Java Implementation of CWE Algorithm (OpenDaylight Controller)

```
// CWE Algorithm Implementation (Java) - (OpenDaylight Controller)

package org.opendaylight.cwe;

import org.opendaylight.controller.md.sal.binding.api.DataBroker;
import org.opendaylight.controller.sal.binding.api.NotificationProviderService;
import org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.PacketProcessingService;
import org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.PacketReceived;
import org.opendaylight.yang.gen.v1.urn.opendaylight.packet.service.rev130709.PacketReceivedListener;
import org.opendaylight.yang.gen.v1.urn.ietf.params.xml.ns.yang.ietf.inet.types.rev130715.Ipv4Address;
import org.opendaylight.yang.gen.v1.urn.ietf.params.xml.ns.yang.ietf.inet.types.rev130715.PortNumber;
import org.opendaylight.yang.gen.v1.urn.ietf.params.xml.ns.yang.ietf.yang.types.rev130715.MacAddress;
import org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeConnectorId;
import org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.NodeId;
import org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.Nodes;
import org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.nodes.Node;
import org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.nodes.NodeKey;
import org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.nodes.NodeConnector;
import org.opendaylight.yang.gen.v1.urn.opendaylight.inventory.rev130819.nodes.NodeConnectorKey;
import org.opendaylight.yangtools.yang.binding.InstanceIdentifier;
import org.osgi.service.component.annotations.Activate;
import org.osgi.service.component.annotations.Component;
import org.osgi.service.component.annotations.Deactivate;
import org.osgi.service.component.annotations.Reference;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

import weka.classifiers.Classifier;
import weka.classifiers.trees.J48;
import weka.classifiers.trees.RandomForest;
import weka.classifiers.lazy.IBk;
import weka.classifiers.functions.SMO;
import weka.classifiers.meta.AdaBoostM1;
import weka.core.Attribute;
import weka.core.DenseInstance;
import weka.core.Instances;

@Component(service = {CWEModule.class, PacketReceivedListener.class}, immediate = true)
public class CWEModule implements PacketReceivedListener {

    private static final Logger LOG = LoggerFactory.getLogger(CWEModule.class);

    @Reference
    private PacketProcessingService packetProcessingService;

    @Reference
    private DataBroker dataBroker;

    @Reference
    private NotificationProviderService notificationProviderService;
```

```
    private Map<String, Classifier> classifiers;
    private Map<String, Double> weights;
    private double threshold = 0.7;
    private long updateInterval = 60;
    private Instances trainingData;
    private ScheduledExecutorService executor;

    @Activate
    public void activate() {
        LOG.info("CWEModule Activated");
        notificationProviderService.registerListener(this);

        classifiers = new HashMap<>();
        classifiers.put("KNN", new IBk());
        classifiers.put("DT", new J48());
        classifiers.put("RF", new RandomForest());
        classifiers.put("SVM", new SMO());
        classifiers.put("XGBoost", new AdaBoostM1());

        weights = new HashMap<>();
        for (String classifierName : classifiers.keySet()) {
            weights.put(classifierName, 1.0);
        }

        ArrayList<Attribute> attributes = new ArrayList<>();
        attributes.add(new Attribute("srcIP"));
        attributes.add(new Attribute("dstIP"));
        attributes.add(new Attribute("srcPort"));
        attributes.add(new Attribute("dstPort"));
        attributes.add(new Attribute("method"));
        attributes.add(new Attribute("payloadSize"));
        attributes.add(new Attribute("errorCode"));
        attributes.add(new Attribute("anomalyFlag"));
        attributes.add(new Attribute("attack", new ArrayList<String>() {{
            add("normal");
            add("attack");
        }}));
        trainingData = new Instances("MetaData", attributes, 0);
        trainingData.setClassIndex(trainingData.numAttributes() - 1);

        executor = Executors.newScheduledThreadPool(1);
        executor.scheduleAtFixedRate(this::updateWeights, updateInterval, updateInterval,
TimeUnit.SECONDS);
    }

    @Deactivate
    public void deactivate() {
        LOG.info("CWEModule Deactivated");
        notificationProviderService.unregisterListener(this);
        executor.shutdown();
    }

    @Override
    public void onPacketReceived(PacketReceived packet) {
        ByteBuffer payload = packet.getPayload();
        if (payload == null) {
            return;
        }

        byte[] payloadBytes = payload.array();
        if (payloadBytes.length < 14) {
            return;
```

```
        }

        if (payloadBytes[12] == 0x08 && payloadBytes[13] == 0x00) {
            // IPv4 packet
            try {
                processPacket(payloadBytes, packet.getIngress().getValue().getNodeId());
