Python Implementation of CWE Algorithm – (Ryu Controller)

```
# CWE Algorithm Implementation (Python)- (Ryu Controller)

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet, ethernet, ipv4, tcp
import weka.core.jvm as jvm
from weka.core.converters import ConverterUtils
from weka.core.instances import Instances
from weka.core.dense_instance import DenseInstance
from weka.classifiers.trees import J48, RandomForest
from weka.classifiers.lazy import IBk
from weka.classifiers.functions import SMO
from weka.classifiers.meta import AdaBoostM1
import threading
import time

class CWEApp(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(CWEApp, self).__init__(*args, **kwargs)
        self.classifiers = {
            'KNN': IBk(),
            'DT': J48(),
            'RF': RandomForest(),
            'SVM': SMO(),
            'XGBoost': AdaBoostM1()
        }
        self.weights = {classifier: 1.0 for classifier in self.classifiers}
        self.threshold = 0.7
        self.update_interval = 60
        self.attributes = [
            'srcIP', 'dstIP', 'srcPort', 'dstPort', 'method',
            'payloadSize', 'errorCode', 'anomalyFlag', 'attack'
        ]
        self.training_data = Instances('MetaData', [
            (attr, 'NUMERIC' if attr != 'attack' else ['normal', 'attack'])
            for attr in self.attributes
        ], 0)
        self.training_data.class_index = len(self.attributes) - 1
        self.start_weight_updates()
        jvm.start()

    def start_weight_updates(self):
        def update_weights():
            while True:
                time.sleep(self.update_interval)
                self.update_weights_logic()

        thread = threading.Thread(target=update_weights)
        thread.daemon = True
        thread.start()

    def update_weights_logic(self):
        for classifier_name, classifier in self.classifiers.items():
            accuracy = self.calculate_classifier_accuracy(classifier_name)
            self.weights[classifier_name] *= (1 + accuracy)

    def calculate_classifier_accuracy(self, classifier_name):
        correct = 0
        total = self.training_data.num_instances
        if total == 0:
            return 0

        try:
            classifier = self.classifiers[classifier_name]
            for i in range(max(0, total - 10), total):
                prediction = classifier.classify_instance(self.training_data.instance(i))
                if prediction == self.training_data.instance(i).class_value():
                    correct += 1
        except Exception as e:
            self.logger.error(f"Error calculating accuracy for {classifier_name}: {e}")

        return correct / min(10, total)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]
        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocol(ethernet.ethernet)
        if eth:
            if eth.ethertype == 0x0800:
                ipv4_pkt = pkt.get_protocol(ipv4.ipv4)
                if ipv4_pkt:
                    if ipv4_pkt.proto == 6:
                        tcp_pkt = pkt.get_protocol(tcp.tcp)
                        if tcp_pkt:
                            if tcp_pkt.dst_port == 80 or tcp_pkt.dst_port == 443:
                                payload = msg.data[14 + (ipv4_pkt.header_length * 4) + (tcp_pkt.offset *
4):]
                                self.process_metadata(payload)
    def process_metadata(self, metadata_bytes):
        metadata_str = metadata_bytes.decode('utf-8')
        metadata_values = metadata_str.split(',')

        if len(metadata_values) < 8:
            return

        try:
            metadata = [float(value) for value in metadata_values]
        except ValueError as e:
            self.logger.error(f"Error parsing metadata: {e}")
            return

        instance = DenseInstance(len(self.attributes) - 1)
        for i, value in enumerate(metadata):
            instance.set_value(i, value)
        instance.dataset = self.training_data

        predictions = {}
        confidence_scores = {}
        for classifier_name, classifier in self.classifiers.items():
            try:
                prediction = classifier.distribution_for_instance(instance)[1]
                confidence = classifier.classify_instance(instance)
                predictions[classifier_name] = prediction
                confidence_scores[classifier_name] = confidence
            except Exception as e:
                self.logger.error(f"Error with classifier {classifier_name}: {e}")

        weighted_score = self.calculate_weighted_score(predictions, confidence_scores)

        if weighted_score > self.threshold:
            self.send_alert("Coordinated attack detected", weighted_score)

        instance_with_class = DenseInstance(len(self.attributes))
        for i, value in enumerate(metadata):
            instance_with_class.set_value(i, value)
        instance_with_class.set_value(len(self.attributes) - 1, 'attack' if weighted_score >
self.threshold else 'normal')
        instance_with_class.dataset = self.training_data
        self.training_data.add(instance_with_class)

        for classifier in self.classifiers.values():
            classifier.build_classifier(self.training_data)

    def calculate_weighted_score(self, predictions, confidence_scores):
        weighted_score = 0.0
        total_weight = 0.0
        for classifier_name, prediction in predictions.items():
            weighted_score += prediction * self.weights[classifier_name] *
confidence_scores[classifier_name]
            total_weight += self.weights[classifier_name]
        return weighted_score / total_weight if total_weight > 0 else 0.0

    def send_alert(self, message, score):
        self.logger.info(f"Alert: {message} (Score: {score})")
       # Implement logic to send alert to CRS module
```

