# Contents

# IPFire-Wall: Consolidated Project Report

*Generated on: Thu Feb 5 02:52:07 PM CET 2026*

# Chapter 1: Refactoring & Optimization

This chapter documents the structural and performance improvements implemented during the refactoring phase of the IPFire-Wall project.

## 1.1. Modular Source Reorganization

The module was transitioned from a flat, monolithic directory structure to a specialized modular hierarchy to improve maintainability and codebase clarity.

### Directory Mapping

| Subdirectory | Responsibility |
|---|---|
| `common/` | Shared data structures, constants, and global utility functions. |
| `filter/` | The core filtering engine, rule traversal logic, and stateful hooks. |
| `filter/state/` | State machine implementation and state table management. |
| `nat/` | Translation logic for DNAT, SNAT, and Masquerade. |
| `netlink/` | Communication layer between the kernel and userspace commands. |
| `proc/` | Implementation of the `/proc/ipfire` interface. |
| `includes/` | Centralized repository for all internal headers. |

## 1.2. Per-CPU Statistics Optimization

To ensure maximum performance on multi-core systems, the statistics tracking was moved from global atomics (which cause cache-line bouncing) to purely per-CPU counters.

- **Infrastructure**: Counters are instances of `struct ipfi_counters` allocated via `alloc_percpu`.
- **Latency Reduction**: Updates in the network hot path (like `INPUT` or `FORWARD`) use the `IPFI_STAT_INC` macro, which translates to a single instruction per-core increment.
- **On-Demand Aggregation**: Global totals are only computed when a user runs `ipfire -s`, at which point the kernel iterates over all online CPUs and sums the values.

### 1.3. Netlink Protocol Simplification

The Netlink message structure was streamlined by removing legacy per-packet sequence IDs (`packet_id` and `logu_id`).

- **Rationale**: Sequential IDs created a bottleneck for parallel packet processing.
- **Reporting Transition**: Reliability is now monitored by the kernel itself. If a Netlink send fails (e.g., due to a full buffer), the kernel increments a `total_lost` counter. The userspace app now requests this counter on-demand rather than inferring loss from gaps in sequence IDs.

### 1.4. Build System Modernization

A shadow-tree build system was implemented using a `build/` directory. - **Feature**: Source files are symlinked into `build/` before compilation. - **Benefit**: All intermediate artifacts (`.o`, `.mod`, etc.) are hidden from the main source tree, keeping the environment clean for development.

---

## Chapter 2: Kernel Module Architecture

This chapter explains the internal workings of the IPFire-Wall kernel module, its integration with Netfilter, and how it evaluates security policies.

### 2.1. Netfilter Hook Integration

IPFire-Wall registers itself at several points in the Linux network stack (Hooks).

```
graph TD
    In[Packet In] --> Pre[PRE_ROUTING: DNAT/De-SNAT]
    Pre --> Routing{Routing Decision}
    Routing -->|Local| InHook[LOCAL_IN: Input Filter]
    Routing -->|Forward| FwdHook[FORWARD: Transit Filter]
    InHook --> App[Local Application]
    App --> OutHook[LOCAL_OUT: Output Filter]
    FwdHook --> Post[POST_ROUTING: SNAT/Masquerade]
    OutHook --> Post
    Post --> Out[Packet Out]
```

### 2.2. Rule Hierarchy & Evaluation Logic

When a packet hits a filtering hook (`LOCAL_IN`, `LOCAL_OUT`, or `FORWARD`), it is evaluated against several lists of rules in a prioritized sequence.

**The Priority Chain:**

1. **Blacklist (Dropped Rules)**: The engine first scans the `dropped` list. If a match is found, the packet is discarded immediately. This ensures that blocked entities cannot bypass later permission checks.
2. **Administrator Rules (Root)**: The `allowed` list is scanned for rules inserted by the system administrator.
3. **User Rules**: If enabled, rules defined by non-root users are checked next.

**Evaluation Mechanism:**

- **First-Match Wins**: The evaluation stops as soon as a rule matches.
- **Default Policy**: If no rule matches any list, the module applies the `default_policy` (configured as either `ACCEPT` or `DROP`).

### 2.3. Filtering Granularity

Rules can match packets based on a wide array of criteria: - **L3 (IP)**: Source/Destination IP addresses and IP Options. - **L4 (Transport)**: Protocol (TCP, UDP, ICMP, IGMP) and Port numbers. - **Direction**: Inbound, Outbound, or Forwarded. - **Interface**: Incoming or Outgoing network device indices. - **Payload**: FTP command inspection (for dynamic NAT).

---

## Chapter 3: Stateful Connection Management

IPFire-Wall employs a stateful inspection engine that tracks the status of network connections to improve both security and performance.

### 3.1. The Stateful Fast-Path

One of the core design goals is to avoid re-evaluating the entire rule list for every packet in a high-volume stream.

- **Slow Path**: The **first packet** (e.g., a TCP SYN) triggers a full scan of the rule lists. If accepted by a stateful rule, an entry is created in the `state_table`.
- **Fast Path**: Lookups for **subsequent packets** are performed against the state table using a hash-based mechanism (`jhash_3words`). This lookup is **O(1)**, meaning it stays fast regardless of how many rules are active.

### 3.2. State Transition Tracking

The module implements a dedicated state machine for different protocols.

**TCP Connection Lifecycle:**

- `SYN_SENT`: Initial request seen.
- `SYN_RECV`: Reply from server seen (SYN/ACK).
- `ESTABLISHED`: Handshake complete (ACK seen).
- `FIN_WAIT / CLOSE_WAIT`: Connection termination in progress.

