The P4-Enabled Stateful Traffic Monitoring (P4-STM) module can be implemented using the P4 programming language, designed for use on programmable data plane switches. Below is a step-by-step guide for writing and deploying the P4 code for the P4-STM module. This code will implement the core components of the P4-STM module, including match-action tables, state tables, header parsing, and traffic feature extraction.

**1. Header Definitions**

We define the common headers, such as Ethernet, IPv4, and TCP/UDP, which are necessary for identifying and extracting features from IoT traffic.

|  |
| --- |
| **// Define standard headers (Ethernet, IPv4, TCP)**  **header ethernet\_t {**  **bit<48> dstAddr;**  **bit<48> srcAddr;**  **bit<16> etherType;**  **}**  **header ipv4\_t {**  **bit<4> version;**  **bit<4> ihl;**  **bit<8> diffserv;**  **bit<16> totalLen;**  **bit<16> identification;**  **bit<3> flags;**  **bit<13> fragOffset;**  **bit<8> ttl;**  **bit<8> protocol;**  **bit<16> hdrChecksum;**  **bit<32> srcAddr;**  **bit<32> dstAddr;**  **}**  **header tcp\_t {**  **bit<16> srcPort;**  **bit<16> dstPort;**  **bit<32> seqNo;**  **bit<32> ackNo;**  **bit<4> dataOffset;**  **bit<6> reserved;**  **bit<6> flags;**  **bit<16> window;**  **bit<16> checksum;**  **bit<16> urgentPtr;**  **}**  **header udp\_t {**  **bit<16> srcPort;**  **bit<16> dstPort;**  **bit<16> length;**  **bit<16> checksum;**  **}** |

**2. Metadata Definition**

Metadata is used for storing information about extracted traffic features.

|  |
| --- |
| **struct metadata\_t {**  **bit<32> flow\_id;**  **bit<32> packet\_size;**  **bit<64> timestamp;**  **}** |

**3. Parser**

The parser processes the incoming packets and extracts the necessary headers. It focuses on Ethernet, IPv4, and TCP/UDP headers.

|  |
| --- |
| **parser MyParser(packet\_in packet,**  **out headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **state start {**  **transition parse\_ethernet;**  **}**  **state parse\_ethernet {**  **packet.extract(hdr.ethernet);**  **transition select(hdr.ethernet.etherType) {**  **0x0800: parse\_ipv4; // IPv4**  **default: accept;**  **}**  **}**  **state parse\_ipv4 {**  **packet.extract(hdr.ipv4);**  **transition select(hdr.ipv4.protocol) {**  **6: parse\_tcp; // TCP**  **17: parse\_udp; // UDP**  **default: accept;**  **}**  **}**  **state parse\_tcp {**  **packet.extract(hdr.tcp);**  **transition accept;**  **}**  **state parse\_udp {**  **packet.extract(hdr.udp);**  **transition accept;**  **}**  **}** |

**4. Match-Action Tables and Stateful Tables**

State tables are used for storing flow information, such as packet counts and timestamps. Match-action tables will match packets based on specific features (e.g., source IP, destination IP, protocol).

|  |
| --- |
| **// Define a state table to store flow information**  **table state\_table {**  **key = {**  **hdr.ipv4.srcAddr : exact;**  **hdr.ipv4.dstAddr : exact;**  **hdr.ipv4.protocol : exact;**  **hdr.tcp.srcPort : exact;**  **hdr.tcp.dstPort : exact;**  **}**  **actions = {**  **update\_state\_table;**  **create\_new\_entry;**  **}**  **size = 1024;**  **default\_action = create\_new\_entry;**  **}**  **// Action to update an existing entry in the state table**  **action update\_state\_table(bit<32> flow\_id, bit<32> packet\_size, bit<64> timestamp) {**  **// Update flow details, such as packet count and timestamp**  **// Example: Increment packet count**  **register\_flow\_count[flow\_id] += 1;**  **register\_last\_seen[flow\_id] = timestamp;**  **}**  **// Action to create a new entry in the state table**  **action create\_new\_entry() {**  **bit<32> flow\_id = hash(hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.ipv4.protocol);**  **register\_flow\_count[flow\_id] = 1;**  **register\_last\_seen[flow\_id] = meta.timestamp;**  **}** |

**5. Ingress Processing**

The ingress processing logic will extract traffic features, look up the state table, and update or create entries based on packet information.