            } catch (Exception e) {
                LOG.error("Error processing packet", e);
            }
        }
    }

    private void processPacket(byte[] payloadBytes, NodeId nodeId) throws Exception {
        // Parse Ethernet, IP, TCP, and Data
        int ipHeaderLength = (payloadBytes[14] & 0x0F) * 4;
        int tcpHeaderLength = ((payloadBytes[14 + ipHeaderLength + 12] >> 4) & 0x0F) * 4;

        if (payloadBytes[14 + 9] != 6) {
            return; // Not TCP
        }

        int srcPort = ((payloadBytes[14 + ipHeaderLength] & 0xFF) << 8) | (payloadBytes[14 +
ipHeaderLength + 1] & 0xFF);
        int dstPort = ((payloadBytes[14 + ipHeaderLength + 2] & 0xFF) << 8) | (payloadBytes[14 +
ipHeaderLength + 3] & 0xFF);

        if (dstPort != 80 && dstPort != 443) {
            return; // Not HTTP/HTTPS
        }

        int dataOffset = 14 + ipHeaderLength + tcpHeaderLength;
        if (payloadBytes.length <= dataOffset) {
            return; // No data
        }

        byte[] dataBytes = new byte[payloadBytes.length - dataOffset];
        System.arraycopy(payloadBytes, dataOffset, dataBytes, 0, dataBytes.length);

        processMetadata(dataBytes);
    }

    private void processMetadata(byte[] metadataBytes) {
        String metadataStr = new String(metadataBytes);
        String[] metadataValues = metadataStr.split(",");

        if (metadataValues.length < 8) return;

        double[] metadata = new double[8];
        for (int i = 0; i < 8; i++) {
            try {
                metadata[i] = Double.parseDouble(metadataValues[i]);
            } catch (NumberFormatException e) {
                LOG.error("Error parsing metadata value: {}", metadataValues[i]);
                return;
            }
}

        DenseInstance instance = new DenseInstance(9);
        for (int i = 0; i < 8; i++) {
            instance.setValue(i, metadata[i]);
        }
        instance.setDataset(trainingData);
```

```
        Map<String, Double> predictions = new HashMap<>();
        Map<String, Double> confidenceScores = new HashMap<>();
        for (String classifierName : classifiers.keySet()) {
            try {
                Classifier classifier = classifiers.get(classifierName);
                double prediction = classifier.distributionForInstance(instance)[1];
                double confidence = classifier.classifyInstance(instance);
                predictions.put(classifierName, prediction);
                confidenceScores.put(classifierName, confidence);
            } catch (Exception e) {
                LOG.error("Error with classifier {}: {}", classifierName, e.getMessage());
            }
        }

        double weightedScore = calculateWeightedScore(predictions, confidenceScores);

        if (weightedScore > threshold) {
            sendAlert("Coordinated attack detected", weightedScore);
        }

        instance.setClassValue(weightedScore > threshold ? "attack" : "normal");
        trainingData.add(instance);
        try {
            for (Classifier classifier : classifiers.values()) {
                classifier.buildClassifier(trainingData);
            }
        } catch (Exception e) {
            LOG.error("Error rebuilding classifiers: {}", e.getMessage());
        }
    }

    private double calculateWeightedScore(Map<String, Double> predictions, Map<String, Double>
confidenceScores) {
        double weightedScore = 0.0;
        double totalWeight = 0.0;
        for (String classifierName : classifiers.keySet()) {
            weightedScore += predictions.get(classifierName) * weights.get(classifierName) *
confidenceScores.get(classifierName);
            totalWeight += weights.get(classifierName);
        }
        return totalWeight > 0 ? weightedScore / totalWeight : 0.0;
    }

    private void updateWeights() {
        for (String classifierName : classifiers.keySet()) {
            double accuracy = calculateClassifierAccuracy(classifierName);
            weights.put(classifierName, weights.get(classifierName) * (1 + accuracy));
        }
    }

    private double calculateClassifierAccuracy(String classifierName) {
        int correct = 0;
        int total = trainingData.numInstances();
        if (total == 0) return 0;

        try {
            Classifier classifier = classifiers.get(classifierName);
            for (int i = Math.max(0, total - 10); i < total; i++) {
                double prediction = classifier.classifyInstance(trainingData.instance(i));
                if (prediction == trainingData.instance(i).classValue()) {
                    correct++;
```

```
                }
            }
        } catch (Exception e) {
            LOG.error("Error calculating accuracy for {}: {}", classifierName, e.getMessage());
        }

        return (double) correct / Math.min(10, total);
    }

    private void sendAlert(String message, double score) {
        LOG.info("Alert: {} (Score: {})", message, score);
        // Implement logic to send alert to CRS module (e.g., via REST API)
    }
}
```