**UDP "Pseudo-States":**

Since UDP is connectionless, the engine creates virtual states (`UDP_NEW` -> `UDP_ESTABLISHED`) to allow return traffic (e.g., DNS responses) through the firewall for a defined period.

### 3.3. Table Management

- **Lookups**: Perform bidirectional hashing. Both sides of a connection (`A:port1 <-> B:port2` and `B:port2 <-> A:port1`) produce the same hash key, allowing consistent tracking of bidirectional flows.
- **Lifetimes**: Every state entry has an associated kernel timer. If no traffic is seen for a specific duration (e.g., 3600s for ESTABLISHED TCP, or ~30s for UDP), the entry is automatically purged to free resources.
- **Capacity**: The firewall enforces a `max_state_entries` limit to prevent resource exhaustion attacks.

### 3.4. FTP Support

The engine includes a specialized parser for the FTP protocol. It monitors the "control" channel for `PASV` commands and dynamically injects "Data Channel" states, allowing passive FTP to function through the NAT without requiring manual rule openings for high-numbered ports.

---

## Chapter 4: Userspace Application (ipfire)

The `ipfire` utility is the primary administrative tool for interacting with the kernel-space firewall module.

### 4.1. Core Usage & CLI Flags

The application communicates with the kernel via Netlink sockets using the `IPFI_CONTROL` and `IPFI_DATA` protocols.

| Flag | Action | Description |
|---|---|---|
| `-v` | Version / Status | Shows if the module is loaded and current global settings. |
| `-s` | Statistics | Displays a combined report of kernel per-CPU counters and userspace logging counts. |
| `-X` | Flush | Clears all current filtering rules and resets the state table. |
| `-p <policy>` | Set Policy | Changes the default policy to `accept` or `drop`. |
| `-a <rule>` | Add Rule | Inserts a new rule into the appropriate chain. |

### 4.2. Configuration Files

The application behavior can be customized via config files, typically located in `/etc/ipfire/`.

- `allowed.base`: List of rules to be automatically loaded on startup.
- `ipfire.conf`: Global options such as logging levels, max NAT entries, and stateful tracking defaults.

### 4.3. Interpreting Statistics

The `-s` (Statistics) output is divided into three sections: 1. **Userspace Stats**: Counts of packets actually received and displayed by the tool. Useful for auditing. 2. **Kernel Stats**: High-level counters for `INPUT`, `OUTPUT`, `FORWARD`, and `POST-ROUTING`. 3. **Transmission Health**: Specifically reports `total_lost` (packets the kernel tried to log but couldn't due to buffer pressure).

### 4.4. Logging and Real-time Monitoring

When running, `ipfire` can act as a listener, printing headers for every packet matched by a rule with the `NOTIFY` flag. These logs include: - Timestamp and user ID. - Hook location and verdict (ACCEPT/DROP). - Detailed IP/TCP/UDP header information.

---

## Chapter 5: Packet Flow Walkthroughs

This chapter provides step-by-step analysis of how specific real-world packet scenarios traverse the IPFire-Wall engine.

### 5.1. Scenario 1: DNAT to Forwarded Internal Server

An external client connects to the firewall's public IP on port 80, which is redirected to an internal web server.

1. **PRE_ROUTING Hook**:
   - The packet `Src:Client, Dst:Firewall_Public` is intercepted.
   - `dnat_translation()` matches a rule and changes the destination to `Firewall_Internal_Server`.
   - The kernel is notified of the `daddr` change and clears the old route.
2. **Routing Decision**: The kernel finds the route for the new internal destination; it is not local, so the packet moves to the FORWARD hook.
3. **FORWARD Hook**:
   - `ipfire_filter()` is called.
   - It identifies this as a new connection.
   - It matches a stateful rule allowing traffic to the server.
   - `keep_state()` creates an entry in the state table.
4. **POST_ROUTING Hook**: The packet leaves the firewall towards the internal server.

```
sequenceDiagram
    participant C as External Client
    participant P as PRE_ROUTING (DNAT)
    participant F as FORWARD (Filter/State)
    participant S as Internal Server
```

```
C->>P: SYN (Dst: Public IP)
P->>P: Translate Dst -> 192.168.1.50
P->>F: Routed to Forward
F->>F: Match Rule & Create State
F->>S: Transmit SYN (Dst: 192.168.1.50)
```

### 5.2. Scenario 2: Masqueraded Outbound Connection

An internal client connects to the internet; the firewall hides the internal IP.

1. **LOCAL_OUT / FORWARD**: The packet is accepted by the filtering engine.
2. **POST_ROUTING**:
   - `masquerade_translation()` is called.
   - The engine dynamically finds the IP address of the outgoing WAN interface (`get_ifaddr`).
   - The packet's source address is replaced.
   - An entry is added to the `snat_table` to handle return traffic.

### 5.3. Scenario 3: Stateful Return Traffic

The reply from a previously accepted connection arrives.

1. **PRE_ROUTING**: If the original connection was NATed, the reverse translation is applied here.
2. **LOCAL_IN / FORWARD**:
   - `check_state()` is called.
   - A **reverse match** is found in the state table.
   - **Verdict: ACCEPT**. Rule evaluation is skipped entirely.
   - The state machine transitions (e.g., to `SYN_RECV` or `ESTABLISHED`).
3. **Delivery**: The packet is delivered to the initial requestor.

---

# IPFire Packet Flow Walkthrough

This document provides comprehensive walkthroughs of packet flows through the IPFire kernel firewall module for three representative rules from `allowed.base`. Each walkthrough demonstrates different firewall concepts and provides detailed diagrams showing packet traversal through Netfilter hooks and the IPFire filtering engine.

