NTNU

Approximate search in intrusion detection

Slobodan Petrović, NTNU Gjøvik, Norway
COINS summer school, Metochi, July 2018

NTNU

Contents

- Fundamental concepts
- Search techniques for IDS
- The Aho-Corasick algorithm
- Bit-parallelism and exact search
- Approximate search in IDS

Fundamental concepts

- Intrusion
  - A set of actions aimed at compromising the security goals (confidentiality, integrity, availability of a computing or a networking resource)
- Intrusion detection
  - The process of detecting and identifying intrusion activities

Fundamental concepts

- Intrusion prevention
  - The process of both detecting/identifying intrusion activities and managing responsive actions throughout a computer system or a network
- Intrusion detection system (IDS)
  - A system that *automatically* performs the process of intrusion detection

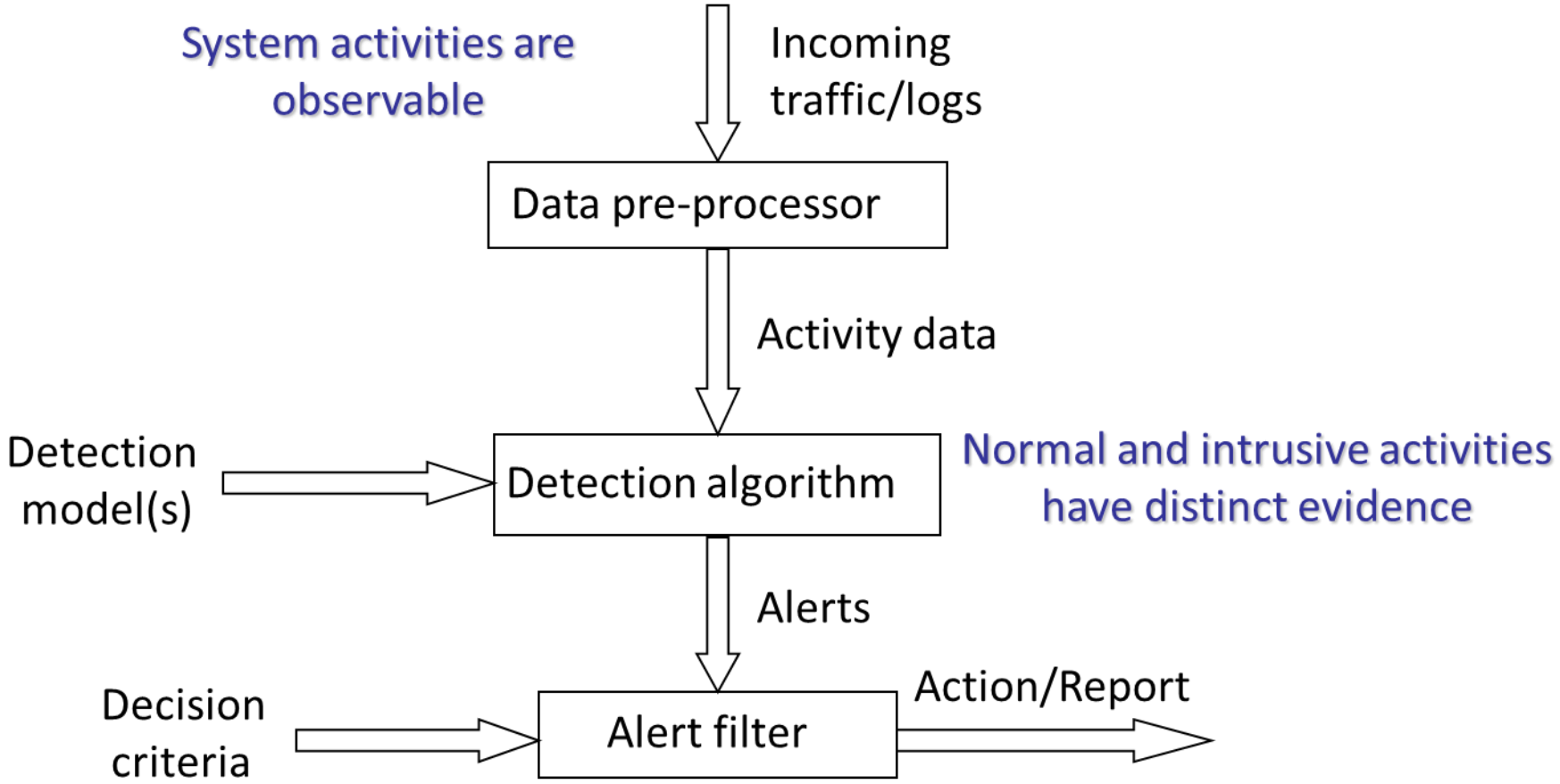
Fundamental concepts

- Intrusion prevention system (IPS)
  - A system that automatically detects/identifies intrusions and manages responsive actions
  - A convergence of a firewall and an IDS
    - The IDS component detects and identifies the malicious traffic
    - The firewall component prevents the malicious traffic from entering/exiting the network
  - For high performance, uses sophisticated *search algorithms* (can be implemented in hardware - ASIC, FPGA)

