P4 Implementation of P4-EWTP Algorithm - Real-time Attack Detection and Alerting

```
// P4-EWTP Algorithm Implementation (P4) - Real-time Attack Detection and Alerting

#include <core.p4>
#include <v1model.p4>

/* Header Definitions */
header ethernet_t {
    bit dstAddr;
    bit srcAddr;
    bit etherType;
}

header ipv4_t {
    bit version;
    bit ihl;
    bit diffserv;
    bit totalLen;
    bit identification;
    bit flags;
    bit fragOffset;
    bit ttl;
    bit protocol;
    bit hdrChecksum;
    bit srcAddr;
    bit dstAddr;
}

header tcp_t {
    bit srcPort;
    bit dstPort;
    bit seqNo;
    bit ackNo;
    bit dataOffset;
    bit reserved;
    bit ns;
    bit cwr;
    bit ece;
    bit urg;
    bit ack;
    bit psh;
    bit rst;
    bit syn;
    bit fin;
    bit windowSize;
    bit checksum;
    bit urgentPtr;
}

header http_t {
    bit method; // Simplified HTTP method representation
    bit payloadSize;
    string userAgent;
    string referrer;
    bit errorCode;
    string requestParams;
    string cookies;
}

header metadata_t {
    bit srcIP;
    bit dstIP;
    bit srcPort;
    bit dstPort;
    bit method;
    bit payloadSize;
    string userAgent;
    string referrer;
    bit errorCode;
    string requestParams;
    string cookies;
    bit anomalyFlag;
}

/* Parser */
parser MyParser {
    state start {
        transition parseEthernet;
    }

    state parseEthernet {
        extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            0x0800: parseIPv4;
            default: ingress;
        }
    }

    state parseIPv4 {
        extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol) {
            6: parseTCP;
            default: ingress;
        }
    }

    state parseTCP {
        extract(hdr.tcp);
        transition select(hdr.tcp.dstPort) {
            80: parseHTTP;
            443: parseHTTPS;
            default: ingress;
        }
    }

    state parseHTTP {
        extract(hdr.http);
        transition ingress;
    }

    state parseHTTPS {
        extract(hdr.http); // Simplified, assuming HTTPS headers are similar
        transition ingress;
    }
}

/* Tables */
table RequestRateTable {
    key = {
        hdr.ipv4.srcAddr: exact;
        hdr.ipv4.dstAddr: exact;
        hdr.tcp.srcPort: exact;
        hdr.tcp.dstPort: exact;
    }
    actions = {
        incrementCounter;
        createEntry;
        NoAction;
    }
    size = 1024;
}

table MethodTable {
    key = {
        hdr.ipv4.dstAddr: exact;
    }
    actions = {
        incrementMethodCounter;
        createMethodEntry;
        NoAction;
    }
    size = 256;
}

table PayloadSizeTable {
    key = {
        hdr.ipv4.srcAddr: exact;
        hdr.ipv4.dstAddr: exact;
        hdr.tcp.srcPort: exact;
        hdr.tcp.dstPort: exact;
    }
    actions = {
        incrementPayloadSizeCounter;
        createPayloadSizeEntry;
        NoAction;
    }
    size = 1024;
}

table UserAgentTable {
    key = {
        hdr.http.userAgent: exact;
    }
    actions = {
        incrementUserAgentCounter;
        createUserAgentEntry;
        NoAction;
    }
    size = 256;
}

table ErrorCodeTable {
    key = {
        hdr.ipv4.dstAddr: exact;
    }
    actions = {
        incrementErrorCodeCounter;
        createErrorCodeEntry;
        NoAction;
    }
    size = 256;
}

table URLRequestTable {
    key = {
        hdr.ipv4.dstAddr: exact;
        hdr.http.requestParams: exact; // Simplified URL representation
    }
    actions = {
        incrementURLRequestCounter;
        createURLRequestEntry;
        NoAction;
    }
    size = 1024;
}

table CookieTable {
    key = {
        hdr.http.cookies: exact;
    }
    actions = {
        incrementCookieCounter;
        createCookieEntry;
        NoAction;
    }
    size = 256;
}

/* Actions */
action incrementCounter() {
    register.requestRate = register.requestRate + 1;
}

action createEntry() {
    register.requestRate = 1;
}

action incrementMethodCounter() {
    register.method[hdr.http.method] = register.method[hdr.http.method] + 1;
}

action createMethodEntry() {
    register.method[hdr.http.method] = 1;
}

action incrementPayloadSizeCounter() {
    register.payloadSize = register.payloadSize + hdr.http.payloadSize;
}

action createPayloadSizeEntry() {
    register.payloadSize = hdr.http.payloadSize;
}

action incrementUserAgentCounter() {
    register.userAgent = register.userAgent + 1;
}

action createUserAgentEntry() {
    register.userAgent = 1;
}

action incrementErrorCodeCounter() {
    register.errorCode[hdr.http.errorCode] = register.errorCode[hdr.http.errorCode] + 1;
}

action createErrorCodeEntry() {
    register.errorCode[hdr.http.errorCode] = 1;
}

action incrementURLRequestCounter() {
    register.urlRequest = register.urlRequest + 1;
}

action createURLRequestEntry() {
    register.urlRequest = 1;
}

action incrementCookieCounter() {
    register.cookie = register.cookie + 1;
}

action createCookieEntry() {
    register.cookie = 1;
}

action sendAlert(bit dip, string message) {
    meta.anomalyFlag = 1;
}

action NoAction() {}

/* Control Block */
control ingress {
    apply(RequestRateTable);
    apply(MethodTable);
    apply(PayloadSizeTable);
    apply(UserAgentTable);
    apply(ErrorCodeTable);
    apply(URLRequestTable);
    apply(CookieTable);

    /* Metadata Generation */
    meta.srcIP = hdr.ipv4.srcAddr;
    meta.dstIP = hdr.ipv4.dstAddr;
    meta.srcPort = hdr.tcp.srcPort;
    meta.dstPort = hdr.tcp.dstPort;
    meta.method = hdr.http.method;
    meta.payloadSize = hdr.http.payloadSize;
    meta.userAgent = hdr.http.userAgent;
    meta.referrer = hdr.http.referrer;
    meta.errorCode = hdr.http.errorCode;
    meta.requestParams = hdr.http.requestParams;
    meta.cookies = hdr.http.cookies;
    meta.anomalyFlag = 0; // Initialize anomaly flag

}

/* Anomaly Detection Logic (Simplified) */
    if (register.requestRate > RR_TH) {
        sendAlert(hdr.ipv4.dstAddr, "High Request Rate");
    }

    if (register.errorCode[500] > ECF_TH) { // Example: Check for 500 errors
        sendAlert(hdr.ipv4.dstAddr, "High Error Code Frequency");
    }

    // Example: Check for unusual method distribution (simplified)
    if (register.method[1] > METHOD_GET_TH && register.method[3]> METHOD_PUT_TH) { // Example GET and
```

```
PUT
        sendAlert(hdr.ipv4.dstAddr, "Suspicious HTTP Methods");
    }

    /* Send Metadata */
    emit(meta);

    // Clear Registers (Simplified, would need a timer for real implementation)
    register.requestRate = 0;
    register.payloadSize = 0;
    for (bit i = 0; i < 256; i = i + 1) {
        register.errorCode[i] = 0;
        register.method[i] = 0;
    }
    register.userAgent = 0;
    register.urlRequest = 0;
    register.cookie = 0;
}

control egress {
    apply(v1model.default_egress_pipeline);
}

control verifyChecksum {
    apply(v1model.default_verify_checksum);
}

control computeChecksum {
    apply(v1model.default_compute_checksum);
}

control deparser {
    apply(v1model.default_deparser);
}
```