---

## Rule Selection

We analyze three conceptually different rules that showcase the firewall's capabilities:

1. **HTTP Connection (Stateful)** - Basic stateful connection tracking
2. **FTP Control Connection (Passive FTP)** - Dynamic state creation for data channels
3. **SSH Bidirectional Access** - Both INPUT and OUTPUT paths with state tracking

---

## Rule 1: HTTP Connection (Stateful)

### Rule Definition

```
RULE
NAME=me -> www
DIRECTION=OUTPUT
MYSRCADDR
PROTOCOL=6
DSTPORT=80
KEEP_STATE=YES
```

### Scenario

A user on the firewall machine initiates an HTTP connection to a web server at `203.0.113.50:80`.

### Outgoing SYN Packet Flow

```
flowchart TD
    subgraph Userspace
        App([Application]) -->|sendto/write| Socket[Socket Layer]
    end

    subgraph Kernel_Network_Stack [Linux Kernel Network Stack]
        Socket --> TCP[TCP Stack: Create SYN]
        TCP --> Hook_LOCAL_OUT{NF_IP_LOCAL_OUT}
    end

    subgraph IPFire_Engine [IPFire Filtering Engine]
        Hook_LOCAL_OUT -- Packet --> Process[ipfire.c: process]
        Process --> Response[ipfi_response]

        subgraph Logic [Filtering Logic]
            Response --> CheckState{check_state}
            CheckState -- Miss --> Filter[ipfire_filter]
            Filter -- Match: 'me -> www' --> KeepState[keep_state]
        end

        KeepState --> NewEntry[Create state_table entry]
        NewEntry --> Hash[Add to state_hashtable]
    end

    subgraph State_Transitions [State Machine]
        NewEntry --> SYN_SENT[[State: SYN_SENT]]
        SYN_SENT --> Timer[Start Timeout Timer]
    end
```

```
    KeepState --> Verdict[Verdict: IPFI_ACCEPT]
    Verdict --> Accept[NF_ACCEPT]
    Accept --> Hook_POST_ROUTING{NF_IP_POST_ROUTING}
    Hook_POST_ROUTING --> Net((Network))

    %% Styling
    classDef hook fill:#f9f,stroke:#333,stroke-width:2px;
    classDef engine fill:#bbf,stroke:#333,stroke-width:2px;
    classDef state fill:#dfd,stroke:#333,stroke-width:2px;
    classDef match fill:#9f9,stroke:#333,stroke-width:2px;

    class Hook_LOCAL_OUT,Hook_POST_ROUTING hook;
    class Process,Response,Logic,KeepState engine;
    class SYN_SENT state;
    class Verdict match;
```

### Key Operations

1. **Hook Entry**: `NF_IP_LOCAL_OUT` invokes `ipfi_response()` with `flow.direction = IPFI_OUTPUT`
2. **State Check**: `check_state()` searches hash table - no match (new connection)
3. **Rule Matching**: `ipfire_filter()` iterates through permission rules
4. **Rule Match**: Matches `me -> www` (protocol=6, dport=80, MYSRCADDR, OUTPUT)
5. **State Creation**: `keep_state()` allocates new `state_table`:

   ```
   state_table {
       saddr: 192.0.2.100         // Local firewall IP
       daddr: 203.0.113.50        // Web server IP
       sport: 54321               // Ephemeral port
       dport: 80                  // HTTP
       protocol: IPPROTO_TCP
       state: SYN_SENT
       direction: IPFI_OUTPUT
       rule_id: <hash of rule>
   }
   ```
6. **Hash Table**: Entry added to `state_hashtable` using `jhash_3words(saddr, daddr, ports)`
7. **Timer**: Setup timer expires in ~120 seconds (setup/shutdown timeout)
8. **Verdict**: Returns `NF_ACCEPT`

### Returning SYN-ACK Packet Flow

```
flowchart TD
    Net((Network)) --> Hook_PRE{NF_IP_PRE_ROUTING}
```

```
    subgraph IPFire_Pre [IPFire Pre-Processing]
        Hook_PRE --> PreProcess[ipfi_pre_process]
        PreProcess --> DNAT{Check DNAT}
        DNAT -- No Match --> Route[Routing Decision]
    end

    Route -- Local Delivery --> Hook_LOCAL_IN{NF_IP_LOCAL_IN}

    subgraph IPFire_Core [IPFire Core Engine]
        Hook_LOCAL_IN --> ProcessIn[ipfire.c: process]
        ProcessIn --> ResponseIn[ipfi_response]

        subgraph LogicIn [Stateful Lookup]
            ResponseIn --> CheckStateIn{check_state}
            CheckStateIn -- "Hit (Reverse)" --> Entry[Existing state_table]
        end
    end

    subgraph Machine [State Machine]
        Entry --> Transition[state_machine: SYN_SENT -> SYN_RECV]
        Transition --> UpdateTimer[Refresh Timer]
    end

    Entry --> VerdictIn[Verdict: IPFI_ACCEPT]
    VerdictIn --> AcceptIn[NF_ACCEPT]
    AcceptIn --> App([Application])

    %% Styling
    classDef hook fill:#f9f,stroke:#333,stroke-width:2px;
    classDef engine fill:#bbf,stroke:#333,stroke-width:2px;
    classDef state fill:#dfd,stroke:#333,stroke-width:2px;
    classDef match fill:#9f9,stroke:#333,stroke-width:2px;

    class Hook_PRE,Hook_LOCAL_IN hook;
    class PreProcess,ResponseIn,LogicIn engine;
    class Transition,UpdateTimer state;
    class VerdictIn match;
```

**Key Operations**

1. **Reverse Match**: `check_state()` finds entry with **reverse** matching:

   ```
   // Packet has: src=203.0.113.50:80, dst=192.0.2.100:54321
   // State table: saddr=192.0.2.100:54321, daddr=203.0.113.50:80
   reverse_state_match() -> returns 1
   ```