Fundamental concepts

- Basic assumptions regarding operation of an IDS/IPS
  - System activities are observable
    - To detect/identify intrusions, the IDS/IPS must be connected to the defended system
      - If some content is encrypted, the IDS must be able to decrypt it
  - Normal and intrusive activities have distinct evidence
    - Very often, the differences between normal and intrusive activities/traffic are very small
      - The task of an IDS/IPS is to detect these differences

Fundamental concepts



Fundamental concepts

- Data pre-processor
  - Collects and formats the input data
- Detection algorithm
  - Based on the detection model, detects the difference between ”normal” and intrusive traffic/log events
- Alert filter
  - Estimates the severity of alerts and warns the operator or manages responsive activities (usually blocking)

Fundamental concepts

- Incoming traffic/log data
  - Packets
    - The header contains routing information
    - The content may also be important (and *is* more and more) for detecting intrusions
  - Logs
    - A chronological set of records of system activity

Fundamental concepts

- Detection algorithm
  - Checks the incoming data for presence of anomalous content – *search*
  - A major detection problem
    - There is no sharp limit between “normal” and “intrusive”
  - Consequence – *false positives*

Fundamental concepts

- IDS classification
  - By scope of protection (by location)
    - *Host-based IDS*
    - *Network-based IDS*
    - Application-based IDS
    - Target-based IDS
  - By detection model
    - Misuse detection
    - Anomaly detection

Fundamental concepts

- Host-based IDS
  - Collect data from sources internal to a host, usually at the operating system level (various logs etc.)
  - Monitor user activities
  - Monitor execution of system programs

Fundamental concepts

- Network-based IDS
  - Collect network packets
  - Have sensors deployed at strategic locations
  - Inspect network traffic
  - Monitor user activities on a network

Fundamental concepts

- Application-based IDS
  - Collect data from running applications
    - Typically, large applications such as database systems
  - The data sources include
    - Application event logs
    - Other data collections internal to the application

Fundamental concepts

- Target-based IDS (integrity verification)
  - Generate their own data
    - By adding code or a hash value/checksum to the executable
  - Use these checksums or cryptographic hash functions to detect alterations to system objects and then compare these alterations to a policy
  - Trace calls to other programs from the monitored application

Fundamental concepts

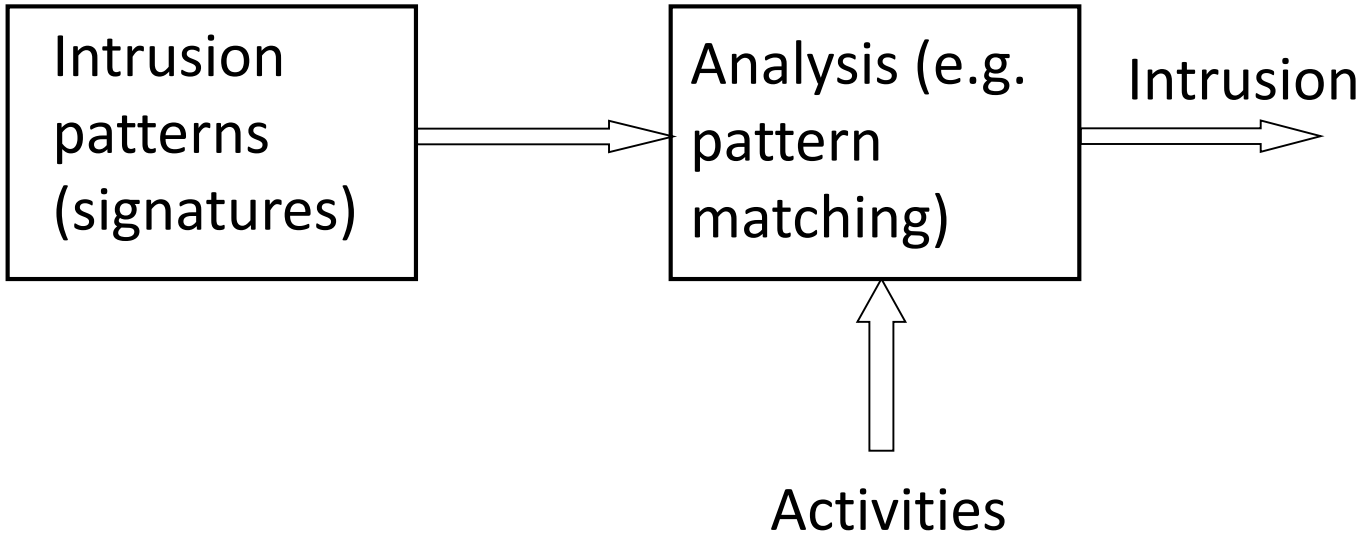
- Misuse detection (1)
  - Gathering information about indicators of intrusion in a database in advance
  - Determining whether these indicators can be found in incoming data

Fundamental concepts

- Misuse detection (2)
  - For misuse detection, the following is needed
    - Good understanding of what constitutes misuse behavior
      - Intrusion patterns, or *signatures*
    - A reliable record of user activity
    - A reliable technique for analyzing the activity record
      - Very often – pattern matching, which includes *search*

Fundamental concepts

- Misuse detection (3)



Signature example: **if** src_ip = dst_ip **then** “land attack”

