NFReducer: Redundant Logic Elimination for Network Functions with Runtime Configurations

Bangwen Deng, Wenfei Wu
Tsinghua University
Background: Network Function Virtualization (NFV)

Proprietary software and hardware

NFV

General software deployed in commodity servers or virtual machines

*Figure from ETSI*
Background: Network Function Virtualization (NFV)

Proprietary software and hardware

General software deployed in commodity servers or virtual machines

*Figure from ETSI*
Background: Network Functions

- Critical components in modern network

- Growing impact:
  - Various network scenarios
  - Diverse functions (e.g., Firewall, NAT, IDS, Load Balancer)
Background: Network Functions

• Critical components in modern network

• Growing impact:
  • Various network scenarios
  • Diverse functions (e.g., Firewall, NAT, IDS, Load Balancer)

• NF’s efficiency in flow processing is critical:
  • Affects network’s end-to-end performance in a significant way
    (e.g., latency accumulation, throughput bottleneck)

• DevOps concept introduces more NF optimization space.
Outline

• *Background*

• Motivation and Objective

• NFReducer Design and Implementation

• Evaluation

• Conclusion and Future work
Motivation: Three Types of Redundancy in NF

• **Unused Logic:** mismatch of the protocol space in development and **deployment**.
  • Covering a large protocol space in development
  • Configuring a subspace of the entire protocol space in deployment
Motivation: Three Types of Redundancy in NF

- **Unused Logic**: mismatch of the protocol space in development and deployment.
  - Covering a large protocol space in development
  - Configuring a subspace of the entire protocol space in deployment
Motivation: Three Types of Redundancy in NF

- **Unused Logic**: mismatch of the protocol space in development and deployment.
- **Duplicated Logic**: duplicated operations in NFs of an NF chain.

Monitor and IDS in a chain share similar packet parsing logic.
Motivation: Three Types of Redundancy in NF

- Unused Logic: mismatch of the protocol space in development and deployment.
- Duplicated Logic: duplicated operations in NFs of an NF chain.
- Overwritten Logic: overwritten actions between NFs in an NF chain.

![Diagram of NFs flow with Monitor and IDS]

- Ingress flows → Monitor → Egress flows
- UDP packets processing is redundant
- Block UDP packets
- IDS

Objective

- Unused Logic: mismatch of the protocol space in development and deployment.
- Duplicated Logic: duplicated operations in NFs of an NF chain.
- Overwritten Logic: overwritten actions between NFs in an NF chain.

**Goal:** Identify and eliminate redundant logic in NFs and NF chains with the operation-time configurations.
Snort IDS Code (Simplified)

```c
/* One example Snort rule: drop tcp 10.0.0.0/24 any -> 10.1.0.0/24 any */
struct {
    unsigned long sip, dip;
    unsigned short sport, dport;
    ...
} net;

void main() {
    LoadRules();
    while(1) {
        pkt = ... // get a packet
        DecodeEthPkt(pkt); // decode a packet
        ApplyRules(); // match rules
    }
}

void DecodeEthPkt(u_char *pkt) {
    DecodeIPPkt(pkt);
}

void DecodeIPPkt(u_char *pkt) {
    net.dip = ...;
    net.sip = ...
    net.protocol = ...
    log(net.sip, net.dip, net.protocol);
    if (net.protocol == TCP)
        DecodeTCPPkt(pkt);
    else if (net.protocol == UDP)
        DecodeUDPPkt(pkt);
    else if (...) {...}
}

void DecodeTCPPkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}

void DecodeUDPPkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}

void ApplyRules() {
    while(...) { // iterate each rule r
        if (MatchRule(r)) {
            Action();
            return;
        }
    }
}

int MatchRule(Rule *r){
    if (r->sip != net.sip) return 0;
    if (r->dip != net.dip) return 0;
    if (r->protocol != net.protocol) return 0;
    if (r->sport != net.sport) return 0;
    if (r->dport != net.dport) return 0;
    return 1;
}
```
Snort IDS Code (Simplified)

```c
/* One example Snort rule:
 drop tcp 10.0.0.0/24 any -> 10.1.0.0/24 any */
struct {
    unsigned long sip, dip;
    unsigned short sport, dport;
    ...
} net;
void main() {
    LoadRules();
    while(1) {
        pkt = ... // get a packet
        DecodeEthPkt(pkt); // decode a packet
        ApplyRules(); // match rules
    }
}
void DecodeEthPkt(u_char *pkt) {
    DecodeIPPkt(pkt);
}
void DecodeIPPkt(u_char *pkt) {
    net.dip = ...;
    net.sip = ...;
    net.protocol = ...
    log(net.sip, net.dip, net.protocol);
    if (net.protocol == TCP)
        DecodeTCPPkt(pkt);
    else if (net.protocol == UDP)
        DecodeUDPPkt(pkt);
    else if (...) { ... }
}
void DecodeTCPPkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}
void DecodeUDPPkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}
void ApplyRules() {
    while(...) { // iterate each rule r
        if (MatchRule(r)) {
            Action();
            return;
        }
    }
    MatchRule(Rule *r){
        if (r->sip != net.sip) return 0;
        if (r->dip != net.dip) return 0;
        if (r->protocol != net.protocol) return 0;
        if (r->sport != net.sport) return 0;
        if (r->dport != net.dport) return 0;
        return 1;
    }
}
```