2. **State Machine**: `state_machine()` transitions `SYN_SENT + (SYN|ACK)` → `SYN_RECV`

3. **Timer Update**: `update_timer_of_state_entry()` extends timeout to established connec-

tion timeout (~3600 seconds)

4. **No Rule Check**: Since state matched, `ipfire_filter()` is **not called**
5. **Verdict**: Returns `NF_ACCEPT` based on state match

#### Established Connection Data Flow

```
flowchart TD
    subgraph Traffic [Bidirectional Traffic]
        Pkt[Subsequent Packet] --> Lookup{check_state}
    end

    subgraph Fast_Path [Stateful Fast Path]
        Lookup -- "Match (Direct/Reverse)" --> State[State: ESTABLISHED]
        State --> Refresh[Update Timer]
        Refresh --> Accept[Verdict: IPFI_ACCEPT]
    end

    Accept --> Bypass[[Bypass Rule Evaluation]]
    Bypass --> NF_Accept[NF_ACCEPT]

    %% Styling
    classDef engine fill:#bbf,stroke:#333,stroke-width:2px;
    classDef fast fill:#9f9,stroke:#333,stroke-width:2px;

    class Lookup engine;
    class State,Refresh,Accept,Bypass fast;
```

**Performance Note** Once state is `ESTABLISHED`, all subsequent packets bypass rule evaluation entirely, providing high-performance stateful filtering via hash table lookup.

---

### Rule 2: FTP Control Connection (Passive FTP)

#### Rule Definition

```
RULE
NAME=me -> ftp control
DIRECTION=OUTPUT
MYSRCADDR
PROTOCOL=6
DSTPORT=21
KEEP_STATE=YES
FTP_SUPPORT=YES
```

#### Scenario

User initiates FTP connection to `203.0.113.100:21` and enters passive mode (PASV).

### Control Connection Establishment

The initial FTP control connection follows the same flow as HTTP (Rule 1), with state tracking for `<local>:ephemeral <-> <server>:21`.

### PASV 227 Response Flow

When the server sends a PASV 227 reply like:

```
227 Entering Passive Mode (203,0,113,100,195,210)
```

This encodes data channel endpoint: `203.0.113.100:50130` (195*256 + 210)

```
flowchart TD
    subgraph Control_Connection [Control Channel - state: FTP_LOOK_FOR]
        PacketIn[Packet with '227' code] --> CheckState{check_state}
        CheckState -- Reverse Hit --> StateTable[State table entry]
    end

    subgraph FTP_Helper [FTP Helper: helpers/ftp.c]
        StateTable --> Helper[ftp_support]
        Helper --> Parse[packet_contains_ftp_params]
        Parse --> Extract[Extract IP/Port from Payload]
    end

    subgraph Dynamic_State_Creation [State Management]
        Extract --> NewEntry[Create Dynamic state_table]
        NewEntry --> Flags[Set flag: FTP_DEFINED]
        Flags --> AddList[add_ftp_dynamic_rule]
    end

    AddList --> Accept[NF_ACCEPT]

    %% Styling
    classDef engine fill:#bbf,stroke:#333,stroke-width:2px;
    classDef helper fill:#fba,stroke:#333,stroke-width:2px;
    classDef dynamic fill:#dfd,stroke:#333,stroke-width:2px;

    class CheckState,StateTable engine;
    class Helper,Parse,Extract helper;
    class NewEntry,Flags,AddList dynamic;
```

### Key Operations

1. **FTP Flag Check**: Control connection state has `ftp = FTP_LOOK_FOR`
2. **Payload Inspection**: `ftp_support()` in `helpers/ftp.c` scans TCP payload
3. **227 Detection**: `data_start_with_227()` confirms "227" at start of data
4. **Parameter Extraction**:

```
// Parses: (203,0,113,100,195,210)
ftp_info {
    ftp_addr: 203.0.113.100 (in network order)
    ftp_port: 50130 (195*256 + 210, in network order)
    valid: 1
}
```

5. **Dynamic State Creation**:

```
struct state_table *newt = kmalloc(...)
newt->saddr = <local_ip>
newt->sport = 0              // ANY source port
newt->daddr = 203.0.113.100  // From FTP response
newt->dport = 50130          // From FTP response
newt->ftp = FTP_DEFINED      // Special FTP state
newt->state = IPFI_NOSTATE
```

6. **Special Matching**: When matching FTP_DEFINED states, source port is **ignored** in first packet

### Data Connection Flow

```
flowchart TD
    DataPkt[Data Packet: SYN to 50130] --> Hook_OUT{NF_IP_LOCAL_OUT}

    subgraph State_Match [State Stateful Match]
        Hook_OUT --> Lookup{check_state}
        Lookup -- Match ignoring sport --> Match[FTP_DEFINED Entry]
    end

    subgraph State_Upgrade [State Evolution]
        Match --> Upgrade[Update entry with actual sport]
        Upgrade --> Established[Set flag: FTP_ESTABLISHED]
    end

    Established --> Machine[state_machine: NEW -> SYN_SENT]
    Machine --> Accept[NF_ACCEPT]

    %% Styling
    classDef hook fill:#f9f,stroke:#333,stroke-width:2px;
    classDef engine fill:#bbf,stroke:#333,stroke-width:2px;
    classDef evolution fill:#dfd,stroke:#333,stroke-width:2px;

    class Hook_OUT hook;
    class Lookup,Match engine;
    class Upgrade,Established,Machine evolution;
```

### FTP State Transitions

```
Control: ESTABLISHED (ftp=FTP_LOOK_FOR)
```

```
         ↓ (227 response detected)
Dynamic: Created (ftp=FTP_DEFINED, sport=0)
         ↓ (First outgoing packet)
Data:    ESTABLISHED (ftp=FTP_ESTABLISHED, sport=<actual>)
```

---

## Rule 3: SSH Bidirectional Access

### Rule Definitions

```
RULE
NAME=me -> secure shell
DIRECTION=OUTPUT
MYSRCADDR
PROTOCOL=6
DSTPORT=22
KEEP_STATE=YES

RULE
NAME=secure shell -> me
DIRECTION=INPUT
MYDSTADDR
PROTOCOL=6
DSTPORT=22
KEEP_STATE=YES
```

### Scenario A: Outgoing SSH Connection

This follows the same stateful flow as Rule 1 (HTTP), but to `dport=22`.

