A High Throughput String Matching Architecture for Intrusion Detection and Prevention

Lin Tan, Timothy Sherwood

Appeared in ISCA 2005

Presented by: Sailesh Kumar
Discussion Leader: Max Podlesny
Overview of IDS/IPS systems
   » String matching and regular expression matching

String matching algorithms
   » Aho-Corasick algorithm

Overview of the proposed architecture

Analysis of the proposed architecture

Conclusion
IDS Overview

- Intrusion detection systems, IDS must be capable of distinguishing between normal (not security-critical) and abnormal user activities
  » However translating the user behaviors into a security-related decision is often not that easy

- One approach is **anomaly detection**
  » Anomaly detectors construct profiles that represent normal usage
  » If profile of current data doesn’t match the normal profile => a possible attack
  » Typically results in high false positives and few negatives too
  » An intelligent intruder can train the system to behave otherwise

<table>
<thead>
<tr>
<th>Normal</th>
<th>Unpredictable</th>
<th>Abnormal</th>
</tr>
</thead>
</table>
IDS Overview

- Another approach is some kind of **signature detection**

- Misbehavior signatures fall into two categories:
  - **Attack signatures** – action patterns that may pose a security threat. Typically presented as a series of activities interleaved (may be) with neutral ones
    - E.g. sequence of flags set on some specific sequence of packets
  - **Selected text strings** – signatures to match text strings which look for suspicious action (e.g. – calling /etc/passwd).
    - Effective against attacks that exploits programming flaws.

- Effective but not immune to novel attacks
  - Difficulties in updating information on new types of attacks
  - Still most widely used
IDS Overview

- At the heart of most signature based IDS systems, lies a string (regular expression) matching engine
  » Since this engine has to scan every byte of data, it might become the throughput bottleneck

- This paper presents a high throughput string matching architecture

- Advantages of proposed scheme:
  » Claimed to be space efficient
    – Entire rule set can be stored on-chip
  » Claimed to be able to achieve high throughput

- Limitations (will cover later)
  » New systems do regex matching, FSM is typically too large to fit on chip, even with the proposed scheme.
  » Simple FSM compression schemes may beat the proposed idea
String Matching Algorithm (Aho-Corasick)

- For a set $P$ of patterns, it builds a DFA, which accepts all patterns in $P$. 
String Matching Algorithm (Aho-Corasick)

- Lets say, $P$ is \{he, she, his, hers\}

**Initial State**

**Transition Function**

**State**

**Accepting State**

Example from Lin Tan et. al.
Properties of DFA created by Aho-Corasick

- Consumes one character per state traversal
  - A modified automaton is also used in many systems, which rescans a character if there is no outgoing edge for it (fail ptr)
    - Reduces the DFA size tremendously

Example from Lin Tan et. al.
How to implement efficiently on-chip

**Problem:** Size too big to be on-chip
- ~ 10,000 nodes for SNORT rule set with ~1000 strings
- 256 out edges per node
- Requires 10,000*256*14 = ~5MB

**Solution:** partition each state machine into small state machines such that
- Each machine handles a subset of strings only
- Also, each machine has few outgoing edges only
Bit-Splitting Algorithm Overview

- **Bit-Split String Matching Algorithm**
  » Reduces out edges from 256 to 2.

- Partition the rule set P into smaller sets $P_0$ to $P_n$

- Build AC state-machine for each subset $P_i$

- For each $P_i$, rip its DFA apart into 8 tiny state-machines, $B_{i0}$ through $B_{i7}$

- Each binary FSM operates on 1 bit of the 8 bit input characters
  » A match is announced only if all 8 machines finds a match
Bit-splitting String Matching Architecture

- Lets say we have 4 strings in P {he, she, his, hers}

- Consider FSM 3
  » It looks at 3rd bit of the input char and makes state transitions based upon whether it is 0 or 1

- Example, lets say our input stream is hxhe

<table>
<thead>
<tr>
<th>Char</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>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>h</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Binary Encoding of input stream “hxhe”

Example from Lin Tan et. al.
Bit FSM Construction

- Use subset construction algorithm to build bit FSMs

Example from Lin Tan et. al.
Bit FSM Construction

- Remove the non-accepting states from the small FSMs

Example from Lin Tan et. al.
Bit-splitting String Matching Architecture

- Build all bit FSMs $B_i$
- Match announced, if all bit FSMs announce a match

Example from Lin Tan et. al.
An Example Matching

Only if all the state machines agree, is there actually a match.