Fundamental concepts

- Misuse detection (4)
  - Best suited for reliably detecting known misuse patterns
    - By means of signatures
  - Impossible to detect previously unknown attacks (*zero-day*)
    - A single bit of difference in misuse patterns is enough for an IDS to miss a new attack
    - It is possible to use the existing knowledge (for instance, of consequences of attacks) to recognize *new forms of old attacks*

Fundamental concepts

- Misuse detection (5)
  - *False positives*
    - Misuse detection systems sometimes generate alerts even if the activities are in fact normal
      - Normal activities often closely resemble the suspicious ones
      - The attackers do their best to achieve a high level of similarity between normal activities and attacks
  - Careful adjustment of the IDS parameters is needed to reduce the number of false positives

Fundamental concepts

- Misuse detection (6)
  - New (zero-day) attacks require new signatures
  - The increasing number of attack signatures causes the signature databases to grow over time

Fundamental concepts

- Misuse detection (7)
  - Every data unit (a packet or a session) must be compared to each signature for the IDS to detect intrusions
    - Computationally expensive as the bandwidth increases
  - When the bandwidth overwhelms the capabilities of the IDS, it causes the IDS to miss or *drop* packets
    - In such a situation, *false negatives* are possible
      - Attack traffic present in the dropped packets

Fundamental concepts

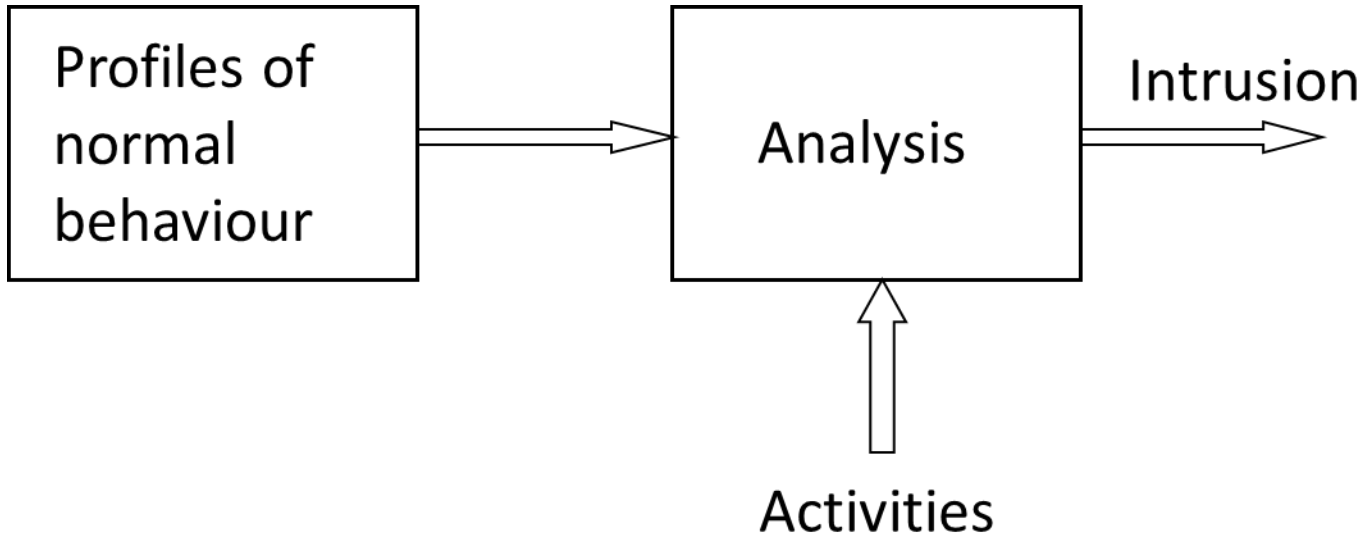
- Anomaly detection (1)
  - Establishing profiles of normal user/network behavior
  - Comparing actual behavior to those profiles
  - Alerting if deviations from the normal behavior are detected
  - Profiles are often defined as sets of *metrics*
    - Measures of particular aspects of user/network behavior
    - A metric is associated with either a threshold or a range of values

Fundamental concepts

- Anomaly detection (2)
  - *False positives*
    - Strange behavior patterns do not always indicate intrusions
      - Sometimes, rare traffic sequences represent normal behavior
    - This is a major problem in anomaly detection
  - *False negatives*
    - If the IDS thresholds are set too high, we may miss the attacks

Fundamental concepts

- Anomaly detection (3)



Fundamental concepts

- Anomaly detection (4)
  - In misuse detection, the analysis engine alerts if the analyzed activity *matches* an entry in the signature database
  - In anomaly detection, the analysis engine alerts if the analyzed activity *does not match* any of the established profiles of normal behavior
  - Search through the *whole database* is needed for each analyzed activity (e.g. for each packet) in the worst case in both misuse and anomaly detection

Fundamental concepts

- Anomaly detection (5)
  - The number of profiles of normal behavior is much larger than the number of known attacks
    - If we would want to make a database of normal behavior profiles, this database would be too large for efficient real time processing that is needed in an IDS
    - Consequently, a size reduction of such a database is needed
      - Then, the profiles of normal behavior are not precise enough
      - Consequence – a large number of false positives, larger than with misuse-based IDS in general

