Redundant Logic Elimination in Network Functions

Bangwen Deng¹, Wenfei Wu¹, Linhai Song²
1: Tsinghua University 2: The Pennsylvania State University
Network Functions: Critical components in 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)
Network Functions: Critical components in network

• Mismatch of the protocol space in the development and that in the deployment leads to redundant logic:
  • Covering a large protocol space in development
  • Configuring a subspace of the entire protocol space in deployment
Network Functions: Critical components in network

- Mismatch of the protocol space in the development and that in the deployment leads to redundant logic:
  - Covering a large protocol space in development
  - Configuring a subspace of the entire protocol space in deployment
Network Functions: Critical components in network

• Mismatch of the protocol space in the development and that in the deployment leads to redundant logic:
  • Covering a large protocol space in development
  • Configuring a subspace of the entire protocol space in deployment

**Goal:**
To use compiler techniques to optimize away the redundancy.
Outline

• Introduction
• Design Intuition
• NFReducer Implementation
• Preliminary Evaluation
• Conclusion
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 = ...
    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(...) {
      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;
}
```
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 rule
    }
    DecodeEthPkt(u_char *pkt) {
        DecodeIPPkt(pkt);
    }
    void DecodeIPPkt(u_char *pkt) {
        net.dip = ...
        net.sip = ...
        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
Type-I Redundancy: Unused layer parsing

- Example

<table>
<thead>
<tr>
<th>Parsing</th>
<th>Match</th>
<th>Action</th>
</tr>
</thead>
</table>
| IP address (L3) Port (L4) | Pkt.IP == Rule.IP  
Pkt.Port == Rule.Port | Drop   
Pass         |
Type-I Redundancy: 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>
Type-I Redundancy: Unused layer parsing

• Example

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

Parsing

Unused

IP address (L3)

Port (L4)

Match

Wildcard

Pkt.IP == Rule.IP

Pkt.Port == Rule.Port

Action

Drop

Pass

Always True
Type-I Redundancy: Method to Solve

- Apply Rules

\[
\begin{align*}
\text{Parsing:} & \quad \text{IP address (L3)} \\
\text{Port (L4)} & \\
\text{Match:} & \quad \text{Pkt.IP == Rule.IP} \\
\text{Pkt.Port == *} & \\
\text{Action:} & \quad \text{Drop} \\
\text{Pass} & \\
\end{align*}
\]

\[<10.0.0.1->*, s/d\ port=*, drop>\]
Type-I Redundancy: Method to Solve

- Apply Rules
- Constant Folding and Propagation

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

- Parsing
  - IP address (L3)
  - Port (L4)
- Match
  - \( \text{Pkt.IP} == \text{Rule.IP} \)
    - True
- Action
  - Drop
  - Pass
Type-I Redundancy: Method to Solve

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

\[
<10.0.0.1-*; s/d port=*; drop>
\]

- Parsing
  - IP address (L3)
  - Port (L4)
- Match
  - Pkt.IP == Rule.IP
    - True
- Action
  - Drop
  - Pass
Type-II Redundancy: 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

Match
Type-II Redundancy: Unused Protocol (Branch) Parsing

• Branches in Parse and Match

If NF processes TCP packets only, E.g., \(<10.0.0.0/24, \text{tcp}, 80, \text{drop}>\)

Redundant Logic

TCP Parsing  UDP Parsing

Proto==TCP  Proto==UDP

TCP Parsing  UDP Parsing

Proto==TCP  Proto==UDP

Port==80  Port!=80  Port==*

drop  pass  pass
Type-II Redundancy: Method to Solve

- Extract Feasible Execution Path
Type-II Redundancy: Method to Solve

- Extract Feasible Execution Path
- Constant Folding and Propagation
Type-II Redundancy: Method to Solve

- Extract Feasible Execution Path
- Constant Folding and Propagation
- Dead Code Elimination
Type-II Redundancy: Method to Solve

- Extract Feasible Execution Path
- Constant Folding and Propagation
- Dead Code Elimination

**Diagram:**

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

- Match
  - JP
    - Proto==TCP
    - Port==80
    - Port!=80
      - drop
      - pass
      - False
Type-III Redundancy: Cross-NF Redundancy

- If a monitor deployed before an IDS instance who blocks UDP packets, all the parsing and counting for UDP packets in the monitor is redundant.