Parsing
Snort IDS Code (Simplified)

```c
/* One example Snort rule: */
drop tcp 10.0.0.0/24 any -> 10.1.0.0/24 any
*/

struct {
    unsigned long sip, dip;
    unsigned short sport, dport;
    ...
} net;

void main() {
    LoadRules();
    while(1) {
        pkt = ... // get a packet
        DecodeEthPkt(pkt); // decode a packet
        ApplyRules(); // match rules
    }
}

void DecodeEthPkt(u_char *pkt) {
    DecodeIPPkt(pkt);
}

void DecodeIPPkt(u_char *pkt) {
    net.dip = ... 
    net.sip = ...
    net.protocol = ...
    log(net.sip, net.dip, net.protocol);
    if (net.protocol == TCP)
        DecodeTCPpkt(pkt);
    else if (net.protocol == UDP)
        DecodeUDPPkt(pkt);
    else if (...) { ... }
}

void DecodeTCPpkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}

void DecodeUDPPkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}

void ApplyRules() {
    while(...) { // iterate each rule r
        if (MatchRule(r)) {
            Action();
            return;
        }
    }
}

int MatchRule(Rule *r) {
    if (r->sip != net.sip) return 0;
    if (r->dip != net.dip) return 0;
    if (r->protocol != net.protocol) return 0;
    if (r->sport != net.sport) return 0;
    if (r->dport != net.dport) return 0;
    return 1;
}
```
Snort IDS Code(Simplified)

```c
/* One example Snort rule: */
drop tcp 10.0.0.0/24 any -> 10.1.0.0/24 any
*/
struct {
    unsigned long sip, dip;
    unsigned short sport, dport;
    ...
} net;

void main() {
    LoadRules();
    while(1) {
        pkt = ... // get a packet
        DecodeEthPkt(pkt); // decode a packet
        ApplyRules(); // match rules
    }
}

void DecodeEthPkt(u_char *pkt) {
    DecodeIPPkt(pkt);
}

void DecodeIPPkt(u_char *pkt) {
    net.dip = ...
    net.sip = ...
    net.protocol = ...
    log(net.sip, net.dip, net.protocol);
    if (net.protocol == TCP)
        DecodeTCPPkt(pkt);
    else if (net.protocol == UDP)
        DecodeUDPPkt(pkt);
    else if (...) { ... }
}

void DecodeTCPPkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}

void DecodeUDPPkt(u_char *pkt) {
    net.dport = ...
    net.sport = ...
    log(net.sport, net.dport);
}

void ApplyRules() {
    while(...) { // iterate each rule r
        if(MatchRule(r)) {
            Action();
            return;
        }
    }
}

int MatchRule(Rule *r)
{
    if(r->sip != net.sip) return 0;
    if(r->dip != net.dip) return 0;
    if(r->protocol != net.protocol) return 0;
    if(r->sport != net.sport) return 0;
    if(r->dport != net.dport) return 0;
    return 1;
}
```