### Scenario B: Incoming SSH Connection

User connects FROM `203.0.113.200` TO the firewall's SSH server at `192.0.2.100:22`.

```
flowchart TD
    Client((External Client)) --> Hook_PRE{NF_IP_PRE_ROUTING}

    subgraph Core [IPFire Filtering Core]
        Hook_PRE --> Routing[Routing: Local]
        Routing --> Hook_IN{NF_IP_LOCAL_IN}
        Hook_IN --> Response[ipfi_response]

        subgraph Logic [Rule Check]
            Response --> CheckState{check_state}
            CheckState -- Miss --> Filter[ipfire_filter]
            Filter -- "Match: 'ssh -> me'" --> Match[Match Found]
        end

        Match --> KeepState[keep_state]
```

```
    end

    subgraph State [State Creation]
        KeepState --> NewEntry[New state_table entry]
         NewEntry --> SYN_RECV[[State: SYN_RECV]]
    end

    NewEntry --> Accept[NF_ACCEPT] --> SSHD([SSH Daemon])

    %% Styling
    classDef hook fill:#f9f,stroke:#333,stroke-width:2px;
    classDef engine fill:#bbf,stroke:#333,stroke-width:2px;
    classDef state fill:#dfd,stroke:#333,stroke-width:2px;

    class Hook_PRE,Hook_IN hook;
    class Response,Logic,Filter engine;
    class NewEntry,SYN_RECV state;
```

#### Return Traffic (SYN-ACK from SSH daemon)

```
flowchart TD
    SSHD([SSH Daemon]) --> Hook_OUT{NF_IP_LOCAL_OUT}

    subgraph Stateful_Engine [IPFire Stateful Engine]
        Hook_OUT --> Lookup{check_state}
        Lookup -- "Hit (Reverse)" --> Entry[Existing State]
    end

    subgraph Transitions [State Machine]
        Entry --> Machine[state_machine: SYN_RECV -> ESTABLISHED]
    end

    Machine --> Accept[NF_ACCEPT] --> Client((External Client))

    %% Styling
    classDef hook fill:#f9f,stroke:#333,stroke-width:2px;
    classDef engine fill:#bbf,stroke:#333,stroke-width:2px;
    classDef state fill:#dfd,stroke:#333,stroke-width:2px;

    class Hook_OUT hook;
    class Lookup,Entry engine;
    class Machine state;
```

#### Bidirectional Flow Diagram

```
sequenceDiagram
    participant Client as External Client<br/>203.0.113.200
    participant FW_IN as Firewall<br/>INPUT Hook
```

```
    participant FW_OUT as Firewall<br/>OUTPUT Hook
    participant Daemon as SSH Daemon<br/>192.0.2.100:22

    Client->>FW_IN: SYN (dport=22)
    Note over FW_IN: Rule: "secure shell -> me"<br/>Creates state (direction=INPUT)
    FW_IN->>Daemon: SYN (ACCEPT)

    Daemon->>FW_OUT: SYN-ACK
    Note over FW_OUT: State: REVERSE match<br/>No rule check needed
    FW_OUT->>Client: SYN-ACK (ACCEPT)

    Client->>FW_IN: ACK + Data
    Note over FW_IN: State: Direct match<br/>(ESTABLISHED)
    FW_IN->>Daemon: Data packets

    Daemon->>FW_OUT: Data
    Note over FW_OUT: State: Reverse match<br/>(ESTABLISHED)
    FW_OUT->>Client: Data packets
```

---

## State Matching Logic

### Direct vs. Reverse Matching

The firewall uses sophisticated matching to handle bidirectional traffic:

#### Direct Match

```
// State table: saddr=A, daddr=B, sport=X, dport=Y, direction=OUTPUT
// Packet:      src=A,   dst=B,   sport=X, dport=Y, hook=LOCAL_OUT
// Result: MATCH (same direction, same addresses/ports)
```

#### Reverse Match

```
// State table: saddr=A, daddr=B, sport=X, dport=Y, direction=OUTPUT
// Packet:      src=B,   dst=A,   sport=Y, dport=X, hook=LOCAL_IN
// Result: MATCH (opposite direction, swapped addresses/ports)
```

### Hash Table Optimization

State lookups use bidirectional hash normalization:

```
u32 get_state_hash(__u32 saddr, __u32 daddr, __u16 sport, __u16 dport, __u8 proto)
{
    // Normalize: smaller address/port first
    if (saddr > daddr || (saddr == daddr && sport > dport)) {
        swap(saddr, daddr);
        swap(sport, dport);
    }
```

```c
    return jhash_3words(saddr, daddr, (sport << 16) | dport, proto);
}
```

This ensures both directions of a connection hash to the same bucket.