Search techniques for IDS

- Checking traffic against a large database is a resource-intensive task that involves *search*
  - Because of that, the choice of the search algorithm is very important in a misuse based IDS
  - In most practical cases, *exact search* is used
    - For every variation of the same attack, a new signature must be produced, otherwise an exact search algorithm fails to detect it
    - To avoid this, *approximate search* can be used
      - Still in the experimental phase, but promising

Search techniques for IDS

- When we discuss IDS that implement search algorithms, we usually refer to *network-based systems*
  - Most often used open-source and commercial IDS (e.g. Snort, Suricata, etc.) are network-based

Search techniques for IDS

- The detection module is the core IDS component
  - Compares the header/content of the analyzed packet or a reassembled set of packets (a *session*) with the attack signatures the IDS is aware of
  - For such a comparison, a *search algorithm* is necessary
    - Detects known attack patterns in the intercepted traffic
  - Efficiency of the detection module depends on the *choice* of the search algorithm and its efficient *implementation*

Search techniques for IDS

- The factors that determine IDS search efficiency (1)
  - The need for multi-pattern search
  - The case sensitivity of search patterns
  - The size of the search patterns
  - The number of search patterns processed during a single rule/signature processing
  - The size of the alphabet

Search techniques for IDS

- The factors that determine IDS efficiency (2)
  - The possibility of launching so-called *algorithmic attacks* against the IDS
    - Here, the attacker deliberately sends traffic that is difficult to analyze by the IDS search algorithm
      - Happens when the *average time complexity* of the search algorithm is different from the *worst-case time complexity*
  - The search text size
  - The frequency of searches

Search techniques for IDS

- The need for multi-pattern search
  - Processing of a signature often requires search for multiple patterns in an analyzed data unit (packet or session)
    - Attack signatures can be very complex

Search techniques for IDS

- The case sensitivity of search patterns
  - In order to mitigate IDS evasion, case insensitive search is desirable
    - Changing case of the keywords by the attacker may lead to missing the detection by the IDS
  - If a search pattern is found in the analyzed packet, an additional check might be performed whether there are any issues in the signature that have anything to do with the case of the letters

Search techniques for IDS

- The size of the search pattern
  - For some search algorithms, the size of the search pattern affects their efficiency to a great extent
  - So-called *skip algorithms* (e.g. Wu-Manber) are particularly sensitive to this
    - Their worst-case complexity might be much higher than the average-case complexity
    - Inadequate for application in intrusion detection modules
      - Sensitive to algorithmic attacks

Search techniques for IDS

- The number of search patterns processed during a single signature processing
  - In general, the performance of a search algorithm is reduced as the number of search patterns increases
    - Consequence of reduced benefit of usage of processor cache memory, due to the fact that the size of this memory is limited
    - This inevitable degradation of performance should be sub-linear
      - Important for scalability

Search techniques for IDS

- The size of the alphabet
  - Has significant influence on the search algorithm efficiency
  - In IDS, alphabets are large (e.g. UNICODE)
  - Any search algorithm used in IDS must be efficient enough to cope with this size of the alphabet

Search techniques for IDS

- The possibility of launching algorithmic attacks (1)
  - Algorithmic attacks
    - Exploit the fact that some search algorithms used in IDS detection modules have the average-case complexity much better than the worst-case complexity
    - By launching specially designed packets that contain patterns, whose processing is less efficient, the attackers may significantly reduce performance of the IDS

Search techniques for IDS

- The possibility of launching algorithmic attacks (2)
  - The search algorithms, whose average-case and worst-case complexities do not differ too much are especially convenient to be used in IDS detection modules
    - The *Aho-Corasick algorithm* - very often used
      - Multi-pattern search – all the patterns detected in one pass
      - The average-case complexity and the worst-case complexity are the same

Search techniques for IDS

- The size of the search text (1)
  - The size of the search text in IDS may vary
    - It can be very small, just a few bytes
      - Example - short packets, such as the ICMP packets
    - It can also be very long
      - Example - long HTTP reassembled sessions

Search techniques for IDS

- The size of the search text (2)
  - Implementation of any search algorithm requires *initialization*
    - Certain fixed number of operations before the very execution
    - The cost per processed byte of search text may be high, especially if the string is short
  - Because of that, search algorithms requiring less initialization operations are more convenient for application in IDS detection modules

Search techniques for IDS

- The frequency of searches
  - May be high in high bandwidth networks
  - The frequency of execution of the search algorithm and the size of the search text (i.e. lengths of the inspected packets) are related
    - The high frequency of execution of the search algorithm might make the search setup costs high per processed byte
  - Better search algorithms for IDS have fewer setup operations

The Aho-Corasick algorithm

- Bearing in mind the factors that determine IDS efficiency
  - The *Aho-Corasick* search algorithm is very often chosen to be implemented in misuse-based IDS
    - Its properties make it more suitable for IDS search than the other search algorithms

The Aho-Corasick algorithm

- Properties of the Aho-Corasick algorithm (1)
  - A multi-pattern search algorithm
    - Its performance can be further improved by pre-processing (using *Deterministic Finite Automaton* (*DFA*))
  - The case-sensitivity issue is easy to handle
  - The size of the search patterns does not affect the time performance, it only affects memory consumption
  - The scalability is good, regarding the number of search patterns

