Java Implementation of CWE Algorithm (ONOS Controller)

```java
// CWE Algorithm Implementation (Java) - (ONOS Controller)

package org.onosproject.cwe;

import org.onlab.packet.Ethernet;
import org.onlab.packet.IPv4;
import org.onlab.packet.TCP;
import org.onosproject.core.ApplicationId;
import org.onosproject.core.CoreService;
import org.onosproject.net.packet.InboundPacket;
import org.onosproject.net.packet.PacketContext;
import org.onosproject.net.packet.PacketProcessor;
import org.onosproject.net.packet.PacketService;
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.osgi.service.component.annotations.ReferenceCardinality;
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 = CWEApp.class, immediate = true)
public class CWEApp implements PacketProcessor {

    private final Logger log = LoggerFactory.getLogger(getClass());

    @Reference(cardinality = ReferenceCardinality.MANDATORY)
    protected CoreService coreService;

    @Reference(cardinality = ReferenceCardinality.MANDATORY)
    protected PacketService packetService;

    private ApplicationId appId;

    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
    protected void activate() {
        appId = coreService.registerApplication("org.onosproject.cwe");
        packetService.addProcessor(this, PacketProcessor.director(2));

        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);

        log.info("Started");
    }

    @Deactivate
    protected void deactivate() {
        packetService.removeProcessor(this);
        executor.shutdown();
        log.info("Stopped");
    }

    @Override
    public void process(PacketContext context) {
        if (context.isHandled()) {
            return;
        }

        InboundPacket pkt = context.inPacket();
        Ethernet eth = pkt.parsed();

        if (eth == null || eth.getEtherType() != Ethernet.TYPE_IPV4) {
            return;
        }

        IPv4 ipv4 = (IPv4) eth.getPayload();
        if (ipv4.getProtocol() != IPv4.PROTOCOL_TCP) {
            return;
        }

        TCP tcp = (TCP) ipv4.getPayload();
        if (tcp.getDestinationPort() != 80 && tcp.getDestinationPort() != 443) {
            return;
        }

        ByteBuffer payload = pkt.unparsed();
        if (payload == null) {
            return;
        }

        byte[] packetData = payload.array();
        try {
            processPacket(packetData);
        } catch (Exception e) {
            log.error("Error processing packet", e);
        }
    }

    private void processPacket(byte[] packetData) throws Exception {
        int ipHeaderLength = (packetData[14] & 0x0F) * 4;
        int tcpHeaderLength = ((packetData[14 + ipHeaderLength + 12] >> 4) & 0x0F) * 4;

        int dataOffset = 14 + ipHeaderLength + tcpHeaderLength;
        if (packetData.length <= dataOffset) {
            return;
        }

        byte[] dataBytes = new byte[packetData.length - dataOffset];
        System.arraycopy(packetData, 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)
    }
}
```

<u>Explanation:</u>

1. **Imports:**
   - Imports ONOS, packet, and Weka libraries.
2. **Class Definition:**
   - CWEApp implements PacketProcessor.
3. **Variables:**
   - coreService, packetService: ONOS core services.
   - appId: Application ID.
   - classifiers, weights, threshold, updateInterval, trainingData, executor: CWE algorithm parameters.
4. **activate() Method:**
   - Registers the application and adds the packet processor.
   - Initializes classifiers, weights, and training data.
   - Schedules updateWeights() to run periodically.
5. **deactivate() Method:**
   - Removes the packet processor and shuts down the executor.
6. **process() Method:**
   - Handles incoming packets.
   - Parses Ethernet, IPv4, and TCP headers.
   - Calls processPacket() for HTTP/HTTPS traffic.
7. **processPacket() Method:**
   - Parses data payload.
   - Calls processMetadata().
8. **processMetadata() Method:**
   - Extracts metadata.
   - 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 previous examples.

Deployment Instructions:

Emulated Environment (Mininet-WiFi):

1. **Install ONOS:**
   - Install ONOS on a Linux machine.
2. **Add Weka Dependency:**
   - Add the Weka dependency to your ONOS application's pom.xml.
3. **Create CWE Application:**
   - Create a new ONOS application project.
   - Create CWEApp.java and paste the code.
4. **Build and Install Application:**
   - Build the ONOS application: mvn clean install
   - Install the application in ONOS: onos app install target/<app-name>.oar
   - Activate the application: onos app activate org.onosproject.cwe
5. **Create Mininet-WiFi Topology:**
   - Create a Mininet-WiFi topology with a P4 software switch.
6. **Configure P4 Switch:**
   - Configure the P4 switch to send metadata packets to the ONOS controller.
7. **Generate Traffic:**
   - Use tools like hping3 or curl to generate HTTP/HTTPS traffic.
8. **Monitor ONOS Logs:**
   - Monitor the ONOS logs for alerts.

Real-World Environment:

1. **ONOS Installation:**
   - Install ONOS on a cluster of servers for redundancy and scalability.
2. **P4-Capable Hardware:**
   - Use a P4-programmable switch.
3. **Network Configuration:**
   - Configure the network to forward traffic to the P4 switch.
4. **Deploy ONOS Application:**
   - Deploy the ONOS CWE application.
5. **Training:**
   - Train the Weka classifiers with real network data.
6. **CRS Integration:**
   - Implement sendAlert() to send alerts to the CRS module.
7. **Monitoring:**
   - Monitor ONOS logs and the CRS module for alerts.
   - Use network monitoring tools.
8. **Performance Tuning and Security:**
   - Adjust parameters, optimize classifiers, and secure the system.
9. **Maintenance:**
   - Regularly update models and monitor the system.