---

## Timer Management

### State Timeouts

| State | Timeout | Description |
|---|---|---|
| SYN_SENT | 120s | Setup phase |
| SYN_RECV | 120s | Setup phase |
| ESTABLISHED | 3600s | Active connection |
| FIN_WAIT | 120s | Shutdown phase |
| TIME_WAIT | 120s | Connection closing |

### Timer Optimization

Timers are only updated if >1 second has passed since last update:

```c
void update_timer_of_state_entry(struct state_table *sttable)
{
    unsigned long now = jiffies;
    if (time_after(now, sttable->last_timer_update + HZ)) {
        mod_timer(&sttable->timer_statelist,
                  jiffies + get_timeout_by_state(sttable->protocol, sttable->state) * HZ);
        sttable->last_timer_update = now;
    }
}
```

This reduces `mod_timer` overhead for high-throughput connections.

---

## Performance Characteristics

### Rule Evaluation Bypass

Once a state is established: - **State lookup**: O(1) hash table lookup - **Rule evaluation**: Skipped entirely - **Throughput impact**: Minimal (only hash computation + state machine update)

### Comparison

| Packet Type | State Lookup | Rule Evaluation | Verdict Source |
|---|---|---|---|
| New connection SYN | Miss | Full scan | Rule match |
| Return SYN-ACK | Hit (reverse) | Skipped | State |
| Established data | Hit (direct/reverse) | Skipped | State |
| Unrelated packet | Miss | Full scan | Default policy |

---

### Summary

These three rules demonstrate:

1. **Stateful HTTP**: Basic hash-based connection tracking eliminates rule re-evaluation
2. **FTP with Passive Mode**: Dynamic state creation allows data channels through firewall
3. **Bidirectional SSH**: Separate INPUT/OUTPUT rules with unified state tracking

The IPFire architecture achieves high performance through: - Hash table-based state lookups - Bidirectional connection normalization
- Timer optimization for high-throughput connections - Bypass of rule evaluation for established states

---

## TCP/UDP State Machine Analysis

### Overview

This document analyzes the IPFI kernel state machine implementation to verify correct connection tracking for TCP and UDP protocols, focusing on: - TCP three-way handshake tracking - UDP state transitions
- GUESS state handling (mid-flow connection tracking) - Permission rules on established flows

### TCP State Machine

#### Three-Way Handshake Tracking

The state machine in ipfi_state_machine.c correctly implements TCP three-way handshake tracking:

##### Normal Connection Establishment

```
// 1. Client sends SYN (lines 88-91)
if (syn && !ack)
    state = SYN_SENT;

// 2. Server responds with SYN/ACK (lines 110-112)
if ((current_state == SYN_SENT) && (syn == 1) && (ack == 1) && (reverse == 1))
    state = SYN_RECV;

// 3. Client sends ACK (lines 123-126)
if ((current_state == SYN_RECV) && (!rst) && (!fin) && (!syn) && (reverse == 0) && (ack))
    state = ESTABLISHED;
```

**Flow**: `IPFI_NOSTATE` → `SYN_SENT` → `SYN_RECV` → `ESTABLISHED`

**Fast Path Optimization (lines 73-77)** The most common case is optimized as the first check:

```
if (current_state == ESTABLISHED && ack && !syn && !rst && !fin)
    return ESTABLISHED;
```

This short-circuits state checking for already-established connections with ACK-only packets.

#### Connection Teardown

The machine tracks various FIN/RST scenarios:

```
FIN_WAIT (FIN seen) → CLOSE_WAIT (ACK after FIN) → LAST_ACK (FIN after FIN) → IPFI_TIME_WAIT
```

**RST handling**: Any RST packet transitions to `CLOSED` state (lines 144-145, 193-194).

#### Edge Cases

- **SYN retransmission** (line 116-117): Remains in `SYN_SENT`
- **Invalid combinations** (lines 188-191):
  - NULL flags (no SYN/ACK/RST/FIN) → `NULL_FLAGS`
  - Invalid combinations (SYN+RST, SYN+FIN, FIN+RST) → `INVALID_FLAGS`

### UDP State Machine

UDP is stateless but IPFI tracks "pseudo-states" for connection-like behavior:

```
// 1. First UDP packet seen (lines 32-33)
if (current_state == IPFI_NOSTATE)
    state = UDP_NEW;

// 2. Second packet in same flow (lines 35-36)
else if (current_state == UDP_NEW)
    state = UDP_ESTAB;

// 3. Subsequent packets (lines 38-39)
else if (current_state == UDP_ESTAB)
    state = UDP_ESTAB;

// 4. Unexpected state (lines 40-41)
else
    state = UDP_UNKNOWN;
```

**Flow**: `IPFI_NOSTATE` → `UDP_NEW` → `UDP_ESTAB` → `UDP_ESTAB` (persistent)

#### UDP_UNKNOWN Case

`UDP_UNKNOWN` occurs when `current_state` is neither `IPFI_NOSTATE`, `UDP_NEW`, nor `UDP_ESTAB`. This should theoretically never happen in correct operation, as UDP only transitions through these three states. It serves as a defensive fallback.

### GUESS States

#### Purpose

GUESS states handle **mid-flow connection tracking** - when the firewall starts seeing a connection that was already established before the firewall came up or rules were loaded.