The Aho-Corasick algorithm

- Properties of the Aho-Corasick algorithm (2)
  - Can process large alphabets, without performance degradation
  - The average-case and the worst-case complexities are the same – the algorithmic attacks against IDS implementing this search algorithm have no effect
  - The setup process is efficient enough
    - It is possible to process short packets and frequent packets in a satisfactory way

The Aho-Corasick algorithm

- Consists of 2 phases
  - In the first phase, a *pattern matching machine* (a Finite State Machine – FSM) is constructed
    - A directed graph describing transitions between the states of the machine after receiving certain input characters (appearance of an input character is called an *event* in FSM terminology)
  - In the second phase, the input string is processed and all the patterns from the given set of search patterns are found in it in a single pass

The Aho-Corasick algorithm

- The pattern matching machine (1)
  - Let $Y = \{y_1, y_2, \ldots, y_n\}$ be the set of search patterns and let X be the search string
  - The string X and the members of Y consist of symbols from the alphabet $\mathcal{A}$
  - The search problem
    - Locate and identify all substrings of X that are patterns in Y
    - There may be overlap of the substrings

The Aho-Corasick algorithm

- The pattern matching machine (2)
  - Given the set Y, the pattern matching machine for Y is a structure that takes X as input and produces as output the locations in X, at which the patterns from Y appear as substrings
  - The pattern matching machine consists of nodes called *states*, labeled with numbers

The Aho-Corasick algorithm

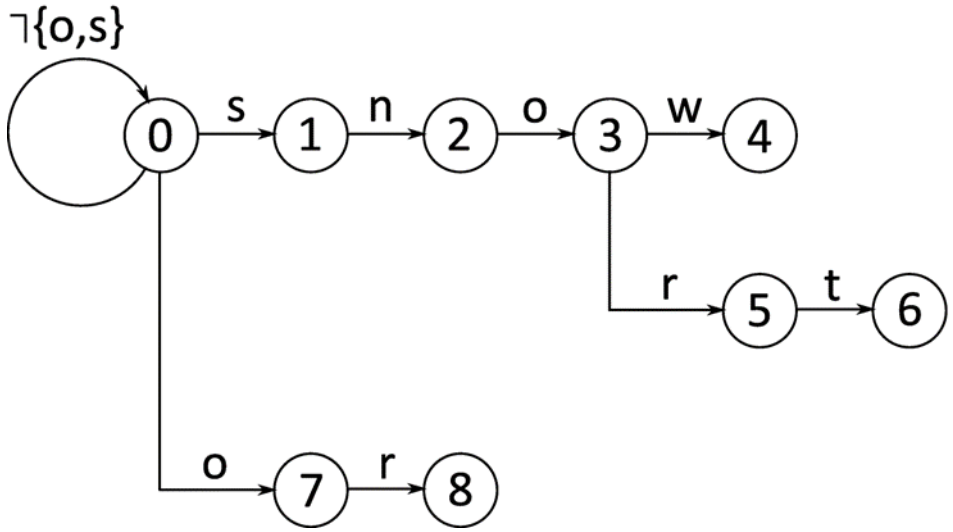
- The pattern matching machine (3)
  - The machine takes input symbols from X, one at a time, and processes them by making state transitions and occasionally producing output
  - The output of the machine consists of patterns from Y, together with the state labels of the machine where they were detected

The Aho-Corasick algorithm

- The pattern matching machine (4)
  - A pattern matching machine is defined by means of three functions
    - A *goto transition function* g that maps a pair (state, input symbol) into a state or a message *fail*
    - A *failure transition function* f that maps a state into a state
    - An *output function* o that associates a set of patterns from Y (possibly empty) to any state

The Aho-Corasick algorithm

- The pattern matching machine (5)
  - Example (1)
    - Let $Y = \{snow, snort, or\}$
    - The goto transition function g



The Aho-Corasick algorithm

- The pattern matching machine (6)
  - Example (2)
    - The failure transition function f for Y

i	1	2	3	4	5	6	7	8
$f(i)$	0	0	7	0	8	0	0	0

    - f is not defined in the state 0 since no input can produce the message *fail* from that state

The Aho-Corasick algorithm

- The pattern matching machine (7)
  - Example (3)
    - The output function o for Y

i	1	2	3	4	5	6	7	8
$o(i)$	$\{\}$	$\{\}$	$\{\}$	$\{snow\}$	$\{or\}$	$\{snort\}$	$\{\}$	$\{or\}$

The Aho-Corasick algorithm

- The pattern matching machine (8)
  - To construct the goto function and the failure function, two separate algorithms are used
  - The output function is constructed during the execution of both algorithms
    - Initialized during the construction of the goto function
    - Updated during the construction of the failure function

The Aho-Corasick algorithm

- The pattern matching machine (9)
  - Constructing the goto function
    - Input - the set Y of search patterns
    - Start with a single vertex 0
    - Each pattern from Y defines a separate path in the goto graph
    - That same search pattern is added to the output function of the state at which the path terminates
    - At the end, a loop is added in the state 0, corresponding to the transitions on input symbols different than the initial symbols of the patterns from Y

The Aho-Corasick algorithm

- The pattern matching machine (10)