Explanation:

1. **Header Definitions:**
    - Defines the structure of Ethernet, IPv4, TCP, and HTTP headers.
    - metadata_t is used to carry extracted features and anomaly flags to the SDN controller.
2. **Parser:**
    - Parses incoming packets, extracting relevant header fields.
    - Filters out non-HTTP/HTTPS traffic.
3. **Tables:**
    - State tables for storing request rates, method distributions, payload sizes, etc.
    - Uses registers to count values.
4. **Actions:**
    - Actions to increment counters, create table entries, and send alerts.
    - sendAlert action sets the anomalyFlag in the metadata.
5. **Control Block (ingress):**
    - Applies the tables to update state.
    - Extracts features and populates metadata.
    - Implements simplified anomaly detection logic.
    - Clears the registers to simulate the end of a time window.
    - Emits metadata to the SDN controller.

Deployment Instructions:

Emulated Environment (Mininet-WiFi):

1. **Install Prerequisites:**
   - Mininet-WiFi
   - P4 compiler (p4c)
   - Behavioral Model v2 (bmv2)
2. **Write P4 Code:**
   - Save the P4 code into two files (e.g., ewtp_ingress.p4 and ewtp_alert.p4).
3. **Compile P4 Code:**
   - Use p4c to compile the P4 code into a JSON file that bmv2 can understand.
   - p4c --target bmv2 --arch v1model ewtp_ingress.p4 --output ewtp_ingress.json
   - p4c --target bmv2 --arch v1model ewtp_alert.p4 --output ewtp_alert.json
4. **Start Mininet-WiFi:**
   - Create a Mininet-WiFi topology with P4 switches.
   - Use the --p4runtime-files option to load the compiled JSON files onto the switches.
5. **Configure SDN Controller:**
   - Write an SDN controller application to receive metadata from the switches.
   - Implement logic to handle alerts and enforce mitigation policies.
6. **Test the Setup:**
   - Generate attack traffic using tools like hping3 or curl.
   - Verify that the P4 switches detect anomalies and send alerts to the controller.

Real-World Environment:

1. **P4-Capable Hardware:**
   - Use a P4-programmable switch that supports the v1model architecture.
2. **P4 Compiler:**
   - Use the appropriate P4 compiler for your target hardware.
3. **Deployment Tools:**
   - Use the vendor-provided tools to deploy the compiled P4 program onto the switch.
4. **SDN Controller:**
   - Use an SDN controller that supports P4Runtime or the vendor's API.
5. **Network Configuration:**
   - Configure the switch and network devices to forward traffic to the P4 switch.
6. **Testing:**
   - Generate real-world attack traffic and monitor the switch and controller for alerts.
   - Use network monitoring tools to verify the traffic flow.