Explanation of the Python Code:

1. **Imports:** Imports Ryu, packet, and Weka libraries.
2. **Class Definition:** CWEApp inherits from app_manager.RyuApp.
3. **Variables:** classifiers, weights, threshold, update_interval, training_data: CWE algorithm parameters.
4. **__init__() Method:**
    - Initializes classifiers (KNN, DT, RF, SVM, XGBoost) and sets initial weights.
    - Sets the attack detection threshold and weight update interval.
    - Defines the structure of metadata attributes and creates a Weka Instances object.
    - Starts a background thread for periodic weight updates.
    - Starts the JVM for Weka.
5. **start_weight_updates() Method:** Creates and starts a thread for periodic weight updates.
6. **update_weights_logic() Method:** Updates classifier weights based on their accuracy.
7. **calculate_classifier_accuracy() Method:** Calculates classifier accuracy by comparing predictions to training data.
8. **switch_features_handler() Method:** Handles switch feature events and installs a default flow to send packets to the controller.
9. **add_flow() Method:** Installs a flow rule on the switch.
10. **_packet_in_handler() Method:**
    - Handles packet-in events.
    - Parses Ethernet, IPv4, and TCP headers.
    - Calls process_metadata() for HTTP/HTTPS traffic (TCP ports 80 or 443).
11. **process_metadata() Method:**
    - Extracts metadata from the packet payload.
    - Creates a Weka DenseInstance from the metadata.
    - Performs ensemble classification using the Weka classifiers.
    - Calculates the weighted score based on classifier predictions and weights.
    - Makes a decision based on the threshold and sends an alert if an attack is detected.
    - Adds the processed instance to the training data and rebuilds the classifiers.
12. **calculate_weighted_score() Method:** Calculates the weighted average of classifier predictions.
13. **send_alert() Method:** Sends an alert message to the log (and can be modified to send to a CRS).

Deployment Instructions:

Emulated Environment (Mininet-WiFi):

1. **Install Prerequisites:**
    - Python 3.
    - Ryu: pip install ryu
    - Python Weka Wrapper: pip install python-weka-wrapper3
    - Mininet-WiFi.
2. **Create Ryu Application:** Create a Python file (e.g., cwe_app.py) and paste the application code.
3. **Run Ryu:** Start Ryu using ryu-manager cwe_app.py.
4. **Create Mininet-WiFi Topology:** Create a Mininet-WiFi topology with a P4 software switch.
5. **Configure P4 Switch:** Configure the P4 switch to send metadata packets to the Ryu controller.
6. **Generate Traffic:** Generate HTTP/HTTPS traffic using tools like hping3 or curl.
7. **Monitor Ryu Logs:** Monitor the Ryu controller's console logs for alerts.

Real-World Environment:

1. **Install Ryu:** Install Ryu on a server.
2. **P4 Hardware:** Use a P4-programmable switch.
3. **Network Configuration:** Configure the network to forward traffic to the P4 switch.
4. **Run Ryu Application:** Run the Ryu application.
5. **Training:** Train the Weka classifiers with real network data.
6. **CRS Integration:** Implement send_alert() to send alerts to the CRS module.
7. **Monitoring:** Monitor Ryu logs and the CRS module for alerts.
8. **Performance and Security:** Adjust parameters, optimize classifiers, and secure the system.
9. **Maintenance:** Regularly update models and monitor the system.