Algorithm 1

Construction of the goto function

Input: A set of search patterns $Y = \{y_1, y_2, \ldots, y_n\}$
Output: The goto function g that corresponds to the set Y and partially computed output function o
Remarks: $o[s] = \{\}$ when the state s is first created. $g[s, \alpha] = \text{fail}$ if α is not defined or $g[s, \alpha]$ has not yet been defined

```
begin
    newState ← 0;
    for i ← 1 to n do
        Let y_i = α_1 α_2 ··· α_m;
        state ← 0; j ← 1;
        while g[state, α_j] ≠ fail do
            state ← g[state, α_j];
            j ← j + 1;
        end
        for k ← j to m do
            newState ← newState + 1;
            g[state, α_k] ← newState;
            state ← newState;
        end
        o[state] = y_i;
    end
    forall the α such that g[0, α] = fail do
        g[0, α] ← 0;
    end
end
```

The Aho-Corasick algorithm

- The pattern matching machine (11)
  - The failure function determines the state to which the pattern matching machine returns if a ”wrong” character is encountered at the input, after the machine has found itself in a non-initial state

The Aho-Corasick algorithm

- The pattern matching machine (12)
  - The failure function f is constructed in the *breadth-first* manner
    - Let *depth* of a state s be length of the shortest path from 0 to s
    - The values of f are first determined at the depth 1, and they are all equal to 0
    - For the depth d, the values of the failure function are obtained based on the values for the depth $d-1$
    - The values of the output function are updated during the execution of this algorithm

The Aho-Corasick algorithm

- The pattern matching machine (13)

Algorithm 2

Construction of the failure function

Input: The goto function g and the output function o obtained by means of Algorithm 1
Output: The failure function f and the updated output function o
Remarks: $queue$ is a FIFO (First-In-First-Out) structure. The operation "+" on $queue$ means inserting an element at the last position in $queue$ whereas the operation "−" means taking out the first element of $queue$

```
begin
    queue ← [ ];
    foreach α such that g[0, α] = s ≠ 0 do
        queue ← queue + s;
        f[s] = 0;
    end
    while queue ≠ [ ] do
        Let r be the next state in queue;
        queue ← queue − r;
        foreach α such that g[r, α] = s ≠ fail do
            queue ← queue + s;
            state ← f[r];
            while g[state, α] = fail do
                state ← f[state];
            end
            f[s] ← g[state, α];
            o[s] ← o[s] ∪ o[f[s]];
        end
    end
end
```

The Aho-Corasick algorithm

- The pattern matching machine (14)
  - The pattern matching machine obtained with the algorithms 1 and 2 is called *Non-deterministic Finite Automaton* (*NFA*)
    - In general, it makes more than one state transition after receiving a single input symbol

The Aho-Corasick algorithm

- The pattern matching machine (15)
  - It can be shown that any NFA can be transformed into a *Deterministic Finite Automaton* (*DFA*)
    - Makes only one state transition after receiving an input symbol
  - Transforming NFA to DFA can sometimes contribute to improved efficiency of the Aho-Corasick algorithm

The Aho-Corasick algorithm

- Processing of an input string (1)
  - A pattern matching machine constructed by means of the algorithms 1 and 2 is capable of finding all the patterns from the given pattern set Y in any input string X
  - The symbols from X are fed into the machine in a serial manner

The Aho-Corasick algorithm

- Processing of an input string (2)
  - Whenever the machine finds a pattern from Y in X, it produces non-empty output string(s) corresponding to the value of the output function o at the state where the pattern is found

The Aho-Corasick algorithm

- Processing of an input string (3)
  - Each input symbol starts an operating cycle of the pattern matching machine
  - The machine starts in the state 0 and processes the first symbol of the sequence X
  - For any state s and input symbol α, the pattern matching machine may either make a goto transition (if $g[s, \alpha] = s$') or a failure transition (if $g[s, \alpha] = fail$)

The Aho-Corasick algorithm

- Processing of an input string (4)
  - In the case of a goto transition
    - If $o[s'] \neq \{\}$ the machine produces the output $o[s']$ together with the position of the current input symbol
  - In the case of a failure transition
    - The failure function value $f[s] = s'$ is evaluated and the machine repeats the cycle with s' as the current state and α as the current input symbol

The Aho-Corasick algorithm

- Processing of an input string (5)

Operation of the pattern matching machine

Input: An input string X and a pattern matching machine M with the goto function g, the failure function f and the output function o obtained by means of the algorithms 1 and 2

Output: Locations at which the patterns from Y occur in X

```
begin
    state ← 0;
    Let X = α1α2 ··· αn;
    for i ← 1 to n do
        while g[state, αi] = fail do
            state ← f[state]
        end
        state = g[state, αi];
        if o[state] ≠ {} then
            print i;
            print o[state];
        end
    end
end
```

The Aho-Corasick algorithm

- Processing of an input string (6)
  - Example: $X =$“snort on snow”
    - The operation of the pattern matching machine

Input	-	s	n	o	r	t		o	n		s	n	o	w
State	0	1	2	3	5	6	0	7	0	0	1	2	3	4
Output	-	-	-	-	or	snort	-	-	-	-	-	-	-	snow

Bit-parallelism and exact search

- Deterministic Finite Automaton (DFA)
  - A finite state machine (FSM) with a property that from each state, given an input symbol, there is only one outgoing branch to the next state

