XFA: Faster Signature Matching with Extended Automata

Randy Smith
Cristian Estan
Somesh Jha
University of Wisconsin—Madison
Intrusion Prevention Systems (IPS) a critical component of modern network infrastructure

**IPS Operation**

- Intrusion Prevention Systems (IPS) a critical component of modern network infrastructure
Between a Rock and a Hard Place

- Increase in network speeds
  - 1-10 Gbps common, 40 Gbps coming

- Increase in signature counts
  - April 2005 - 3166 Sourcefire rules
  - April 2007 - 8868 Sourcefire rules

- Increase in signature complexity
  - Strings easily evaded, imprecise
  - Regular expressions now standard

- Presence of active adversary
- Must perform at wirespeed
- Computational resources constant
Problem: find all signature occurrences that match the payload up to the currently scanned byte

Example:

Signatures = 

\[ \begin{align*} 
\text{sig}_1: & \quad /(.*)\text{shadow}/, \\
\text{sig}_2: & \quad /(.*)\text{user}.*(.*)\text{root}/, \\
\text{sig}_3: & \quad /(.*)[Pp][Aa][Ss][Ss][Ww][Dd]/ 
\end{align*} \]

Ideal: match signatures simultaneously in a single pass
Signature Trends

- Early IPS systems detected exploits with strings
  - Fast matching, bounded memory, but too weak

- Currently, 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
State-space explosion

- Matching progress encoded only as states

- Combined automaton tracks progress of individual automata simultaneously
  - Combined states = tuples of individual automata states
  - Distinct state for each reachable combination

- For IPS signatures, REs overlap or subsume each other
  - Matching progress *interleaved*
  - Many distinct combinations of reachable states

\[ S_c = (S_1, S_2, S_3, \ldots, S_n) \]
Polynomial Blowup

200 states just for counting!

\[ a \Sigma_{b,c} = b \Sigma_{b,c} \]

Cause: “counting states” subsume other automata
Exponential Blowup

Cause: RE substrings can be interleaved
Time vs. Space

- DFAs
- NFAs

Memory Usage

Ideal

Processing Time
No State Explosion

\[ /.*abc/ \]

\[ /.*abe/ \]
XFA: Extended Finite Automata

- No distinction between DFA state and computation state

- Idea: extend DFAs with variables that more efficiently track computation state
  - Variables reside in a small auxiliary memory
  - Small programs update data during inspection

- Intuition: including variables change shape of automata, make them look like strings
Using Bits

\[ \text{if (b1)} \]
\[ \text{accept(sig1)} \]

/.\text{ab.}.*\text{cd}/

/.\text{ab.}.*\text{cd}/
XFAs with Bits

\[ /.*ab.*cd/ \]

\[ /.*ef.*gh/ \]

n signatures – \( O(n) \) states
XFAs with Bits

/.*ab.*cd/  /.*ef.*gh/

n signatures – \(O(n)\) states
XFAs with Counters

\[ c++; \]
\[ \text{if (c==200)} \]
\[ \text{accept(sig1)}; \]
\[ c = \text{invalid}; \]
\[ c = 0; \]
\[ \text{accept(sig2)}; \]

\[ \Sigma{\text{-ln}} \]
\[ \Sigma{-b} \]
\[ \Sigma{-b-c} \]

\[ \Sigma{\text{-ln}} \]
\[ \Sigma{\text{-ln}} \]
\[ \Sigma{\text{-ln}} \]

\[ a \]
\[ \Sigma{\text{-ln}} \]
\[ \Sigma{\text{-ln}} \]
\[ \Sigma{\text{-ln}} \]

\[ c = 0; \]
\[ c = \text{invalid}; \]
\[ c = 0; \]
\[ c = \text{invalid}; \]

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

n signatures – \( O(n) \) states
XFAs with Counters

\[ a \Sigma - b \Sigma - c = \]

\[ n \text{ signatures} - O(n) \text{ states} \]
XFA Key Contribution

- XFA = DFAs + auxiliary variables
  - Changes shape of automata
  - Tames state space explosion
Outline

- Problem Definition
- XFA Introduction
- XFA Model and Operation
- Experimental Results
- Conclusion
XFAs, formally

- Formally, XFAs are defined as follows:
  - States and transitions
  - Abstract domain (set of integers)
  - Update functions and acceptance conditions

