# Java Implementation of CRS Algorithm

```
// CRS Algorithm Implementation (Java)

package org.webdefend.crs;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class CRSApp {

    private static final Logger log = LoggerFactory.getLogger(CRSApp.class);

    private P4SwitchController p4Switch; // Interface to P4 switch
    private WebServerFirewallController firewall; // Interface to web server firewall
    private CWEController cwe; // Interface to CWE module

    private Map<String, String> currentMitigationStrategies; // Target -> Strategy
    private ScheduledExecutorService executor;

    public CRSApp(P4SwitchController p4Switch, WebServerFirewallController firewall, CWEController cwe)
{
        this.p4Switch = p4Switch;
        this.firewall = firewall;
        this.cwe = cwe;
        this.currentMitigationStrategies = new HashMap<>();
        this.executor = Executors.newScheduledThreadPool(1);
    }

    public void start() {
        log.info("CRS Application started.");
        executor.scheduleAtFixedRate(this::monitorAttacks, 10, 10, TimeUnit.SECONDS); // Monitor every
10 seconds
    }

    public void stop() {
        log.info("CRS Application stopped.");
        executor.shutdown();
    }

    private void monitorAttacks() {
        AttackSignal signal = cwe.receiveSignal();
        if (signal != null) {
            processAttackSignal(signal);
        }

        // Monitor existing attacks and adjust strategies
        for (String target : currentMitigationStrategies.keySet()) {
            String strategy = currentMitigationStrategies.get(target);
            if (isMitigationIneffective(target, strategy)) {
                adjustMitigationStrategy(target, strategy);
            }
        }
    }
```

```
    private void processAttackSignal(AttackSignal signal) {
        String attackType = signal.getAttackType();
        String target = signal.getTarget();
        double confidence = signal.getConfidence();

        int severity = analyzeSeverity(attackType, target, confidence);
        String strategy = selectStrategy(attackType, severity);

        implementMitigation(target, strategy);
        currentMitigationStrategies.put(target, strategy);
    }

    private int analyzeSeverity(String attackType, String target, double confidence) {
        // Implement logic to analyze severity based on attack type, target, and confidence
        if (confidence > 0.8) {
            return 2; // High severity
        } else if (confidence > 0.5) {
            return 1; // Medium severity
        } else {
            return 0; // Low severity
        }
    }

    private String selectStrategy(String attackType, int severity) {
        // Implement logic to select mitigation strategy based on attack type and severity
        if (severity == 2) {
            return "Blocking";
        } else if (severity == 1) {
            return "Rate Limiting";
        } else {
            return "Redirection";
        }
    }

    private void implementMitigation(String target, String strategy) {
        log.info("Implementing mitigation strategy: {} for target: {}", strategy, target);
        if (strategy.equals("Rate Limiting")) {
            p4Switch.rateLimit(target);
        } else if (strategy.equals("Blocking")) {
            p4Switch.block(target);
            firewall.block(target);
        } else if (strategy.equals("Redirection")) {
            p4Switch.redirect(target, "honeypot"); // Replace "honeypot" with actual honeypot address
        }
    }

    private boolean isMitigationIneffective(String target, String strategy) {
        // Implement logic to monitor the effectiveness of the current mitigation strategy
        // This could involve querying P4 switch statistics, firewall logs, or network monitoring data
        return false; // Placeholder
    }

    private void adjustMitigationStrategy(String target, String currentStrategy) {
        // Implement logic to adjust the mitigation strategy based on the current strategy
        String newStrategy;
        if (currentStrategy.equals("Rate Limiting")) {
            newStrategy = "Blocking";
        } else if (currentStrategy.equals("Blocking")) {
            newStrategy = "Redirection";
        } else {
            newStrategy = "Rate Limiting";
        }
```

```
            log.info("Adjusting mitigation strategy for target: {} to: {}", target, newStrategy);
            implementMitigation(target, newStrategy);
            currentMitigationStrategies.put(target, newStrategy);
        }

        // Interfaces for interacting with other components
        public interface P4SwitchController {
            void rateLimit(String target);
            void block(String target);
            void redirect(String target, String honeypot);
        }

        public interface WebServerFirewallController {
            void block(String target);
        }

        public interface CWEController {
            AttackSignal receiveSignal();
        }

        public static class AttackSignal {
            private String attackType;
            private String target;
            private double confidence;

            public AttackSignal(String attackType, String target, double confidence) {
                this.attackType = attackType;
                this.target = target;
                this.confidence = confidence;
            }

            public String getAttackType() {
                return attackType;
            }

            public String getTarget() {
                return target;
            }

            public double getConfidence() {
                return confidence;
            }
        }
}
```

**Explanation of the Code:**

1. **Imports:** Imports logging and concurrency libraries.
2. **Class Definition:** CRSApp is the main class for the CRS module.
3. **Variables:**
    - p4Switch, firewall, cwe: Interfaces to interact with the P4 switch, web server firewall, and CWE module.
    - currentMitigationStrategies: A map to store the current mitigation strategies for each target.
    - executor: A scheduled executor for periodic monitoring.
4. **Constructor:** Initializes the interfaces and the currentMitigationStrategies map.
5. **start() Method:** Starts the CRS application and schedules the monitorAttacks() method for

periodic execution.

