Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata

Randy Smith
Cristian Estan
Somesh Jha
Shijin Kong

August 20, 2008
Sigcomm 2008, Seattle, WA
Deep Packet Inspection

- Packet content increasingly used to classify net traffic
  - Used for intrusion detection, application identification, quality of service
  - Limited resources: classic time v. space tradeoff

- In this work
  - Techniques that reduce both memory and execution time by an order of magnitude or more
Signature Matching

- Problem: find all signature occurrences that match the payload up to the currently scanned byte
- Example:

  Signatures = {
    sig₁: /(.*)shadow/ ,
    sig₂: /(.*)user(.*)root/ ,
    sig₃: /(.*)[Pp][Aa][Ss][Ss][Ww][Dd]/
  }

<table>
<thead>
<tr>
<th>Header</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0 A8 64 01</td>
<td>60 19 00 15</td>
</tr>
</tbody>
</table>

  - s₃ match!
  - s₂ match!
  - s₃ match!

- Ideal: match signatures simultaneously in a single pass
Signatures

- Regular expressions used to *express* signatures
  - Capture vulnerabilities rather than specific exploits
    - Buffer overflow: `/^RETR\s[^\n]{100}/`
    - Format string: `/^SITE\s+EXEC\s[^\n]*%[^\n]*%/`

- Finite automata used to *match* signatures
  - Simple, well-understood model of computation
  - Combine using standard cross-product operation

- *But there’s a problem...*
State-Space Explosion

- State-space “explodes” under combination
  - Less than 100 signatures requires more than 3 GB!
- Why? combined states = tuples of source states
  - Distinct state for each reachable combination

200 states just for counting!

/.*\na[^\n]{200}/  /.*bc/
XFAs: Extended Finite Automata

- Introduced in *IEEE Symposium on Security and Privacy (Oakland) 2008*

- Problem with DFAs: No distinction between DFA state and computation state

- Idea: extend DFAs with variables that more efficiently track computation state
  - Variables reside in a small “scratch” memory
  - Small programs update variables during matching
Main Contributions

- Formal characterization of state-space explosion

- XFA model
  - XFA algorithms easily adapted from DFA algorithms
  - Framework for systematic optimization
Outline

- State space explosion, formally
- Optimizations
- Experimental Evaluation
Ambiguity

- What are the input sequences that lead to a state?

/.*ab.*cd/

All input sequences leading to Q have the same suffix:
- a, aha, aloha, hiya, aaa, aba,...

Input sequences leading to R have different suffixes:
- ab, abe, abs, ab[^c]+,...

Unambiguous

Ambiguous

- A DFA is *unambiguous* iff all its states are unambiguous
Unambiguous Automata

- Property 1:
  \[ D_1 \text{ and } D_2 \text{ unambiguous} \implies D_1 + D_2 \text{ unambiguous} \]

- Property 2:
  \[ D_1 \text{ and } D_2 \text{ unambiguous} \implies |D_1 + D_2| < |D_1| + |D_2| \]

- Unambiguous automata may be freely combined with no state-space explosion
No State Explosion

\[ */^a b c/ \quad + \quad */^n e t/ \quad = \quad */^a b c/ \quad + \quad */^n e t/ \]

\[ L \quad M \quad N \quad O \quad LR \quad MR \quad NR \quad LS \quad LT \quad LU \quad OR \]

accept(sig1)

accept(sig2)
No State Explosion

\[ \text{./.*net/} \]
\[ \text{./.*abc/} \]

\[ \text{./.*abe/} \]
Exponential Explosion

/.*ab.*cd/ + /.*ef.*gh/
Adding a Bit

/.*ab.*cd/ + /.*ef.*gh/ = Unambiguous!

Unambiguous!
XFAs with Counters

200 states just for counting!