- See paper for details
Construction

- New operators change parse tree and add domain values
  - Parallel concatenation ( # ) adds a bit
    - Breaks up RE into string-like components
    - Set a bit when the left operand accepts
    - Test the bit when the right operand accepts

- ex: / .*ab.*cd/ \rightarrow / .*ab#. .*cd/
Construction

Security expert writes signature

Annotate signature ▸ Construct Automata ▸ Map to high-level types → XFA

Build NXFA

Determin. Minimiz.

ex: /.*ab#.*/

\[
\begin{align*}
\Sigma & \rightarrow \Sigma \\
\ast & \rightarrow a \rightarrow b \\
\ast & \rightarrow c \rightarrow d \\
\end{align*}
\]

\[
\begin{align*}
q_{01} & \xrightarrow{\varepsilon} q_r \xleftarrow{\varepsilon} q_{r1} \\
q_{02} & \xrightarrow{\varepsilon} q_r \xleftarrow{\varepsilon} q_{r2} \\
0 & \xrightarrow{0 \rightarrow 1} \\
1 & \rightarrow 1
\end{align*}
\]
Construction

Security expert writes signature

- Annotate signature
- Construct Automata
- Map to high-level types

ex: /.*ab#.*cd/

Epsilon elimination
State determinization
Data determinization
State reduction
Data reduction

Non-deterministic XFA

Diagram showing a transition graph with states P, Q, R, and S, and transitions on symbols a, b, c, d, and transitions 0 → 1.
Construction

- Set-based update functions cumbersome
  - Too abstract, too much memory, difficult to combine

- Templates tie domain values and update functions to efficient data types
Mapping to Efficient Data Types

```c
template bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {  (no-op,   {0→0, 1→1}, ""),
               (bit set, {0→1, 1→1}, "bitSET %d"),
               (bit clear, {0→0, 1→0}, "bitCLR %d"),
               (bit flip, {0→1, 1→0}, "bitFLP %d") },

    Amap = {   (nonaccepting,   {}, ""),
               (accept_ifset, {1}, "if (bit %d) alert"),
               (accept_ifclear, {0}, "if !(bit %d) alert"),
               (unconditional, {0,1}, "alert")
            }
};
```
Mapping to Efficient Data Types

template bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {
        (no-op, {0→0, 1→1}, ""),
        (bit set, {0→1, 1→1}, "bitSET %d"),
        (bit clear, {0→0, 1→0}, "bitCLR %d"),
        (bit flip, {0→1, 1→0}, "bitFLP %d")
    },
    Amap = {
        (nonaccepting, {}, ""),
        (accept_ifset, {1}, "if (bit %d) alert"),
        (accept_ifclear, {0}, "if !(bit %d) alert"),
        (unconditional, {0,1}, "alert")
    }
};
template bit_t = {
D = {0..1},
d0 = 0,
Ufmap = {
(no-op,   {0→0, 1→1}, ""),
(bit set, {0→1, 1→1}, "bitSET %d"),
(bit clear, {0→0, 1→0}, "bitCLR %d"),
(bit flip, {0→1, 1→0}, "bitFLP %d")
},
Amap = {
(nonaccepting, {}, ""),
(accept_ifset, {1}, "if (bit %d) alert"),
(accept_ifclear, {0}, "if !(bit %d) alert"),
(unconditional, {0,1}, "alert")
};
Mapping to Efficient Data Types

template bit_t = {
    D = {0..1},
    d0 = 0,
    Ufmap = {
        (no-op,         {0→0, 1→1}, ""),
        (bit set,       {0→1, 1→1}, "bitSET %d"),
        (bit clear,     {0→0, 1→0}, "bitCLR %d"),
        (bit flip,      {0→1, 1→0}, "bitFLP %d")
    },
    Amap = {
        (nonaccepting,  {}, ""),
        (accept_ifset,  {1}, "if (bit %d) alert"),
        (accept_ifclear, {0}, "if !(bit %d) alert"),
        (unconditional, {0,1}, "alert")
    }
};
Mapping to Efficient Data Types

```
template bit_t {
    D = {0..1},
    d0 = 0,
    Ufmap = {
        (no-op, {0→0, 1→1}, ""),
        (bit set, {0→1, 1→1}, "bitSET %d"),
        (bit clear, {0→0, 1→0}, "bitCLR %d"),
        (bit flip, {0→1, 1→0}, "bitFLP %d")
    },
    Amap = {
        (nonaccepting, {}, ""),
        (accept_ifset, {1}, "if (bit %d) alert"),
        (accept_ifclear, {0}, "if !(bit %d) alert"),
        (unconditional, {0,1}, "alert")
    }
};
```
Mapping to Efficient Data Types

