Java Implementation of CWE Algorithm (Floodlight Controller)

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

package net.floodlightcontroller.cwe;

import net.floodlightcontroller.core.FloodlightContext;
import net.floodlightcontroller.core.IFloodlightProviderService;
import net.floodlightcontroller.core.IOFMessageListener;
import net.floodlightcontroller.core.IOFSwitch;
import net.floodlightcontroller.core.module.FloodlightModuleContext;
import net.floodlightcontroller.core.module.FloodlightModuleException;
import net.floodlightcontroller.core.module.IFloodlightModule;
import net.floodlightcontroller.core.module.IFloodlightService;
import net.floodlightcontroller.packet.Data;
import net.floodlightcontroller.packet.Ethernet;
import net.floodlightcontroller.packet.IPv4;
import net.floodlightcontroller.packet.TCP;
import net.floodlightcontroller.restserver.IRestApiService;
import org.projectfloodlight.openflow.protocol.OFMessage;
import org.projectfloodlight.openflow.protocol.OFType;
import org.projectfloodlight.openflow.types.IPv4Address;
import org.projectfloodlight.openflow.types.IpProtocol;
import org.projectfloodlight.openflow.types.OFPort;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

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 IFloodlightModule, IOFMessageListener {

    protected IFloodlightProviderService floodlightProvider;
    protected IRestApiService restApi;
    protected Map<IOFSwitch, Long> lastSeen;
    protected static Logger log;
    protected ScheduledExecutorService executor;

    // CWE Algorithm Variables
    private Map<String, Classifier> classifiers;
    private Map<String, Double> weights;
    private double threshold = 0.7; // Example threshold
    private long updateInterval = 60; // Example update interval in seconds

    // Weka Instances Definition
    private Instances trainingData;
```

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

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

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

    @Override
    public void init(FloodlightModuleContext context) throws FloodlightModuleException {
        floodlightProvider = context.getServiceImpl(IFloodlightProviderService.class);
        restApi = context.getServiceImpl(IRestApiService.class);
        lastSeen = new HashMap<IOFSwitch, Long>();
        log = LoggerFactory.getLogger(CWEModule.class);
        executor = Executors.newScheduledThreadPool(1);

        // Initialize Classifiers
        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()); // Simplified, replace with actual XGBoost
integration

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

        // Initialize Weka Instances
        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);

        // Schedule weight updates
        executor.scheduleAtFixedRate(this::updateWeights, updateInterval, updateInterval,
```

```
TimeUnit.SECONDS);
    }

    @Override
    public void startUp(FloodlightModuleContext context) throws FloodlightModuleException {
        floodlightProvider.addOFMessageListener(OFType.PACKET_IN, this);
    }

    @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 net.floodlightcontroller.core.IListener.Command receive(IOFSwitch sw, OFMessage msg,
FloodlightContext cntx) {
        if (msg.getType() == OFType.PACKET_IN) {
            Ethernet eth = IFloodlightProviderService.bcStore.get(cntx,
IFloodlightProviderService.CONTEXT_PI_PAYLOAD);
            if (eth.getEtherType() == org.projectfloodlight.openflow.types.EthType.IPv4) {
                IPv4 ipv4 = (IPv4) eth.getPayload();
                if (ipv4.getProtocol() == IpProtocol.TCP) {
                    TCP tcp = (TCP) ipv4.getPayload();
                    if (tcp.getDestinationPort().getPort() == 80 || tcp.getDestinationPort().getPort()
== 443) {
                        Data data = (Data) tcp.getPayload();
                        if (data != null) {
                            processMetadata(ipv4, tcp, data);
                        }
                    }
                }
            }
        }
        return net.floodlightcontroller.core.IListener.Command.CONTINUE;
    }

    private void processMetadata(IPv4 ipv4, TCP tcp, Data data) {
        // Extract metadata from packet data
        String metadataStr = new String(data.getData());
        String[] metadataValues = metadataStr.split(","); // Assuming comma-separated values

        if (metadataValues.length < 8) return; // Ensure all fields are present

        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;
            }
        }
```

