Fast Firewalling Through System Specialization

Sven Hager*, Andreas Fießler**, Björn Scheuermann*

*Humboldt-Universität zu Berlin, Germany
**genua mbH, Germany

24. September 2015
Firewalling and packet classification are common tasks in packet processing systems. Performed by a wide variety of devices, including routers, packet filters, IPS, and more...
The packet classification (and firewalling) problem can be defined as follows:

- Let $\mathcal{P}$ be a tuple of header field values
The packet classification (and firewalling) problem can be defined as follows:

- Let $\mathcal{P}$ be a tuple of header field values
- Let $\mathcal{R}$ be an ordered prioritized list of rules $R_i$, e.g.
  \[
  \text{src\_addr} \in 34.5.12.100/8 \land \text{dst\_port} = 80 \Rightarrow \text{ACCEPT}
  \]
The packet classification (and firewalling) problem can be defined as follows:

- Let $\mathcal{P}$ be a tuple of header field values
- Let $\mathcal{R}$ be an ordered prioritized list of rules $R_i$, e.g.
  $$\text{src\_addr} \in 34.5.12.100/8 \land \text{dst\_port} = 80 \Rightarrow \text{ACCEPT}$$
- **Goal**: Find the highest prioritized rule $R^* \in \mathcal{R}$ that matches on $\mathcal{P}$
The packet classification (and firewallsing) problem can be defined as follows:

- Let $P$ be a tuple of header field values
- Let $R$ be an ordered prioritized list of rules $R_i$, e.g. $\text{src addr} \in 34.5.12.100/8 \land \text{dst port} = 80 \Rightarrow \text{ACCEPT}$
- **Goal:** Find the highest prioritized rule $R^* \in R$ that matches on $P$
- **Desired properties:**
  - As fast as possible
  - As cheap as possible
SECTION I: THE HARDFIRE PROJECT

▶ Cooperation project between genua mbH and HU Berlin
▶ **Goal:** Specialized matching circuitry for stateless/stateful firewalls [1, 2]
The hardFIRE Approach

- *Field Programmable Gate Arrays (FPGAs)* are programmable logic circuits
THE hardFIRE APPROACH

- *Field Programmable Gate Arrays (FPGAs)* are programmable logic circuits
- They can implement each “small enough” logic function “in hardware”

\[
f(x, y, z) := (x \land y) \lor z
\]
The hardFIRE Approach

- *Field Programmable Gate Arrays (FPGAs)* are programmable logic circuits
- They can implement each “small enough” logic function “in hardware”
- They can be reconfigured arbitrarily often

\[
g(x, y, z) := (x \lor y) \land (y \oplus z)
\]
The hardFIRE Approach

▶ A firewall ruleset $\mathcal{R}$ can be regarded as a logic function

$$\mathcal{F}_\mathcal{R} : \{\text{Header fields}\} \rightarrow \{\text{Firewall actions}\}$$

Input firewall ruleset

```
--prot TCP
--sport *
--dport 80
--src *
--dst 100.201.78.15
\text{\texttt{--> ACCEPT}}

--prot TCP
--sport 80
--dport *
--src 100.201.78.15
--dst *
\text{\texttt{--> ACCEPT}}
```
THE HARDFIRE APPROACH

- A firewall ruleset $\mathcal{R}$ can be regarded as a logic function
  $\mathcal{F}_\mathcal{R} : \{\text{Header fields}\} \rightarrow \{\text{Firewall actions}\}$

- **Basic idea:** Generate ruleset-tailored filtering circuit descriptions
THE HARDFIRE APPROACH

- A firewall ruleset $\mathcal{R}$ can be regarded as a logic function
  $$\mathcal{F}_\mathcal{R} : \{\text{Header fields}\} \rightarrow \{\text{Firewall actions}\}$$

- **Basic idea:** Generate ruleset-tailored filtering circuit descriptions

- These circuits are implemented by the FPGA
The hardFIRE Approach

- A firewall ruleset $\mathcal{R}$ can be regarded as a logic function
  \[ \mathcal{F}_\mathcal{R} : \{\text{Header fields}\} \rightarrow \{\text{Firewall actions}\} \]
- **Basic idea:** Generate ruleset-tailored filtering circuit descriptions
- These circuits are implemented by the FPGA
HARDFIRE Matching Circuitry

- The generated matching circuits consists of $N$ parallel match units
HARDFIRE Matching Circuity

