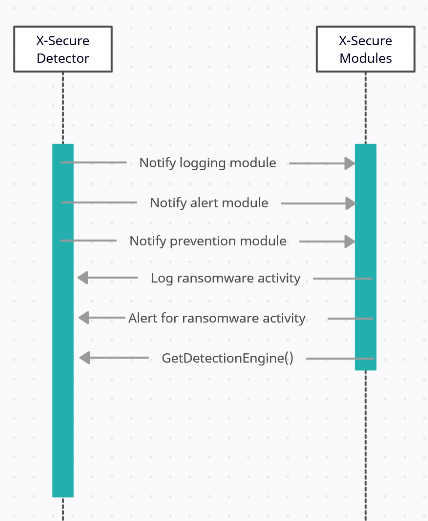


## 4.1.8. Observer Scenario

* *Scenario Name: Real-Time Ransomware Activity Monitoring*
* *Scenario Description: The scenario involves multiple components: a RansomwareDetector as the subject and various Observers (such as LoggingModule, AlertModule, and PreventionModule) that need to be notified when ransomware activity is detected.*
* *Sequence of Events:*

1. *The RansomwareDetector scans the system for suspicious behavior.*
2. *If ransomware activity is detected, the RansomwareDetector notifies all registered Observers.*
3. *The LoggingModule receives the notification and logs the details of the ransomware activity.*
4. *The AlertModule receives the notification and triggers appropriate alerts or notifications.*
5. *The PreventionModule receives the notification and takes necessary actions to mitigate the ransomware attack.*

*Sequence Diagram:*

**

# 5. Supplementary Documentation

Provide any other relevant documentation that may help in understanding the design.