Bit-parallelism and exact search

- Nondeterministic Finite Automaton (NFA) (1)
  - A finite state machine capable of making transitions to more than one state for the same input symbol
  - One possible interpretation of such behavior
    - The machine makes copies of itself in such a situation – *parallel processing*
    - Each copy processes the subsequent input symbols independently

Bit-parallelism and exact search

- Nondeterministic Finite Automaton (NFA) (2)
  - If, after performing such copy making and following one of the paths, an input symbol arrives that does not appear as a label of any edge going out from the reached state, that machine-copy is stopped
    - It becomes *inactive*
  - If any of the copies of the machine reaches the final state, the input string is *accepted*, i.e. recognized

Bit-parallelism and exact search

- Any NFA can be transformed to a DFA
  - The general transformation algorithm has exponential complexity with respect to the length of the string determining the NFA
  - We consider a simple special case, where this transformation can be performed in polynomial time

Bit-parallelism and exact search

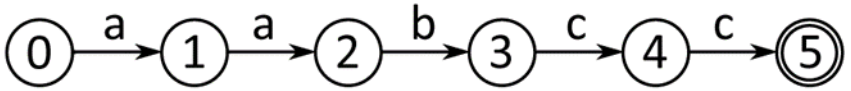
- NFA can be presented in two ways
  - With ε-transitions
  - Without ε-transitions
- ε-transitions
  - Transitions that do not consume any input
- In both cases (with or without ε-transitions), such an NFA recognizes all the *suffixes* of the corresponding search pattern

Bit-parallelism and exact search

- Without ε-transitions
  - In the state 0, the NFA expects the first symbol of the search pattern in the search string
  - We say that 0 is always an *active state*
  - If a machine is made inactive while at the state x, we say that x is an *inactive state*

Bit-parallelism and exact search

- Example – the search pattern is $w = \text{aabcc}$
  - The NFA without ε-transitions corresponding to w



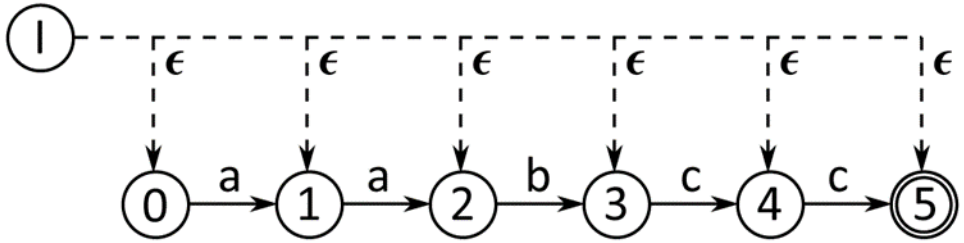
- Each input symbol (i.e. the next symbol from the search string) drives creation of a new copy of the NFA, which starts at the state 0

Bit-parallelism and exact search

- With ε-transitions (1)
  - After receiving an input *string*, an active state of such a machine is the one that corresponds to the end of some path (if it exists) starting from the initial state to that state

Bit-parallelism and exact search

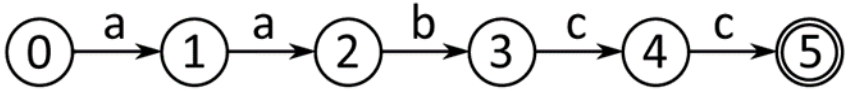
- With ε-transitions (2)
  - Example – the search pattern is $w = \text{aabcc}$
    - The NFA with ε-transitions corresponding to w



    - After the input string *a*, the active states will be 1 and 2
    - After receiving *aab*, the active state will be 3
    - After receiving *aba*, there will be no active states

Bit-parallelism and exact search

- Suppose now that the search string is $S =$ aaabcaabcc and we are searching for the pattern $w =$ aabcc in it by means of the NFA without ε-transitions



- Each time a symbol from S arrives, the machine makes a copy of itself and starts from the 0 state

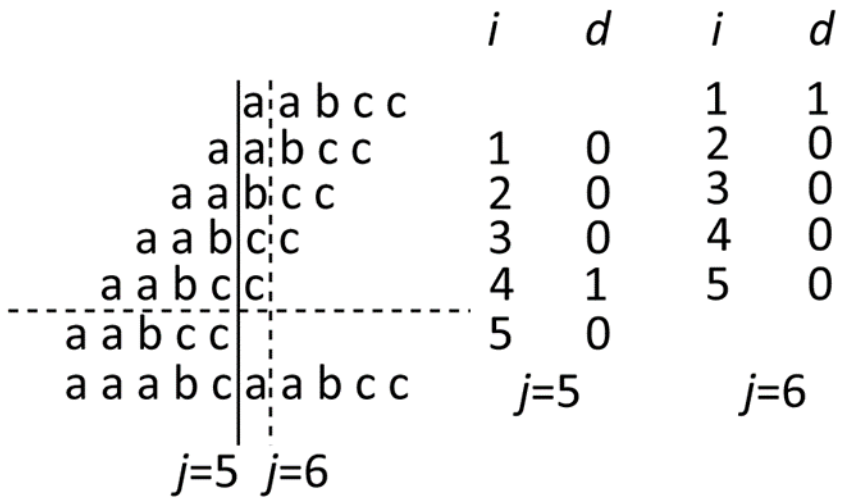
Bit-parallelism and exact search

- We are *simulating* the NFA
  - The maximum number of machines running in parallel is $m = |w|$, in our example $m = 5$
- Denote by j the number of processed symbols from S
  - Then, we have $\min(j, m)$ machines running in parallel, for each j
- After processing j symbols from S, some of these machines are active and some are inactive