### Parsing ➔ Match ➔ Action
Unused Logic: Unused layer parsing

• Example

<table>
<thead>
<tr>
<th>Parsing</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP address (L3)</td>
<td>Pkt.IP == Rule.IP</td>
<td>Drop</td>
</tr>
<tr>
<td>Port (L4)</td>
<td>Pkt.Port == Rule.Port</td>
<td>Pass</td>
</tr>
</tbody>
</table>
Unused Logic: Unused layer parsing

• Example

What if only L3 header is used? E.g., <10.0.0.1-*>, s/d port=*, drop>

<table>
<thead>
<tr>
<th>Parsing</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP address (L3)</td>
<td>Pkt.IP == Rule.IP</td>
<td>Drop</td>
</tr>
<tr>
<td>Port (L4)</td>
<td>Pkt.Port == Rule.Port</td>
<td>Pass</td>
</tr>
</tbody>
</table>
Unused Logic: Unused layer parsing

• Example

What if only L3 header is used? E.g., <10.0.0.1->* s/d port=*, drop>

Parsing
- IP address (L3)
- Port (L4)

Match
- Pkt.IP == Rule.IP
- Pkt.Port == Rule.Port

Action
- Drop
- Pass

Wildcard
Always True
Unused Logic: Unused layer parsing

• Example

What if only L3 header is used? E.g., <10.0.0.1-> *, s/d port=*, drop>

Parsing:
- IP address (L3)
- Port (L4)

Match:
- Pkt.IP == Rule.IP
- Pkt.Port == Rule.Port

Action:
- Drop
- Pass

Unused

Wildcard

Always True
Unused layer parsing: Method to Solve

- Apply Rules

\[ <10.0.0.1->*, s/d \text{ port}=*, \text{ drop}> \]
Unused layer parsing: Method to Solve

- Apply Rules
- Constant Folding and Propagation

<10.0.0.1-*>*, s/d port=*, drop>

Parsing
- IP address (L3)
- Port (L4)

Match
- Pkt.IP == Rule.IP
  - True

Action
- Drop
- Pass
Unused layer parsing: Method to Solve

- Apply Rules
- Constant Folding and Propagation
- Dead Code Elimination

\[
\langle 10.0.0.1->, s/d port=*, \text{drop} >
\]

**Diagram:**
- **Parsing**
  - IP address (L3)
  - Port (L4)

- **Match**
  - Pkt.IP == Rule.IP
  - True

- **Action**
  - Drop
  - Pass
Unused Logic: Unused Protocol (Branch) Parsing

• Branches in Parse and Match

If NF processes TCP packets only, E.g., <10.0.0.0/24, tcp, 80, drop>

```
Parsing

IP Parsing
  Proto==TCP
    TCP Parsing
  Proto==UDP
    UDP Parsing

Match

IP
  Proto==TCP
    Port==80
      drop
    Port!=80
      pass
  Proto==UDP
    Port==*
      pass
```
Unused Logic: Unused Protocol (Branch) Parsing

• Branches in Parse and Match

If NF processes TCP packets only, E.g., <10.0.0.0/24, tcp, 80, drop>

Parsing

IP Parsing
- Proto==TCP
  - TCP Parsing
- Proto==UDP
  - UDP Parsing

Match

IP
- Proto==TCP
  - Port==80
    - drop
  - Port!=80
    - pass
- Proto==UDP
  - Port==*
    - pass
  - Always False
  - True
Unused Logic: Unused Protocol (Branch) Parsing

- Branches in Parse and Match

If NF processes TCP packets only, E.g., <10.0.0.0/24, tcp, 80, drop>

Redundant Logic

True

Always False
Unused Protocol Parsing: Method to Solve

- Extract Feasible Execution Path
Unused Protocol Parsing: Method to Solve

- Extract Feasible Execution Path
- Constant Folding and Propagation
Unused Protocol Parsing: Method to Solve

