Java Implementation of CWE Algorithm (Beacon Controller)

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

package org.openflowsdn.controller.cwe;

import org.openflow.protocol.OFMessage;
import org.openflow.protocol.OFPacketIn;
import org.openflow.protocol.OFType;
import org.openflow.util.HexString;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import net.beaconcontroller.core.IBeaconProvider;
import net.beaconcontroller.core.IOFMessageListener;
import net.beaconcontroller.core.IOFSwitch;
import net.beaconcontroller.core.module.IBeaconModule;
import net.beaconcontroller.core.module.IBeaconModuleContext;
import net.beaconcontroller.core.module.IBeaconService;

import java.util.ArrayList;
import java.util.Collection;
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;

public class CWEModule implements IBeaconModule, IOFMessageListener {

    protected IBeaconProvider beaconProvider;
    protected static Logger log;
    protected ScheduledExecutorService executor;

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

    @Override
    public void init(IBeaconModuleContext context) {
        beaconProvider = context.getService(IBeaconProvider.class);
        log = LoggerFactory.getLogger(CWEModule.class);
        executor = Executors.newScheduledThreadPool(1);

        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.scheduleAtFixedRate(this::updateWeights, updateInterval, updateInterval,
TimeUnit.SECONDS);
    }

    @Override
    public void startUp(IBeaconModuleContext context) {
        beaconProvider.addOFMessageListener(OFType.PACKET_IN, this);
    }

    @Override
    public Collection<Class<? extends IBeaconService>> getModuleServices() {
        return null;
    }

    @Override
    public Map<Class<? extends IBeaconService>, IBeaconService> getServiceImpls() {
        return null;
    }

    @Override
    public Collection<Class<? extends IBeaconService>> getModuleDependencies() {
        Collection<Class<? extends IBeaconService>> l = new ArrayList<Class<? extends
IBeaconService>>();
        l.add(IBeaconProvider.class);
        return l;
    }

    @Override
    public String getName() {
        return CWEModule.class.getSimpleName();
    }

    @Override
    public boolean isCallbackOrderingPrereq(OFType type, String name) {
        return false;
    }

    @Override
    public boolean isCallbackOrderingPostreq(OFType type, String name) {
        return false;
    }

    @Override
    public Command receive(IOFSwitch sw, OFMessage msg) {
        if (msg.getType() == OFType.PACKET_IN) {
            OFPacketIn pi = (OFPacketIn) msg;
            byte[] packetData = pi.getPacketData();
            if (packetData != null && packetData.length > 14 && packetData[12] == 0x08 && packetData[13]
== 0x00) {
                try {
                    processPacket(packetData);
                } catch (Exception e) {
                    log.error("Error processing packet", e);
                }
            }
        }
        return Command.CONTINUE;
    }

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

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

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

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

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

        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)
    }
}
```

Explanation:

1. **Imports:**
    - Imports Beacon, OpenFlow, and Weka libraries.
2. **Class Definition:**
    - CWEModule implements IBeaconModule and IOFMessageListener.
3. **Variables:**
    - beaconProvider, log, executor: Beacon core services and logging.
    - classifiers, weights, threshold, updateInterval, trainingData: CWE algorithm parameters.
4. **init() Method:**
    - Initializes Beacon services and logging.
    - Creates and initializes classifiers and weights.
    - Creates Weka Instances.
    - Schedules updateWeights() to run periodically.
5. **startUp() Method:**
    - Registers CWEModule as an OFMessageListener for PACKET_IN messages.
6. **receive() Method:**
    - Handles PACKET_IN messages.
    - Parses packet data.
    - 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 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 and OpenDaylight examples.

<u>**Deployment Instructions:**</u>

<u>**Emulated Environment (Mininet-WiFi):**</u>

1. **Install Prerequisites:**
    - Java Development Kit (JDK)
    - Apache Ant (Beacon uses Ant for build management)
    - Git
    - Mininet-WiFi
    - Weka
2. **Clone and Build Beacon:**
    - Clone the Beacon repository: git clone https://github.com/OpenFlowSDN/Beacon.git
    - Navigate to the Beacon directory: cd Beacon
    - Checkout a specific version of Beacon: git checkout <version>
    - Build Beacon: ant compile
3. **Add Weka Dependency:**
    - Place the Weka JAR in the lib directory within the Beacon root folder.
    - Update the build.xml file to include the Weka jar in the classpath.
4. **Create CWE Module:**
    - Create a new directory org/openflowsdn/controller/cwe within src.
    - Create CWEModule.java in the new directory and paste the code.
5. **Configure Beacon:**
    - Update src/net/beaconcontroller/core/module/BeaconModuleLoader.java to load the CWEModule.
6. **Run Beacon:**
    - Start Beacon: ant run
7. **Create Mininet-WiFi Topology:**
    - Create a Mininet-WiFi topology with a P4 software switch.
8. **Configure P4 Switch:**
    - Configure the P4 switch to send metadata packets to the Beacon controller.
9. **Generate Traffic:**
    - Use tools like hping3 or curl to generate HTTP/HTTPS traffic.
10. **Monitor Beacon:**
    - Monitor the Beacon console logs for alerts.

<u>**Real-World Environment:**</u>

1. **Beacon Installation:**
    - Install Beacon on a server.
2. **P4-Capable Hardware:**
    - Use a P4-programmable switch.
3. **Network Configuration:**
    - Configure the network to forward traffic to the P4 switch.
4. **Deploy CWE Module:**
    - Deploy the compiled Beacon with the CWE module.
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 Beacon 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.