Example from Lin Tan et. al.
How to map it to hardware efficiently

**Problem:**
- Every state in the bit-FSMs $B_i$ might have up to $S$ accepting states of the original FSM $D$.
  - Note that $\#$ accepting of states $= \#$ of strings $= S$
- Even if we keep bit-vector to represent the accepting states of the original $D$, we will need $S$-bits per state of $B_i$.
- $S$ is typically $>1000$, so not practical

**Solution:**
- Partition the rule set $P$ into smaller subsets $P_i$ each with $g$ rules
  - Build FSM $D_i$ for each $P_i$ and bit-splice $D_i$ into $B_{i0}, \ldots, B_{i7}$
- Now every state in $B_{ij}$ can have at most $g$ accepting states, hence only $g$-bits are needed
How to map it to hardware efficiently

Also note that any D FSM can be ripped into 2 or 4 bit-FSMs as well

» When ripped into 4 different bit-FSM, each one consumes 2-bits of the input characters
» When ripped into 2 different bit-FSM, each one consumes 4-bits
Each State Machine Tile

- $g = 16$
- # of states = 256 (8-bit state encoding)
- D FSM is ripped into 4 FSMs (each accepts 2-bit I/P)

Example from Lin Tan et. al.
Hardware Implementation

Example from Lin Tan et. al.
Non Interrupting Update

Issues:
- Must copy the state of every flip-flop of the affected module onto the replacement module
  - E.g. the current state pointer of the affected module must be copied into the replacement module
  - These issues are not mentioned in the paper
Optimal partitioning

- There are two types of partitions.
  - One is the partition of the set of strings
  - Another is the partitions of each D FSM into multiple bit FSMs
    - 8 binary partition is clearly easy to understand
    - However, it is possible to partition into 2 or 4 partitions, and each partition consumes 4 and 2 input bits of the input character respectively.

- Terminology
  - $S =$ total # of strings
  - $g =$ # of strings per group (# of partition = $S/g$)
  - $n =$ # of partitions of state machine (1, 2, 4, 8)
  - $L =$ Average # of characters per string

- How to choose $g$ and $n$?
  - which consumes minimum area
Optimal partitioning

Total number of bits needed is:

\[ T_{n,g} = n \left\lfloor \frac{S}{g} \right\rfloor 2^{\left\lfloor \log_2(gL) \right\rfloor} \left( \left\lfloor \log_2(gL) \right\rfloor 2^{\frac{S}{n}} + g \right) \]

- # of FSM partitions
- Total # of states in each bit-FSM
- # of bits to encode state
- # of partial match bits
- # of outgoing edges per state
- Total # of modules (string set partition)
Optimal partitioning

- Plot from the review of Mike Wilson (1000 signatures, average length 12)
  » Typical optimal # of FSM partitions = 2, 4
In the analysis presented in the paper, it is claimed that partitioning string sets always reduce space.

» Not quite right when there is overlap among the strings.

- With overlapping strings, fewer states are needed w/o partitioning.
Performance Results - Memory

From Lin Tan et. al.
Performance Results - Throughput

From Lin Tan et. al.
Strength of the Algorithm

- Can be very effective for dense DFAs, when there are plenty of outgoing edges from every state

- In this case, path compression will not help a lot

- However, ripping apart the state machines into bit state machines will reduce the number of outgoing edges to 16
  - (2 edges x 8 FSMs) or (4 edges x 4 FSMs)

  - For dense graphs, upto 16 times reduction in state space
Some Issues

- Tuck [30] used bitmap compression and path compression to reduce the amount of memory needed for SNORT rules to 1.1MB.
- Note that, Tuck didn’t do any string set partitioning.
  - w/o any partitioning, bit-splitting will consume >2 MB.

From Lin Tan et. al.
Conclusion

- **Novel Bit-Split String Matching Algorithm**
  - Reduces out edges from 256 to 2
  - Can be extremely effective for dense graphs

- **Performance/area beats the best techniques by a factor of 10 or more.**

- **0.4MB and 10Gbps for Snort rule set ( >10,000 characters)**
Thank you and Questions