/.*/na[^n]{200}/
XFAs with Counters

\[c-- \text{if} \ (c==0) \ \text{accept}(\text{sig1})\]
\[c = \text{invalid}\]
\[c = 200\]
\[^\backslash n\]
\[^\backslash na\]
\[^\backslash n\]
\[^\backslash nb\]

\[^\backslash n\]
[^\^b]
[^\^bc]

\[^\backslash n\]
[^\^b]
[^\^bc]

\[^\backslash n\]
[^\^b]
[^\^bc]

\[^\backslash n\]
[^\^b]
[^\^bc]

Unambiguous!
Key Contribution

- Ambiguity the culprit in state-space explosion
- XFAs provide a mechanism for controlling ambiguity

Ambiguity
XFA Model

- Start with a DFA, add *update functions* to states

- States and transitions
  - Transitions a function of states and input

- Per-state update and test functions
  - Variable Update a function of states and variable values
Combining XFAs

\[ .*/\text{na}[\text{^n}]\{200\}/ \]

\[ .*/\text{nb}[\text{^n}]\{150\}/ \]

- append instructions to corresponding states
XFA Optimization

- XFA approach: construct individual XFAs, then combine

- Combination “collects” many variables and instructions
  - affects memory size, execution time, per-flow state

- Idea: borrow techniques used in compiler optimization
Optimization: Combine Independent Vars

- Analogous to register allocation used by compilers
  - Map many distinct *logical* vars to fewer *physical* vars

- Reduces instruction count (execution time) and number of variables (per-flow state)

† Analogous to register allocation used by compilers
  - Map many distinct *logical* vars to fewer *physical* vars

† Reduces instruction count (execution time) and number of variables (per-flow state)
Optimization: Code Motion

- Analogous to code motion used by compilers

- *move* instructions to make bits adjacent, *merge* adjacent instructions to a single operation
  - Align on word boundaries, watch out for hazards

- Reduces instruction count (execution time)
Experiment Highlights

- Sizes of combined XFAs up to 10,000x smaller than combined DFAs
  - XFAs typically smaller *and* faster than other methods
- Optimization techniques significantly reduce instruction lengths and per-flow state requirements
Experiment Methodology

- Extracted FTP, SMTP, and HTTP regular expressions from Snort and Cisco rule sets
  - Constructed XFAs and DFAs for each signature
  - Separately combined XFAs, DFAs per-protocol

<table>
<thead>
<tr>
<th>Signature Set</th>
<th>Num Sigs</th>
<th># States (DFA)</th>
<th># States (XFA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cisco FTP</td>
<td>38</td>
<td>&gt; 3.1 M</td>
<td>527</td>
</tr>
<tr>
<td>Cisco SMTP</td>
<td>102</td>
<td>&gt; 3.1 M</td>
<td>3,879</td>
</tr>
<tr>
<td>Cisco HTTP</td>
<td>551</td>
<td>&gt; 3.1 M</td>
<td>11,982</td>
</tr>
<tr>
<td>Snort FTP</td>
<td>72</td>
<td>&gt; 3.1 M</td>
<td>769</td>
</tr>
<tr>
<td>Snort SMTP</td>
<td>56</td>
<td>&gt; 3.1 M</td>
<td>2,415</td>
</tr>
<tr>
<td>Snort HTTP</td>
<td>863</td>
<td>&gt; 3.1 M</td>
<td>15,266</td>
</tr>
</tbody>
</table>
Optimization: Instruction Counts

Signature Set: Snort HTTP

Before Optimization

After Optimization
### Optimization: Per-Flow State

Table: Per-flow state (in bytes) before and after optimization

<table>
<thead>
<tr>
<th>Signature Set</th>
<th>XFA Optimization Before</th>
<th>XFA Optimization After</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cisco FTP</td>
<td>28</td>
<td>10</td>
</tr>
<tr>
<td>Cisco SMTP</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>Cisco HTTP</td>
<td>24</td>
<td>7</td>
</tr>
<tr>
<td>Snort FTP</td>
<td>95</td>
<td>11</td>
</tr>
<tr>
<td>Snort SMTP</td>
<td>66</td>
<td>23</td>
</tr>
<tr>
<td>Snort HTTP</td>
<td>54</td>
<td>36</td>
</tr>
</tbody>
</table>

Summary: \(\sim 3x\) reduction
Performance: Exec Time v. Memory

Cisco SMTP

![Graph showing Performance: Exec Time v. Memory for Cisco SMTP with labels for Multiple DFAs, D2FAs, and XFAs. The graph includes markers for pre-opt and post-opt processing times.]
Performance: Exec Time v. Memory

Snort HTTP

- Multiple DFAs
- D2FAs
- XFAs

Memory usage (bytes logscale)

Processing time (cycles/byte logscale)
Conclusion

- Deep Packet Inspection increasingly important

- Ambiguity is the culprit for state-space explosion
  - Control ambiguity with XFAs

- XFA Model provides framework for optimizations to be systematically applied
  - Optimization effects are significant

- XFAs smaller and faster
Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata

Thank you