|  |
| --- |
| **control IngressControl(inout headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **apply {**  **// Extract flow features (source/destination IP, packet size, etc.)**  **meta.packet\_size = standard\_meta.ingress\_port;**  **meta.timestamp = standard\_meta.ingress\_global\_timestamp;**  **// Perform match-action using the state table**  **if (hdr.ipv4.isValid() && hdr.tcp.isValid()) {**  **state\_table.apply();**  **}**  **}**  **}** |

**6. Egress Processing**

Egress processing can be defined to handle additional logic if required, such as sending alert messages or logging traffic details.

|  |
| --- |
| **control EgressControl(inout headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **apply {**  **// Additional egress processing logic can be added here**  **}**  **}** |

**7. Deparser**

The deparser reconstructs the packet before sending it out.

|  |
| --- |
| **control Deparser(packet\_out packet,**  **in headers\_t hdr) {**  **apply {**  **packet.emit(hdr.ethernet);**  **if (hdr.ipv4.isValid()) {**  **packet.emit(hdr.ipv4);**  **}**  **if (hdr.tcp.isValid()) {**  **packet.emit(hdr.tcp);**  **} else if (hdr.udp.isValid()) {**  **packet.emit(hdr.udp);**  **}**  **}**  **}** |

**8. Pipeline Control**

Finally, we tie all the components together in the main pipeline, defining the flow from parsing to egress.

|  |
| --- |
| **control MyPipeline(inout headers\_t hdr,**  **inout metadata\_t meta,**  **inout standard\_metadata\_t standard\_meta) {**  **MyParser() parser;**  **IngressControl() ingress;**  **EgressControl() egress;**  **Deparser() deparser;**  **apply {**  **parser.apply();**  **ingress.apply();**  **egress.apply();**  **deparser.apply();**  **}**  **}** |

**9. Configuration**

The *MySwitch* block defines how the switch will use the pipeline.

|  |
| --- |
| **package MySwitch(IngressControl ingress,**  **EgressControl egress,**  **MyParser parser,**  **Deparser deparser) {**  **parser parser;**  **control ingress;**  **control egress;**  **deparser deparser;**  **}**  **MySwitch() main;** |

**Step-by-Step Deployment Instructions for the P4-STM Module**

1. **Set Up the Environment**
   * Ensure that all the required tools installed, including the following:
   * P4 development environment (using p4c compiler)
   * Mininet (for simulating network topology)
   * BMv2 (Behavioral Model v2) switch
   * Python runtime for control plane communication (e.g., *simple\_switch\_grpc*)
2. **Write the P4 Program**
   * Save the above P4 code in a file called p4\_stm.p4.
3. **Compile the P4 Program**
   * Use the P4 compiler (p4c) to compile the p4\_stm.p4 program into a format that can be loaded onto the BMv2 switch.
   * *p4c --target bmv2 --arch v1model --std p4-16 p4\_stm.p4 -o p4\_stm.json*
   * This will generate a p4\_stm.json file that can be loaded onto the BMv2 switch.
4. **Set Up a Simple Network Using Mininet**
   * Set up a simple network topology using Mininet. This step requires writing a Python script that uses Mininet’s API

|  |
| --- |
| **from mininet.net import Mininet**  **from mininet.node import RemoteController**  **from mininet.link import TCLink**  **net = Mininet(controller=RemoteController, link=TCLink)**  **# Add switches and hosts**  **s1 = net.addSwitch('s1')**  **h1 = net.addHost('h1', ip='10.0.0.1')**  **h2 = net.addHost('h2', ip='10.0.0.2')**  **# Add links between hosts and switch**  **net.addLink(h1, s1)**  **net.addLink(h2, s1)**  **# Add a controller**  **net.addController('c0', controller=RemoteController, ip='127.0.0.1', port=6653)**  **# Start the network**  **net.start()**  **# Test connectivity**  **net.pingAll()**  **# Keep Mininet running**  **net.interact()** |

* + Save the script as mininet\_topo.py and run it using: *sudo python3 mininet\_topo.py*

1. **Load the P4 Program onto the Switch**
   * Load the compiled *p4\_stm.json* file onto the BMv2 switch using the *simple\_switch* runtime.
   * *sudo simple\_switch --log-console --thrift-port 9090 p4\_stm.json*

1. **Verify Traffic Monitoring**
   * Once the program is running, use packet generation tools (e.g., *hping3, iperf*) to send traffic between hosts, and verify the traffic monitoring module is capturing and processing packets according to the P4-STM implementation.