**Full Java Code Implementation of DDM Module on ONOS Multi-Controller System**

|  |
| --- |
| package org.onosproject.ddm;  import org.onosproject.core.ApplicationId;  import org.onosproject.core.CoreService;  import org.onosproject.net.DeviceId;  import org.onosproject.net.flow.FlowRule;  import org.onosproject.net.flow.FlowRuleService;  import org.onosproject.net.packet.PacketContext;  import org.onosproject.net.packet.PacketProcessor;  import org.onosproject.net.packet.PacketService;  import org.onosproject.net.topology.TopologyService;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.onosproject.net.Device;  import java.util.HashMap;  import java.util.Map;  import java.util.Set;  public class DDMController {  private final Logger log = LoggerFactory.getLogger(getClass());  private ApplicationId appId;  private CoreService coreService;  private FlowRuleService flowRuleService;  private PacketService packetService;  private TopologyService topologyService;  private Map<DeviceId, String> controllerKeys = new HashMap<>();  private static final String APP\_NAME = "org.onosproject.ddm";  // Initialize the DDM module  public void init\_state() {  appId = coreService.registerApplication(APP\_NAME);  log.info("DDM Module initialized with Application ID: {}", appId.id());  }  // Main function to start the DDM algorithm  public void DDM() {  init\_state();  while (true) {  monitor\_network();  }  }  // Function to monitor network for DDoS attack detection  private void monitor\_network() {  log.info("Monitoring network for anomalies...");  // Placeholder for network monitoring logic  boolean attackDetected = detect\_attack();  if (attackDetected) {  Map<String, Object> attack\_info = extract\_attack\_info();  isolate\_network(attack\_info);  broadcast\_alert(attack\_info);  Set<String> mitigation\_actions = select\_mitigation(attack\_info);  distribute\_mitigation(mitigation\_actions);  enforce\_mitigation(mitigation\_actions);  evaluate\_mitigation();  update\_mitigation(mitigation\_actions);  log\_metrics();  }  }  // Function to detect a DDoS attack  private boolean detect\_attack() {  // Placeholder for DDoS detection logic  log.info("Detecting potential DDoS attack...");  return true;  }  // Function to extract attack information  private Map<String, Object> extract\_attack\_info() {  log.info("Extracting attack information...");  Map<String, Object> attack\_info = new HashMap<>();  // Populate attack\_info with collected data  attack\_info.put("type", "DDoS");  attack\_info.put("intensity", "High");  attack\_info.put("affected\_components", "Core Switches");  return attack\_info;  }  // Function to isolate compromised network segments  private void isolate\_network(Map<String, Object> attack\_info) {  log.info("Isolating network segments affected by the attack...");  // Placeholder for network isolation logic  }  // Function to broadcast alert to all controllers  private void broadcast\_alert(Map<String, Object> attack\_info) {  log.info("Broadcasting alert to all controllers...");  // Placeholder for alert broadcasting logic  }  // Function to select appropriate mitigation strategies  private Set<String> select\_mitigation(Map<String, Object> attack\_info) {  log.info("Selecting appropriate mitigation strategies...");  Set<String> mitigation\_actions = Set.of("Rate Limiting", "Flow Blocking");  // Placeholder for mitigation selection logic  return mitigation\_actions;  }  // Function to distribute mitigation actions across controllers  private void distribute\_mitigation(Set<String> mitigation\_actions) {  log.info("Distributing mitigation actions across controllers...");  // Placeholder for mitigation distribution logic  }  // Function to enforce mitigation actions on the network  private void enforce\_mitigation(Set<String> mitigation\_actions) {  log.info("Enforcing mitigation actions...");  for (String action : mitigation\_actions) {  log.info("Enforcing action: {}", action);  // Placeholder for enforcing specific actions  }  }  // Function to evaluate the effectiveness of the mitigation actions  private void evaluate\_mitigation() {  log.info("Evaluating the effectiveness of the applied mitigation actions...");  // Placeholder for evaluation logic  }  // Function to update mitigation strategies based on ongoing network conditions  private void update\_mitigation(Set<String> mitigation\_actions) {  log.info("Updating mitigation strategies based on current network status...");  // Placeholder for updating mitigation logic  }  // Function to log all metrics related to the mitigation process  private void log\_metrics() {  log.info("Logging metrics related to the mitigation process...");  // Placeholder for logging logic  }  } |

To implement the Dynamic DDoS Mitigation (DDM) module on an ONOS multi-controller system, we'll need to develop a Java-based ONOS application that interacts with the network's data plane, leveraging ONOS's APIs for real-time monitoring, decision-making, and mitigation distribution.