- Extract Feasible Execution Path
- Constant Folding and Propagation
- Dead Code Elimination
Unused Protocol Parsing: Method to Solve

- Extract Feasible Execution Path
- Constant Folding and Propagation
- Dead Code Elimination
Cross-NF Redundancy: Duplicated Logic

- If monitor and IDS in a chain share similar packet parsing logic.
Cross-NF Redundancy: Duplicated Logic

- If monitor and IDS in a chain share similar packet parsing logic.

The double packet “parsing” is duplicated redundant logic.
Duplicated Logic: Method to Solve

• The double packet “parsing” is duplicated redundant logic.
• Consolidate to solve:
  • Splice the code
**Duplicated Logic: Method to Solve**

- The double packet “parsing” is duplicated redundant logic.
- Consolidate to solve:
  - Splice the code
  - Common subexpression elimination & copy propagation
  - Dead code elimination
Cross-NF Redundancy: Overwritten Logic

- Two firewall instances are chained together in the setting that two operators manage their rules independently or with priority.
Cross-NF Redundancy: Overwritten Logic

- If firewall instance 1 lets all UDP packets get through, but firewall instance 2 blocks all UDP packets, all work in firewall 1 that is done to UDP becomes redundant.
Cross-NF Redundancy: Overwritten Logic

- If firewall instance 1 lets all UDP packets get through, but firewall instance 2 blocks all UDP packets, all work in firewall 1 that is done to UDP becomes redundant.

The overlapping conflict actions for UDP packets is overwritten redundant logic.
Overwritten Logic: Method to Solve

• The overlapping conflict actions for specific packets is overwritten redundant logic.
• To solve:
  • Consolidate
**Overwritten Logic: Method to Solve**

- The overlapping conflict actions for specific packets is overwritten redundant logic.
- To solve:
  - Consolidate
  - Check and label chain actions
**Overwritten Logic: Method to Solve**

- The overlapping conflict actions for specific packets is overwritten redundant logic.
- To solve:
  - Consolidate
  - Check and label chain actions
  - Dead Code Elimination
Outline

• Background
• Motivation and Objective
• NFReducer Design and Implementation
• Evaluation
• Conclusion and Future work
NFReducer Workflow

- Input:
  - NF program source code
  - Runtime configurations
  - NF chain information

- Output: The optimized NF program
NFReducer Workflow

- Labeling Critical Variables and Actions
**NFReducer Workflow**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
NFReducer Workflow

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Eliminating Redundant Logic
NFReducer Workflow

• Labeling Critical Variables and Actions

  • Critical Variables
    • Packet Variables: Holding the packet raw data.
    • State Variables: Maintaining the NF states. (e.g., counter)
    • Config Variables: Maintaining the config info. (e.g., rules)

  • NF Actions:
    • External Actions (e.g., replying, forward, drop packets)
    • Internal Actions (e.g., updating state variables)
NFReducer Workflow

• Labeling Critical Variables and Actions

• Extracting Packet Processing Logic
  • Removing functionalities unrelated to packet processing (e.g., log).
  • Facilitate the compiler techniques applied later (e.g., symbolic execution).
**NFReducer Workflow**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Eliminating Redundant Logic
  - Unused Logic Elimination
    - Apply Configs
    - Extract Paths

```
Packet Processing Logic
---
Configured Rules
---
Apply Configs & Extract Paths
---
Path1
Path2
...

Constant Folding & Propagation
---
Check path feasibility
---
Dead Code Elimination & Merge
---
Optimized Code
```
**NFReducer Workflow**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Eliminating Redundant Logic
  - Unused Logic Elimination
    - Apply Configs
    - Extract Paths
    - Constant Folding and Propagation
    - Check Path Feasibility
    - Dead Code Elimination