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

        // Ensemble Classification
        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]; // Probability of
attack
                double confidence = classifier.classifyInstance(instance); // Class value
                predictions.put(classifierName, prediction);
                confidenceScores.put(classifierName, confidence);
            } catch (Exception e) {
                log.error("Error with classifier {}: {}", classifierName, e.getMessage());
            }
        }

        // Calculate Weighted Score
        double weightedScore = calculateWeightedScore(predictions, confidenceScores);

        // Make Final Decision
       if (weightedScore > threshold) {
sendAlert("Coordinated attack detected", weightedScore);
        }

        // Update Training Data (Simplified, for demonstration)
        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() {
        // Simplified weight update logic (replace with your actual logic)
        for (String classifierName : classifiers.keySet()) {
            // Example: Increase weight if classifier predicted correctly, decrease otherwise
            double accuracy = calculateClassifierAccuracy(classifierName);
```

```
            weights.put(classifierName, weights.get(classifierName) * (1 + accuracy)); // Example:
 Adjust weight based on 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++) { // Check last 10 instances
                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 necessary Floodlight, OpenFlow, packet, and Weka libraries.
2. **Class Definition:**
   - CWEModule implements IFloodlightModule and IOFMessageListener.
3. **Variables:**
   - floodlightProvider, restApi, lastSeen, log, executor: Floodlight core services and logging.
   - classifiers, weights, threshold, updateInterval: CWE algorithm parameters.
   - trainingData: Weka Instances for storing metadata and training classifiers.
4. **init() Method:**
   - Initializes Floodlight services and logging.
   - Creates and initializes the classifiers (KNN, DT, RF, SVM, XGBoost).
   - Sets initial weights for classifiers.
   - Creates Weka Instances and sets the class index.
   - Schedules the updateWeights() method to run periodically.
5. **startUp() Method:**
   - Registers CWEModule as an OFMessageListener for PACKET_IN messages.
6. **receive() Method:**
   - Handles PACKET_IN messages.
   - Parses Ethernet, IPv4, TCP, and data payloads.
   - Calls processMetadata() for HTTP/HTTPS traffic.

7. **processMetadata() Method:**
    - Extracts metadata from the packet data.
    - Creates a Weka DenseInstance from the metadata.
    - Performs ensemble classification using the classifiers.
    - Calculates the weighted score.
    - Makes a final decision based on the threshold.
    - Sends an alert if an attack is detected.
    - Adds the instance to the training data and rebuilds the classifiers.
8. **calculateWeightedScore() Method:**
    - Calculates the weighted average of classifier predictions.
9. **updateWeights() Method:**
    - Updates classifier weights based on their accuracy (simplified).
10. **calculateClassifierAccuracy() Method:**
    - Calculates classifier accuracy (simplified).
11. **sendAlert() Method:**
    - Sends an alert to the CRS module (simplified).

Deployment Instructions:

1. **Install Floodlight:**
    - Download and install Floodlight.
2. **Install Weka:**
    - Add the Weka library to your Floodlight project's classpath.
3. **Write CWE Module:**
    - Create a new Java class CWEModule.java in your Floodlight project.
    - Copy and paste the code into the file.
4. **Compile and Run Floodlight:**
    - Compile and run Floodlight.
5. **Configure P4 Switch:**
    - Configure your P4 switch to send metadata packets to the Floodlight controller.
6. **Generate Traffic:**
    - Generate HTTP/HTTPS traffic, including attack traffic.
7. **Monitor Floodlight Logs:**
    - Monitor the Floodlight logs for alerts.
8. **Integrate with CRS Module:**
    - Implement the sendAlert() method to send alerts to the CRS module via REST API or another communication mechanism.
9. **Train the classifiers:**
    - Gather real network traffic data and label the data as normal traffic or attack traffic.
    - Use the labeled data to train the classifiers.
    - Save the trained models and load them into the application.