Explanation:

1. **Imports:**
   - Imports OpenDaylight, packet, and Weka libraries.
2. **Class Definition:**
   - CWEModule implements PacketReceivedListener.
3. **Variables:**
   - packetProcessingService, dataBroker, notificationProviderService: OpenDaylight services.
   - classifiers, weights, threshold, updateInterval, trainingData, executor: CWE algorithm parameters.
4. **activate() Method:**
   - Registers CWEModule as a PacketReceivedListener.
   - Initializes classifiers, weights, and training data.
   - Schedules updateWeights() to run periodically.
5. **deactivate() Method:**
   - Unregisters the listener and shuts down the executor.
6. **onPacketReceived() Method:**
   - Handles PacketReceived notifications.
   - Parses Ethernet and IPv4 headers.
   - Calls processPacket() for IPv4 packets.
7. **processPacket() Method:**
   - Parses TCP and data payloads.
   - Calls processMetadata() for HTTP/HTTPS traffic.
8. **processMetadata() Method:**
   - Extracts metadata from the packet data.
   - Creates a Weka DenseInstance.
   - Performs ensemble classification.
   - Calculates the weighted score.
   - Makes a decision and sends an alert.
   - Updates training data and rebuilds classifiers.
9. **calculateWeightedScore(), updateWeights(), calculateClassifierAccuracy(), sendAlert() Methods:**
   - Same as in the Floodlight example.

Deployment Instructions:

Emulated Environment (Mininet-WiFi):

1. **Install OpenDaylight:**
   - Download and install OpenDaylight.
2. **Add Weka Dependency:**
   - Add the Weka library to your OpenDaylight project's pom.xml.
3. **Write CWE Module:**
   - Create a new Java class CWEModule.java in your OpenDaylight project.
   - Copy and paste the code.
4. **Build and Install Bundle:**
   - Build the OpenDaylight bundle and install it using Karaf.
5. **Configure P4 Switch:**
   - Configure your P4 switch to send metadata packets to the OpenDaylight controller.
6. **Generate Traffic:**
   - Generate HTTP/HTTPS traffic, including attack traffic.
7. **Monitor OpenDaylight Logs:**
   - Monitor the OpenDaylight logs for alerts.
8. **Integrate with CRS Module:**
   - Implement sendAlert() to send alerts to the CRS module.

Real-World Environment:

1. **OpenDaylight Installation:**
   - Install OpenDaylight on a server.
2. **P4-Capable Hardware:**
   - Use a P4-programmable switch.
3. **Bundle Deployment:**
   - Deploy the CWE module bundle to OpenDaylight.
4. **Network Configuration:**
   - Configure the network to forward traffic to the P4 switch.
5. **Testing:**
   - Generate real-world attack traffic and monitor OpenDaylight for alerts.
   - Use network monitoring tools to verify the traffic flow.
6. **CRS Integration:**
   - Integrate OpenDaylight with the CRS module using REST APIs or other communication methods.
7. **Training:**
   - Train the classifiers using real network data and load the models into the application