- The generated matching circuits consists of $N$ parallel match units
- Match unit $i$ is individually crafted for rule $i
The generated matching circuits consists of $N$ parallel match units

Match unit $i$ is individually crafted for rule $i$
**HARDFIRE Matching Circuitry**

- The generated matching circuits consists of $N$ parallel match units
- Match unit $i$ is individually crafted for rule $i$
Rule Matching Circuit Optimization

- Duplicated checks between different rules are common, e.g.,
  - \( R_i : (\text{src addr} \in 20.3.50.1/24) \land (\text{prot} = \text{TCP}) \rightarrow \text{PASS} \)
  - \( R_k : (\text{src addr} \in 20.3.50.1/24) \land (\text{prot} = \text{UDP}) \rightarrow \text{DROP} \)
Rule Matching Circuit Optimization

- Duplicated checks between different rules are common, e.g.,
  - $R_i : (\text{src addr} \in 20.3.50.1/24) \land (\text{prot} = \text{TCP}) \rightarrow \text{PASS}$
  - $R_k : (\text{src addr} \in 20.3.50.1/24) \land (\text{prot} = \text{UDP}) \rightarrow \text{DROP}$
- These checks can be shared
IMPACT OF EQUAL CHECKS

- Check equality/similarity between rules can be exploited
**INTERIM RESULTS**

- The generated circuits achieve high throughput rates (40 Gbps and more)
INTERIM RESULTS

- The generated circuits achieve high throughput rates (40 Gbps and more)
- Latency is constant, no “worst case traffic”
Interim Results

- The generated circuits achieve high throughput rates (40 Gbps and more)
- Latency is constant, no “worst case traffic”
- Hardware is comparatively cheap
INTERIM RESULTS

- The generated circuits achieve high throughput rates (40 Gbps and more)
- Latency is constant, no “worst case traffic”
- Hardware is comparatively cheap
- **Challenge:** Each rule update requires a time-consuming circuit re-generation
IntroducTion

HARD-FIRE

JitVector

Conclusion

Interim Results

- The generated circuits achieve high throughput rates (40 Gbps and more)
- Latency is constant, no “worst case traffic”
- Hardware is comparatively cheap
- Challenge: Each rule update requires a time-consuming circuit re-generation

Approach:

- use the HARDFIRE-System as a coprocessor in a CPU-based host system
- handle (possibly complex) updates in software
SECTION II: JitVector

- Fast software-based packet classification on general-purpose CPUs [3]
**Why Software-based Filtering?**

▶ It’s cheap!
**Why Software-based Filtering?**

- It’s cheap!
- Packet filtering can be arbitrarily complex ⇒ hard to achieve in hardware
**Why Software-based Filtering?**

- It’s cheap!
- Packet filtering can be arbitrarily complex $\Rightarrow$ hard to achieve in hardware
- **However:** Naïve matching algorithms severely cut down performance
**Typical Software Approach**

The typical SW implementation of a packet filter looks like this:

```c
int find_index_of_first_matching_rule(Packet* p) {
    for (int i = 0; i < num_rules; ++i) {
        Rule* r = rule_list[i];
        if (((r->src_net_mask & p->src_addr) != r->src_net)
            continue;
        if (r->src_port != p->src_port)
            continue;
        ...  // further checks
        return i;
    }
    return -1;
}
```
Typical Software Approach

Problem: Many data memory accesses to read stored rule set

```c
int find_index_of_first_matching_rule(Packet* p) {
    for (int i = 0; i < num_rules; ++i) {
        Rule* r = rule_list[i];
        if ((r->src_net_mask & p->src_addr) != r->src_net)
            continue;
        if (r->src_port != p->src_port)
            continue;
        ... // further checks
        return i;
    }
    return -1;
}
```
TRANSLATING RULES TO MACHINE CODE

Idea: Replace the generic matching loop with ruleset-specific machine instructions

```c
int find_index_of_first_matching_rule(Packet* p) {
    //
    //
    //
    //
    // * Ruleset-specific comparison machine instructions *
    //
    //
    //
    //
}
```
TRANSLATING RULES TO MACHINE CODE