**Detailed Breakdown of the Code**

1. **Initialization Phase (init\_state):**
   * Registers the application with ONOS and initializes the state of the DDM module.
2. **Main DDM Function (DDM):**
   * Contains the continuous loop where the network is constantly monitored for DDoS attacks.
   * The monitoring process involves checking network traffic for anomalies indicative of DDoS attacks.
3. **Network Monitoring (monitor\_network):**
   * Establish network monitoring. If an attack is detected, it triggers the attack mitigation process.
4. **Attack Information Extraction (extract\_attack\_info):**
   * Extracts and returns attack information such as type, intensity, and affected network components.
5. **Network Isolation (isolate\_network):**
   * Isolates affected network segments based on the extracted attack information to prevent further damage.
6. **Broadcasting Alerts (broadcast\_alert):**
   * Sends an alert to all controllers using the Trusted Communication Channel (TCC).
7. **Mitigation Selection (select\_mitigation):**
   * Chooses appropriate mitigation strategies based on attack information. In this example, strategies include rate limiting and flow blocking.
8. **Mitigation Distribution (distribute\_mitigation):**
   * Distributes the selected mitigation actions across multiple controllers in the network.
9. **Mitigation Enforcement (enforce\_mitigation):**
   * Enforces the selected mitigation strategies by applying them to the network's P4 switches.
10. **Mitigation Evaluation (evaluate\_mitigation):**
    * Assesses the effectiveness of the applied mitigation strategies to determine if they are working as expected.
11. **Mitigation Update (update\_mitigation):**
    * Updates the mitigation strategies based on ongoing network conditions and attack trends.
12. **Logging Metrics (log\_metrics):**
    * Logs all relevant metrics and data related to the DDoS mitigation process for future analysis and refinement.

**Integration and Deployment**

1. **Compile and Deploy:** Package the Java code into an ONOS application using Maven.
2. **Install on ONOS:** Deploy the application on our ONOS cluster using the ONOS CLI and REST API.
3. **Monitor and Adjust:** Once deployed, monitor the performance of the DDM module and make necessary adjustments to the detection and mitigation logic based on real-world traffic patterns and attack scenarios.

**Helper Functions Employed in DDM Algorithm**

|  |
| --- |
| function init\_state()  // Retrieve P4 switch state  // Initialize mitigation\_response list  end  function monitor\_network()  // Continuously monitor network using P4 switches  end  function detect\_attack(attack\_data, network\_status)  // Check for attack signatures in network traffic  // Compare against attack patterns  return bool // True if attack detected  end  function extract\_attack\_info(attack\_data)  // Extract attack type, intensity, affected components  return attack\_info  end  function isolate\_network(attack\_info)  // Divert traffic, identify impacted nodes  end  function broadcast\_alert(attack\_info)  // Send alert to controllers using TCC  end  function select\_mitigation(attack\_info, prediction\_model, thresholds)  // Assess attack severity, predict trend  // Select mitigation actions based on prediction and thresholds  return mitigation\_actions  end  function distribute\_mitigation(mitigation\_actions, multi\_controller\_config)  // Assign mitigation actions to controllers  end  function enforce\_mitigation(mitigation\_actions)  // Implement actions on P4 switches  end  function evaluate\_mitigation(network\_status, attack\_data)  // Assess mitigation effectiveness  return evaluation\_result  end  function update\_mitigation(mitigation\_actions, network\_status, attack\_data)  // Adjust mitigation actions based on network status and attack trends  end  function log\_metrics()  // Record mitigation actions, network status, attack trends  end |

**Explanation and Configuration**

**1. Phase 1: Establishing the Structure of State Tables**

* state\_tables is defined to track IoT traffic based on selected features. It has two actions: update\_state and drop.

**2. Phase 2: Managing State Transitions and Logical Operations**

* The transition\_state action is a placeholder for dynamic state transitions, which should be handled in the control plane.

**3. Phase 3: Utilizing P4-Counters and P4-Registers**

* Counters (packetCount and byteCount) are defined to track network activity. These counters are updated in the ingress control flow.

**4. Phase 4: Implementing Network Functionality using P4**

* The ingress and egress controls are configured to process packets through the state tables. Packet parsing and metadata management occur in the ingress control flow.

**5. Phase 5: Defining Packet Processing Logic**

* Memory allocation and resource assignment are handled outside the P4 program, typically in the switch’s control plane or management system.

**6. Phase 6: Comprehensive Network Monitoring**

* Holistic monitoring is achieved through aggregated data from state tables and counters, usually by an external monitoring system or controller.

**7. Phase 7: Empowering the Threat Hunter**

* Aggregated data are fed into an intrusion detection system, deployed at the cloud server, which is not directly implemented in the P4 program.

**8. Phase 8: Security System Validation and Optimization**

* Testing and refinement are performed outside the P4 program, focusing on performance evaluation and optimization based on traffic scenarios.

**Deployment and Setup**

1. **Compile and Deploy**: we used the “p4c” P4 compiler to compile the P4 code into a format compatible with our P4 switch.
2. **Load onto Switch**: Deployed the compiled program to our P4-enabled switch using the switch’s management interface.
3. **Control Plane Integration**: Integrate with the switch’s control plane and an SDN controller to manage and monitor state tables and counters.
4. **Monitor and Adjust**: Used external monitoring system (SFlow-RT) to aggregate data, evaluate performance, and refine configurations as needed.