6. **stop() Method:** Stops the CRS application and shuts down the executor.
7. **monitorAttacks() Method:**
    - Receives attack signals from the CWE module.
    - Processes the attack signal using processAttackSignal().
    - Monitors existing attacks and adjusts strategies using isMitigationIneffective() and adjustMitigationStrategy().
8. **processAttackSignal() Method:**
    - Extracts attack type, target, and confidence from the attack signal.
    - Analyzes severity using analyzeSeverity().
    - Selects a mitigation strategy using selectStrategy().
    - Implements the mitigation strategy using implementMitigation().
    - Stores the current strategy in the currentMitigationStrategies map.
9. **analyzeSeverity() Method:** Analyzes the severity of the attack.
10. **selectStrategy() Method:** Selects the appropriate mitigation strategy.
11. **implementMitigation() Method:** Implements the selected mitigation strategy by interacting with the P4 switch and web server firewall.
12. **isMitigationIneffective() Method:** Monitors the effectiveness of the current mitigation strategy.
13. **adjustMitigationStrategy() Method:** Adjusts the mitigation strategy if it is ineffective.
14. **Interfaces:** Defines interfaces for interacting with the P4 switch, web server firewall, and CWE module.
15. **AttackSignal Class:** Represents an attack detection signal from the CWE module.

**Deployment Instructions:**

**Emulated Environment (Mininet-WiFi):**

1. **Set up the Environment:**
    - Ensure you have a Java development environment (JDK) and a build tool (Maven or Gradle) installed.
    - Set up Mininet-WiFi with a P4 software switch, a web server, and the SDN controller running the CWE module.
2. **Create CRS Application:**
    - Create a new Java project and add the necessary dependencies.
    - Create the CRSApp.java file and paste the code.
    - Implement the P4SwitchController, WebServerFirewallController, and CWEController interfaces to interact with your emulated environment. This may require using libraries or APIs specific to your chosen SDN controller and P4 software switch.
3. **Run the Application:**
    - Build the CRS application using Maven or Gradle.
    - Run the application, ensuring that the necessary interfaces are properly implemented and connected to the emulated components.
4. **Test the Application:**
    - Generate attack traffic in Mininet-WiFi.
    - Verify that the CRS application receives attack signals from the CWE module.
    - Verify that the CRS application implements the appropriate mitigation strategies by observing the behavior of the P4 switch and the web server.
    - Verify that the CRS application dynamically adjusts mitigation strategies when necessary.
5. **Monitor Logs:** Monitor the CRS application's logs for any errors or warnings.

**Real-World Environment:**

1. **Set up the Environment:**
    - Ensure you have a Java runtime environment (JRE) installed on a server.
    - Set up a real-world network with a P4-programmable switch, a web server, and the SDN controller running the CWE module.
2. **Create CRS Application:**
    - Create a new Java project and add the necessary dependencies.
    - Create the CRSApp.java file and paste the code.
    - Implement the P4SwitchController, WebServerFirewallController, and CWEController interfaces to interact with your real-world components. This may require using libraries or APIs provided by the P4 switch vendor, the web server firewall vendor, and the SDN controller.
3. **Deploy the Application:**
    - Build the CRS application using Maven or Gradle.
    - Deploy the application to the server. This may involve copying the JAR file and any necessary configuration files to the server.
4. **Run the Application:**
    - Run the application on the server, ensuring that the necessary interfaces are properly implemented and connected to the real-world components.
5. **Test the Application:**
    - Generate attack traffic on the real-world network.
    - Verify that the CRS application receives attack signals from the CWE module.
    - Verify that the CRS application implements the appropriate mitigation strategies by observing the behavior of the P4 switch and the web server firewall.
    - Verify that the CRS application dynamically adjusts mitigation strategies when necessary.
6. **Monitor Logs and Metrics:**
    - Monitor the CRS application's logs for any errors or warnings.
    - Monitor network metrics and firewall logs to verify the effectiveness of the mitigation strategies.
7. **Integration with other systems:**
    - Integrate the CRS with a SIEM (Security Information and Event Management) system to centralize alerts and events.
    - Implement mechanisms to handle and escalate alerts to network security personnel.
8. **Security and Performance:**
    - Secure the CRS application and its communication channels.
    - Tune the application's performance to handle high volumes of attack signals and mitigation actions.
9. **Maintenance:**
    - Regularly update the CRS application and its dependencies.
    - Monitor the application's performance and resource usage.
    - Regularly review and update mitigation strategies based on evolving threats