![NFReducer Workflow Diagram](image-url)
NFReducer Workflow

• Labeling Critical Variables and Actions
• Extracting Packet Processing Logic
• Eliminating Redundant Logic
  • Unused Logic Elimination
  • Duplicated Logic Elimination
  • Splice

- NF1
- NF2
  - Splice
  - Common subexpression elimination & Copy propagation
  - Dead Code Elimination
  - Consolidated Code
**NFReducer Workflow**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Eliminating Redundant Logic
  - Unused Logic Elimination
  - Duplicated Logic Elimination
    - Splice
    - Common subexpression elimination & copy propagation
    - Dead code elimination
**NFReducer Workflow**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Eliminating Redundant Logic
  - Unused Logic Elimination
  - Duplicated Logic Elimination
  - Overwritten Logic Elimination
  - Consolidate

Consolidate

Check and label chain actions

Dead Code Elimination

Optimized NF
**NFReducer Workflow**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Eliminating Redundant Logic
  - Unused Logic Elimination
  - Duplicated Logic Elimination
  - Overwritten Logic Elimination
  - Consolidate
- Check and label chain actions
- Dead Code Elimination
- Consolidate
- Check and label chain actions
- Dead Code Elimination
- Optimized NF
Implementation

Source Code

Label Variables and Actions
  - Actions
  - Variables

Extract Packet Processing Logic

Configured Rules

Eliminate Redundant Logic
  - IntraNFOpt()
  - Consolidate()
  - CrossNFOpt()

NF Chain Info

Optimized NF Program

LLVM DG Static Slicer
Outline

• Background
• Motivation and Objective
• NFReducer Design and Implementation
• Evaluation
• Conclusion and Future work
Evaluation: Experimental Setup

• Benchmarks
  • Snort IDS
  • Suricata IDS/IPS
  • Firewall in OpenNetVM platform

• Evaluation Indicators
  • End-to-end performance gain:
    • Throughput (packet per second)
  • Overhead
    • Time for program analysis and optimization
Evaluation: Eliminating unused layer logic

- Setting: Configured with layer-3 rules.

- Increase by nearly 15% for Snort, by 15% to 10X for Suricata (single thread), and by 21% for OpenNetVM-Firewall.

- Suricata is more significant
  - inspects packets deeper in payload than Snort and OpenNetVM-FW.
Evaluation: Eliminating unused layer logic

- Setting: Configured with layer-3 rules.

- Increase by nearly 15% for Snort, by 15% to 10X for Suricata (single thread), and by 21% for OpenNetVM-Firewall

- Suricata is more significant
  - inspects packets deeper in payload than Snort and OpenNetVM-FW.
Evaluation: Eliminating unused layer logic

Throughput of Snort

Throughput of Suricata

Throughput of OpenNetVM-FW

- Setting: Configured with layer-3 rules.
- Increase by nearly 15% for Snort, by 15% to 10X for Suricata (single thread), and by 21% for OpenNetVM-Firewall
- Suricata is more significant
  - inspects packets deeper in payload than Snort and OpenNetVM-FW.
Evaluation: Eliminating unused layer logic

- Setting: Configured with layer-3 rules.

- Increase by nearly 15% for Snort, by 15% to 10X for Suricata (single thread), and by 21% for OpenNetVM-Firewall

- Suricata is more significant
  - inspects packets deeper in payload than Snort and OpenNetVM-FW.
Evaluation: Eliminating unused layer logic

- Setting: Configured with layer-3 rules.

- Increase by nearly 15% for Snort, by 15% to 10X for Suricata (single thread), and by 21% for OpenNetVM-Firewall.

- Suricata is more significant:
  - inspects packets deeper in payload than Snort and OpenNetVM-FW.
Evaluation: Eliminating unused protocol logic

- Setting: Configured with TCP rules only.
- The larger proportion of UDP packets, the larger performance gain.
- 40% performance gain for Snort, $2.5 \times$ for Suricata, and 6.8% for OpenNetVM Firewall
**Evaluation: Eliminating unused protocol logic**

- Setting: Configured with TCP rules only.