Rule 0: dst_port = 80 ⇒ DROP
Rule 1: dst_port = 443 ⇒ ACCEPT

```c
int find_index_of_first_matching_rule(Packet* p) {
    mov RESULT_REG, $-1;
    rule_0: cmp DST_PORT_REG, $80 ;
            jne rule_1 ;
            mov RESULT_REG, $0 ;
            jmp done ;
    rule_1: cmp DST_PORT_REG, $443 ;
            jne done ;
            mov RESULT_REG, $1 ;
    done:   return RESULT_REG ;
}
```
TRANSLATING RULES TO MACHINE CODE

Rule 0: $\text{dst}\_\text{port} = 80 \Rightarrow \text{DROP}$
Rule 1: $\text{dst}\_\text{port} = 443 \Rightarrow \text{ACCEPT}$

```c
int find_index_of_first_matching_rule(Packet* p) {
    mov RESULT_REG, $-1;
    rule_0: cmp DST_PORT_REG, $80;
            jne rule_1;
            mov RESULT_REG, $0;
            jmp done;
    rule_1: cmp DST_PORT_REG, $443;
            jne done;
            mov RESULT_REG, $1;
    done:   return RESULT_REG;
}
```
MOVING TO JitVector

- **Issue:** The generated code can become very large as the rule set size grows
- ⇒ Bad classification performance
Moving to JitVector

- **Issue:** The generated code can become very large as the rule set size grows
- ⇒ Bad classification performance
- **Idea:** We can use another classification algorithm whose search data structure allows for generation of better code
MOVING TO JitVector

▶ **Issue:** The generated code can become very large as the rule set size grows
▶ ⇒ **Bad classification performance**
▶ **Idea:** We can use another classification algorithm whose search data structure allows for generation of better code
▶ **We chose the bit vector algorithm**
SEARCH DATA STRUCTURES

- The data structure separation is possible due to the decompositional nature of the bit vector algorithm
SEARCH DATA STRUCTURES

- The data structure separation is possible due to the decompositional nature of the bit vector algorithm
- The rule set is transformed into sequences of intervals for each search dimension

E.g. source address dimension

\( \{A - B, C - D, E - F, G - H, K - L\} \)

E.g. destination address dimension

\( \{A - B, C - D, E - F, G - H, K - L\} \)
Search Data Structures

- During packet classification, each sequence is binary searched

E.g. source address dimension

Interval sequences

\{\begin{array}{c}
A - B \\
C - D \\
E - F \\
G - H \\
K - L
\end{array}\}

E.g. destination address dimension

\{\begin{array}{c}
A - B \\
C - D \\
E - F \\
G - H \\
K - L
\end{array}\}

...
**Search Data Structures**

- The code generator linearizes the binary search tree implied by the intervals
- Interval bounds are encoded as direct operands in the instruction stream ⇒ no data memory accesses
**MEASUREMENT RESULTS**

- We evaluated JitVector vs. bit vector with synthetic rule sets and traces in an isolated environment.
- We observe faster classification times and longer preprocessing times for JitVector.
### Measurement Results

- As a proof of concept, we also integrated JitVector in the OpenFlow Reference Switch (ORS, v. 1.0)
- Throughput comparison shows significant improvement in matching performance (vs. baseline linear search)

![Graph showing throughput increase with number of rules](image)
CONCLUSION

- Specialization of classification engines for firewalls or other network devices can improve efficiency
CONCLUSION

- Specialization of classification engines for firewalls or other network devices can improve efficiency
- In hardware: Ruleset-optimized parallel matching circuitry
CONCLUSION

▶ Specialization of classification engines for firewalls or other network devices can improve efficiency
▶ In hardware: Ruleset-optimized parallel matching circuitry
▶ In software: Better matching performance due to partially evaluated search functions
CONCLUSION

- Specialization of classification engines for firewalls or other network devices can improve efficiency
- In hardware: Ruleset-optimized parallel matching circuitry
- In software: Better matching performance due to partially evaluated search functions

Thank you for your attention!

References:
Resource usage correlates linearly with the number of rules
Worst case: each check in each rule is unique
**Industry Standard: Parallel Search in TCAMs**

- Fully parallel search in TCAMs (Ternary Content-Addressable Memories)
- Each rule is stored in a match row of $w$ bit
Industry Standard: Parallel Search in TCAMs

- Fully parallel search in TCAMs (Ternary Content-Addressable Memories)
- Each rule is stored in a match row of $w$ bit

**Problem:** Space is wasted if rules need less than $w$ bit

**Example:** $w = 104$, rule checks only a /24 subnet
MEASUREMENT RESULTS

- Preprocessing times for the JitVector and bit vector approaches