template  bit_t = {
D = {0..1},
d0 = 0,
Ufmap = {(no-op, {0→0, 1→1}, "")},
(bit set, {0→1, 1→1}, "bitSET %d"),
(bit clear, {0→0, 1→0}, "bitCLR %d"),
(bit flip, {0→1, 1→0}, "bitFLP %d") },

Amap = {(nonaccepting, {}, ","),
(accept_ifset, {1}, "if (bit %d) alert"),
(accept_ifclear, {0}, "if !(bit %d) alert"),
(unconditional, {0,1}, "alert")
};
Mapping to Efficient Data Types

template bit_t = {
  D = {0..1},
  d0 = 0,
  Ufmap = {
    (no-op, {0→0, 1→1}, ""),
    (bit set, {0→1, 1→1}, "bitSET %d"),
    (bit clear, {0→0, 1→0}, "bitCLR %d"),
    (bit flip, {0→1, 1→0}, "bitFLP %d")
  },
  Amap = {
    (nonaccepting, {}, ""),
    (accept_ifset, {1}, "if (bit %d) alert"),
    (accept_ifclear, {0}, "if !(bit %d) alert"),
    (unconditional, {0,1}, "alert")
  }
};
Combination and Matching

- Straightforward extensions to DFA counterparts

- Combination
  - Perform standard cross product combination
  - “Append” instructions to transitions and states
    - Data type instances are independent

- Matching
  - Execute attached instructions when transitions followed
Outline

- Problem Definition
- XFA Introduction
- XFA Model and Operation
- Experimental Results
- Conclusion
Experiment Highlights

- Combined XFA only slightly larger than sum of sizes of individual automata

- Combined XFA smaller *and* faster than other techniques
Test Environment – Signatures

- 1450 Regular expressions extracted from Snort HTTP

- Retain semantics of Snort rules:

```plaintext
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
  (msg:"WEB-CGI dcboard.cgi invalid..."; flow:to_server,est;
  uricontent:"/dcboard.cgi"; content:"command=register";
  content="&admin"; ... sid:817; rev:10;)

/.*command=register#(.*)&admin) | (.*&admin#(.*)command=register)/
```

- Snort2Bro and custom scripts to compose expressions
- Most annotations automatic
Test Environment – XFA

- Characteristics of combined XFA:
  - 41,994 total states → 42 MB (additive sum: 36,631 states)
  - 195 bits (~25 bytes) of aux memory
  - Instruction memory: 3.5 MB

Table: Distribution of instructions on edges and states

<table>
<thead>
<tr>
<th>#instrs</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>% edges</td>
<td>1.0</td>
<td>94.8</td>
<td>2.7</td>
<td>0.47</td>
<td>0.80</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.13</td>
</tr>
<tr>
<td>% states</td>
<td>78.9</td>
<td>20.0</td>
<td>0.9</td>
<td>0.03</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.05</td>
</tr>
</tbody>
</table>
DFA Set Splitting

- DFA set splitting ("Multiple DFAs") (Yu et al. ANCS’06):
  - Given memory bound $M$, partition REs so that:
    - The number of RE subsets is as small as possible
    - The set of combined automata is less than $M$
  - Use heuristics to group REs that behave well
  - Traces a curve in time vs. space between DFAs and NFAs
    - Increasing the memory ceiling decreases the number of combined automata
Memory vs. Execution Time

- NFAs
- DFAs
- Multiple DFAs
- XFA

Memory usage (bytes logscale)

Processing time (seconds/gigabyte logscale)
Limitations and Future Work

- Expression coverage
  - Complex data types require complex templates

- Hi-level mapping time
  - Backtracking match algorithm can be time-intensive

- Other issues
  - Further work coming in Sigcomm 2008
Conclusion

- DFAs for regular expressions often blow up when combined

- XFAs = DFAs + auxiliary variables
  - Changes shape of automata
  - Tames state space explosion

- Result: compared to other feasible approaches, reduce both time and space
XFA: Faster Signature Matching with Extended Automata

Thank you