- The larger proportion of UDP packets, the larger performance gain.

- 40% performance gain for Snort, $2.5 \times$ for Suricata, and 6.8% for OpenNetVM Firewall
Evaluation: Eliminating Duplicated and Overwritten Logic

- Setting:
  - Mon—Snort: executed in two processes
  - Mon+Snort: directly spliced
  - Mon+Snort-Con: consolidated
  - Mon+Snort-Opt: consolidated and optimized
  - Configured with TCP rules only for Snort
Evaluation: Eliminating Duplicated and Overwritten Logic

- Setting:
  - Mon—Snort: executed in two processes
  - Mon+Snort: directly spliced
  - Mon+Snort-Con: consolidated
  - Mon+Snort-Opt: consolidated and optimized
  - Configured with TCP rules only for Snort

- Consolidation and Redundancy Elimination help improve:
  - By more than 30%

- Performance gain increases as the UDP proportion increases.
Evaluation: Eliminating Duplicated and Overwritten Logic

- Setting:
  - Fw—Fw: executed in two processes
  - Fw + Fw: directly spliced
  - Fw + Fw-Con: consolidated
  - Fw + Fw-Opt: consolidated and optimized
  - Configured with TCP rules only for latter Firewall instance
Evaluation: Eliminating Duplicated and Overwritten Logic

- Setting:
  - Fw—Fw: executed in two processes
  - Fw + Fw: directly spliced
  - Fw + Fw-Con: consolidated
  - Fw + Fw-Opt: consolidated and optimized
  - Configured with TCP rules only for latter Firewall instance

- Consolidation and Redundancy Elimination help improve:
  - By more than 60%

- Performance gain increases as the UDP proportion increases.
Evaluation: Overhead

- Labeling Variables and Actions:
  - Operator-involved
  - Once for an NF

# of Identified Critical Variables in Benchmarks

<table>
<thead>
<tr>
<th></th>
<th># of Packet Variables</th>
<th># of State Variables</th>
<th># of Config Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snort IDS</td>
<td>2</td>
<td>8 (4 FPs)</td>
<td>6</td>
</tr>
<tr>
<td>Suricata IDS</td>
<td>2</td>
<td>5 (1 FP)</td>
<td>10</td>
</tr>
<tr>
<td>OpenNetVM-Firewall</td>
<td>2</td>
<td>4</td>
<td>22</td>
</tr>
</tbody>
</table>
**Evaluation: Overhead**

- Labeling Variables and Actions
- Optimization **Overhead**

<table>
<thead>
<tr>
<th></th>
<th>Extracting packet processing logic</th>
<th>Optimization</th>
<th>Rebuilding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snort IDS</td>
<td>7.6s</td>
<td>26.8s</td>
<td>0.126s</td>
</tr>
<tr>
<td>Suricata IDS</td>
<td>1.2s</td>
<td>83.6s</td>
<td>2.753s</td>
</tr>
<tr>
<td>OpenNetVM-Firewall</td>
<td>0.146s</td>
<td>1.606s</td>
<td>1.571s</td>
</tr>
</tbody>
</table>
Outline

• Background
• Motivation and Objective
• NFReducer Design and Implementation
• Evaluation
• Conclusion
Conclusion

- We show the existence of three types of redundant logic in NFs.
- We design tool named NFReducer to eliminate the redundant logic.
  - Using runtime configurations
  - Applying program analysis methods and compiler techniques.
- We evaluate NFReducer on commodity NFs and platform NFs
  - The performance gain is obvious
  - The overhead is acceptable
Thanks and Q&A!

dbw18@mails.tsinghua.edu.cn
Scope of Usage

• NFs that process a larger protocol space could be benefited significantly.
  • e.g., IDSes, firewalls, and Deep Packet Inspectors (DPI)
• NFs that process a single protocol could benefit less
  • E.g., TCP load balancer, HTTP cache
• In DevOps scenarios
  • Each NF category contains many different NF variants
  • Many NFs synthesize several functionalities
  • One NF would be deployed as many instances