Type-III Redundancy: Cross-NF Redundancy

- If a monitor deployed before an IDS instance who blocks UDP packets, all the parsing and counting for UDP packets in the monitor is redundant.

- Method to Solve:
  - Consolidate
  - Eliminate type-I and type-II redundancy
  - Decompose
Outline

• Introduction
• Design Intuition
• NFReducer Implementation
• Preliminary Evaluation
• Conclusion
NFReducer Architecture

- Labeling Critical Variables and Actions

The architecture of NFReducer
**NFReducer Architecture**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic

The architecture of NFReducer
NFReducer Architecture

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Individual NF Optimization

The architecture of NFReducer
NFReducer Architecture

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Individual NF Optimization
- Cross-NF Optimization
NFReducer Architecture

• 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 Architecture**

- 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).

![Diagram of NFReducer Architecture]

1. Source code
2. Labeled Variables & Actions
3. Program Slicer
4. Packet Processing Logic
**NFReducer Architecture**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Individual NF Optimization
  - Apply Configs
  - Extract Paths
  - Constant Folding and Propagation
  - Check Path Feasibility
  - Dead Code Elimination
**NFReducer Architecture**

- Labeling Critical Variables and Actions
- Extracting Packet Processing Logic
- Individual NF Optimization
- Cross-NF Optimization
  - Preliminary discussion on the optimization of different NF chain execution models.
Implementation

LLVM DG Static Slicer
Outline

• Introduction
• Design Intuition
• NFReducer Implementation
• Preliminary Evaluation
• Conclusion
Evaluation: Eliminating Type-I Redundancy

- Setting: Configured with layer-3 rules.

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

- Suricata is more significant
  - inspects packets deeper in payload than Snort.
Evaluation: Eliminating Type-I Redundancy

- Setting: Configured with layer-3 rules.

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

- Suricata is more significant
  - inspects packets deeper in payload than Snort.
Evaluation: Eliminating Type-I Redundancy

- Setting: Configured with layer-3 rules.

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

- Suricata is more significant
  - inspects packets deeper in payload than Snort.
Evaluation: Eliminating Type-I Redundancy

• Setting: Configured with layer-3 rules.

• Increase by nearly 15% for Snort and by 15% to 10X for Suricata (single thread).

• Suricata is more significant
  • inspects packets deeper in payload than Snort.
Evaluation: Eliminating Type-II Redundancy

- Setting: Configured with TCP rules only.
- The larger proportion of UDP packets, the larger performance gain.
- 40% performance gain for Snort and 2.5× for Suricata
Evaluation: Eliminating Type-II Redundancy

- Setting: Configured with TCP rules only.
- The larger proportion of UDP packets, the larger performance gain.
- 40% performance gain for Snort and 2.5× for Suricata
Evaluation: Eliminating Type-II Redundancy

• Setting: Configured with TCP rules only.

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

• 40% performance gain for Snort and 2.5× for Suricata
**Evaluation: Eliminating Type-III Redundancy**

- Setting:
  - Mon—Snort: executed in two processes
  - Mon+Snort: 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 Type-III Redundancy

- Setting:
  - Mon—Snort: executed in two processes
  - Mon+Snort: 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: Overhead

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

• Extracting the packet processing logic:
  • 7.2s for Snort and 1.2s for Suricata

• Eliminating Redundancy:
  • 26.8s for Snort and 83.6s for Suricata (mainly cost by symbolic execution).

• Rebuilding:
  • 0.126s for Snort and 2.753s for Suricata
Conclusion

• Show the existence of the redundant logic in NF programs
• Propose NFReducer to eliminate the redundancy.
  • Takes user labeled information
  • Applies compiler techniques
• Performance gain and overhead of the two example NFs.

• In future, we will:
  • Complete and automate the whole workflow process further.
  • Apply NFReducer to more NFs.
  • Make complete tests on NFReducer.