#### Three GUESS States

**1. GUESS_SYN_RECV (lines 83-86)**

```
// SYN/ACK as FIRST packet seen
if (!rst && !fin && syn && ack)
    state = GUESS_SYN_RECV;
```

**Scenario**: Firewall sees SYN/ACK but missed the initial SYN.

**2. GUESS_ESTABLISHED (lines 93-96)**

```
// ACK-only packet with no SYN/RST/FIN as FIRST packet
if ((!rst) && (!fin) && (!syn))
    state = GUESS_ESTABLISHED;
```

**Scenario**: Firewall sees data packets from an already established connection.

**3. GUESS_CLOSING (lines 98-101, 153-161)**

```
// FIN+ACK as FIRST packet seen
if (!rst && fin && !syn && ack)
    state = GUESS_CLOSING;
```

**Scenario**: Firewall sees connection teardown but missed the established phase.

#### GUESS State Normalization

In set_state(), GUESS states are immediately normalized to regular states when stored:

```
if (state == GUESS_CLOSING)
    entry->state.state = CLOSED;
else if (state == GUESS_SYN_RECV)
    entry->state.state = SYN_RECV;
else if (state == GUESS_ESTABLISHED)
    entry->state.state = ESTABLISHED;
else
    entry->state.state = state;
```

**Result**: The state machine returns GUESS states, but they're stored as normal states. This allows: - Detection of mid-flow connections (return value) - Normal state progression going forward (stored value)

#### Security Implications

GUESS states allow the firewall to gracefully handle: 1. **Firewall reload** during active connections 2. **Rule updates** without breaking existing flows 3. **Dynamic insertion** into existing network topologies

However, this also means: **Mid-flow attacks** could potentially be accepted if they match a permission rule **Mitigation**: Permission rules should be carefully crafted, and stateful tracking prevents future packets from bypassing rules

### Permission Rules on Established Flows

#### Flow Processing Order

From ipfire_filter():

```
// 1. Check state table FIRST (lines 478-488)
if (flow->direction == IPFI_INPUT || flow->direction == IPFI_OUTPUT || flow->direction == IPFI_
    response = check_state(skb, flow);
    if (response.verdict > 0) {
        response.state = 1U;  // Mark as stateful
        return response;      // EARLY RETURN - skip rule matching
    }
}

// 2. If no state match, check rules (denial, then permission)
// 3. If permission rule matches AND state tracking enabled, create entry (lines 631-645)
```

#### Key Behavior: State Table Bypass

**Established flows bypass permission rules entirely**

Once a flow is in the state table: 1. `check_state()` finds the match (line 481) 2. Returns `IPFI_ACCEPT` immediately (line 486) 3. Permission rules are **never evaluated** (line 486 early return)

This is **correct behavior** because: - The permission rule was already checked when the flow was first created - Reduces CPU overhead for established connections - Prevents re-evaluation which could cause mid-flow policy changes

#### New Flow Permission Rule Application

For **new flows** (no state table match):

```
// Lines 631-637
if ((pass > 0) && ((rule->state) || (ipfi_opts->all_stateful)) && (ipfi_opts->state)) {
    if (flow->direction == IPFI_INPUT || flow->direction == IPFI_OUTPUT || flow->direction == I
        newtable = keep_state(skb, rule, flow);
        response.state = 1U;
    }
}
```

**Conditions for state tracking**: 1. `pass > 0`: Packet matched a permission rule 2. `rule->state || ipfi_opts->all_stateful`: State tracking enabled for rule or globally 3. `ipfi_opts->state`: Global stateful firewall enabled 4. Direction is INPUT, OUTPUT, or FORWARD (not PRE/POST)

**Comment on line 634**:

```
// *CHECK* Why? check_state above excluded an existing match for the current skb
```

This comment questions why `keep_state()` is called since `check_state()` already verified no match exists. The answer: **`keep_state()` creates new entries**, it doesn't just lookup. The comment reveals the developer recognized this might seem redundant but it's actually correct - `check_state()` looks up, `keep_state()` creates.

### Potential Issues & Recommendations

#### 1. GUESS_ESTABLISHED Security

**Issue**: Line 93-96 accepts ANY packet without SYN/RST/FIN as potentially established.

**Risk**: A crafted ACK packet could match a permission rule and be accepted even if connection doesn't exist.

**Mitigation**: Already in place via sequence number validation in TCP stack (outside IPFI).

#### 2. UDP State Lifetime

**Observation**: UDP connections remain in `UDP_ESTAB` indefinitely until timeout.

**Recommendation**: Verify timeout values in `get_timeout_by_state()` are appropriate for UDP (typically shorter than TCP ESTABLISHED).

#### 3. State Table Lookup Performance

**Current**: Lines 370-372 use hash table lookup with RCU read lock.

**Performance**: Hash-based lookup is O(1) average case, appropriate for high-traffic scenarios.

#### 4. Reverse Flag Tracking

**Observation**: Lines 372, 380, 382 carefully track `reverse` flag.

**Purpose**: Distinguishes original direction from reply direction, crucial for: - Asymmetric connection tracking - NAT support - Proper state transitions (e.g., SYN/ACK must have `reverse==1`)

### Summary

**Verified Correct:**

1. **TCP Three-Way Handshake**: SYN → SYN/ACK → ACK → ESTABLISHED
2. **UDP State Tracking**: NOSTATE → NEW → ESTAB

3. **GUESS States**: Properly handle mid-flow connections and normalize to regular states
4. **Permission Rules**: Correctly bypass established flows for performance
5. **State Machine Logic**: Comprehensive coverage of TCP states and edge cases

**Architecture Strengths:**

- Fast path optimization for `ESTABLISHED` state
- Hash-based state table lookup
- RCU locking for read scalability
- Early return for state table matches reduces overhead
- GUESS states provide operational flexibility

**Design Decisions:**

- State table checked **before** rule matching (performance)
- Permission rules applied **once** at flow creation (consistency)
- GUESS states normalized on storage (simplification)
- Reverse flag carefully tracked (correctness for bidirectional flows)

The state machine implementation is **robust and correct** for production use.

---

# NAT and Stateful Flow Analysis

This report details the packet flow and code logic for Source NAT (SNAT), Masquerade, and Destination NAT (DNAT) within the `ipfire-wall` kernel module, specifically focusing on their interaction with the stateful filtering engine.

## 1. Source NAT (SNAT) Flow

SNAT is typically applied to outgoing packets to replace the internal source address with a public one.

### Path: Internal Client → External Server

```
sequenceDiagram
    participant C as Internal Client (192.168.1.10)
    participant K as Kernel Hook: POST_ROUTING
    participant SN as SNAT Logic (snat.c)
    participant S as External Server (8.8.8.8)

    C->>K: Packet (Src: 192.168.1.10, Dst: 8.8.8.8)
    K->>SN: ipfi_post_process() -> snat_translation()
    SN->>SN: Match translation_post rule
    SN->>SN: add_snatted_entry() (Store Mapping)
    SN->>SN: do_source_nat() -> manip_skb()
    SN->>K: Packet Modified (Src: 1.2.3.4, Dst: 8.8.8.8)
    K->>S: Transmit over WAN
```

### Path: External Server → Internal Client (Return Path)

```
sequenceDiagram
    participant S as External Server (8.8.8.8)
```

```
    participant K as Kernel Hook: PRE_ROUTING
    participant SN as De-SNAT Logic (snat.c)
    participant C as Internal Client (192.168.1.10)

    S->>K: Packet (Src: 8.8.8.8, Dst: 1.2.3.4)
    K->>SN: ipfi_pre_process() -> pre_de_snat()
    SN->>SN: lookup_snatted_table() (Find Mapping)
    SN->>SN: de_snat() -> manip_skb()
    SN->>K: Packet Restored (Src: 8.8.8.8, Dst: 192.168.1.10)
    K->>C: Route to Internal Client
```

#### Key Functions

- `snat_translation()`: Dispatcher in `POST_ROUTING`.
- `add_snatted_entry()`: Creates the dynamic mapping for the connection.
- `manip_skb()`: Performs the actual IP/Port replacement and checksum updates (`csum_replace4`, `inet_proto_csum_replace4`).
- `pre_de_snat()`: Discovers and reverses the SNAT in the `PRE_ROUTING` hook for incoming responses.

---

### 2. Masquerade Flow

Masquerade is a specialized form of SNAT that dynamically fetches the IP of the outgoing interface.

#### Logical Differences from SNAT

1. **Dynamic IP**: Unlike `SNAT` which has a fixed `newaddr`, `Masquerade` calls `get_ifaddr(skb)`.
2. **Retrieval**: `get_ifaddr` utilizes `inet_select_addr(dev, dst, RT_SCOPE_UNIVERSE)` to find the most appropriate public IP for the current route.

#### Key Functions

- `masquerade_translation()`: Main handler in `masquerade.c`.
- `get_ifaddr()`: Correctly extracts the interface IP using `inet_select_addr`.
- `do_masquerade()`: Triggers `manip_skb` with the dynamic IP.

---

### 3. DNAT to FORWARD with Stateful Filtering

This scenario demonstrates how a packet's destination is changed, causing it to cross into the `FORWARD` chain where it activates stateful tracking.

#### The Flow: External Client → Internal Server

```
graph TD
    A[Packet In: C -> P] --> B{Hook: PRE_ROUTING}
    B --> C[ipfi_pre_process]
    C --> D[dnat_translation]
```

```
D --> E[Change Dst: C -> S1]
E --> F{Re-Route Decision}
F -->|Dst is not local| G{Hook: FORWARD}
G --> H[ipfi_response]
H --> I[ipfire_filter]
I --> J{State Lookup: check_state}
J -->|First Packet: No Match| K[Scan Allowed Rules]
K --> L{Rule Match: Stateful?}
L -->|Yes| M[keep_state]
M --> N[Create State Entry]
N --> O[Verdict: ACCEPT]
O --> P[Packet Out: C -> S1]
```

### Code Walkthrough Verification

1. **Rule Matching**: In `dnat_translation`, the packet is matched against the `translation_pre` list.
2. **NAT Accounting**: `add_dnatted_entry` is called BEFORE the translation, ensuring the original `old_saddr` and `old_daddr` are captured for later reversal.
3. **Re-routing**: In `ipfire.c:process`, we detected a `daddr` change and cleared the destination cache:

   ```c
   if (daddr != ip_hdr(skb)->daddr) {
       dst_release(skb_dst(skb));
       skb_dst_set(skb, NULL);
   }
   ```

   This is **CRITICAL**; without it, the kernel would try to deliver the packet locally to the non-existent `P` address instead of forwarding it to `S1`.
4. **Stateful Interaction**:
   - In the `FORWARD` chain, `ipfire_filter` calls `check_state`.
   - For the first packet, `check_state` misses.
   - The packet matches a rule in the `FORWARD` list.
   - If that rule has `.state = 1`, `keep_state()` is invoked.
   - `keep_state()` records the flow: `Src: C, Dst: S1, Proto: TCP, Ports: cport, sport`.
5. **Return Path Correctness**:
   - Return packet `S1 -> C` hits `POST_ROUTING`.
   - `de_dnat_translation()` matches `new_daddr` (Server) and restores `old_daddr` (Public IP) as the source.
   - The result `P -> C` is delivered to the client, which correctly recognizes the stream.

### Checksum Integrity

The `manip_skb` function correctly updates: - **IP Header Checksum**: via `csum_replace4`. - **L4 Checksum (TCP/UDP)**: via `inet_proto_csum_replace4` (address change) and `inet_proto_csum_replace2` (port change). - UDP `0` checksum is handled: `if (!pudphead->check) pudphead->check = CSUM_MANGLED_0;` (RFC 768 requirement).

> [!NOTE] All NAT operations are verified to be performed in a single shot per connection (static rules) through the dynamic lookup system, ensuring high-performance packet transformation.

---