Bit-parallelism and exact search

- Define *search status* in a computer word D of m bits
  - In our case $D = d_5 d_4 d_3 d_2 d_1$
- We set $d_i = 1$ if the machine i is active after processing j bits of S
- All the machines are active at the time of creation (since they are all in the state 0)

Bit-parallelism and exact search



Bit-parallelism and exact search

- Passing from $j = 5$ to $j = 6$, the bit d_5 disappears, since we can have maximum m machines at a time
  - This fact is expressed by *shifting* the word D one position to the left (d_4 becomes d_5, d_3 becomes d_4, etc.)
- At the same time, a new machine is created corresponding to the bit d_1, which starts from the state 0 (always active)
  - This fact is expressed by *OR-ing* the word D, shifted to the left, with $0^{m-1}1$ (in our case $m = 5$ so we OR with 00001)

Bit-parallelism and exact search

- In our example, when passing from $j = 5$ to $j = 6$, the next symbol from S to be processed is *a*
- Which input symbol will keep which machine active, if it was active before processing that symbol?
  - An *a* will always keep the machine d_1 active, an *a* will keep the machine d_2 active, a *b* will keep the machine d_3 active, a *c* will keep the machine d_4 active, a *c* will keep the machine d_5 active – it is always the same, can be *pre-computed*

Bit-parallelism and exact search

- We can use this for updating the search status word D automatically, after each shift left and OR-ing with 1, by introducing the pre-computed *bit masks*

Bit-parallelism and exact search

- The bit mask for any symbol only depends on the *search pattern*, not on the search string
- Because of that, we can pre-compute the bit masks for all the symbols from the pattern w
- Given the search pattern $w = w_1 w_2 \dots w_m$, for the bit mask $B[s] = b_1 b_2 \dots b_m$ the following holds
  - If $s = w_i$ then $b_{m+1-i} = 1$, otherwise $b_{m+1-i} = 0$

Bit-parallelism and exact search

- In our example, since $w =$ aabcc, it is easy to see that
  $B[a] = 00011$
  $B[b] = 00100$
  $B[c] = 11000$
- We can now update the search status word D for each new input symbol S_j in the following way

$$D_j = ((D_{j-1} << 1)\ OR\ 1)\ AND\ B[S_j]$$

Bit-parallelism and exact search

- For $j = 6$, we have

$$D_6 = ((D_5 << 1)\ OR\ 1)\ AND\ B[S_6] =$$
$$= ((01000 << 1)\ OR\ 00001)\ AND\ B[a] =$$
$$= 10001\ AND\ 00011 = 00001$$

Bit-parallelism and exact search

- The Shift-AND algorithm (Baeza-Yates, Gonnet, 1992)
  - The same formula for updating the search status word D

    $$D_j = ((D_{j-1} << 1)\ OR\ 1)\ AND\ B[S_j]$$

  - A match is reported if $d_m = 1$ (i.e. MSB=1), for some j
    - That would mean that the machine d_m has processed m symbols from S and is still active, i.e. it has reached the final state

Bit-parallelism and exact search

- Example (1)
  - w = origin, S = original
  - The bit masks

    $B[o] = 000001$

    $B[r] = 000010$

    $B[i] = 010100$

    $B[g] = 001000$

    $B[n] = 100000$
  - The search status word $D = 000000$

Bit-parallelism and exact search

- Example (2)
  - The first character from the search string

    $S[1] =$ ‘o’
  - The corresponding bit mask

    $B[S[1]] = B[o] = 000001$
  - The search status word after processing the character ‘o’

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[o]$

    $D \leftarrow ((000000 \ll 1)$ OR $000001)$ AND $000001 = 000001$

Bit-parallelism and exact search

- Example (3)
  - The second character from the search string

    $S[2] =$ ‘r’
  - The corresponding bit mask

    $B[S[2]] = B[r] = 000010$
  - The search status word after processing the character ‘r’

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[r]$

    $D \leftarrow ((000001 \ll 1)$ OR $000001)$ AND $000010 = 000010$

Bit-parallelism and exact search

- Example (4)
  - The third character from the search string

    $S[3] =$ 'i'
  - The corresponding bit mask

    $B[S[3]] = B[i] = 010100$
  - The search status word after processing the character 'i'

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[i]$

    $D \leftarrow ((000010 \ll 1)$ OR $000001)$ AND $010100 = 000100$

Bit-parallelism and exact search

- Example (5)
  - The fourth character from the search string

    $S[4] =$ ‘g’
  - The corresponding bit mask

    $B[S[4]] = B[g] = 001000$
  - The search status word after processing the character ‘g’

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[g]$

    $D \leftarrow ((000100 \ll 1)$ OR $000001)$ AND $001000 = 001000$

Bit-parallelism and exact search

- Example (6)
  - The fifth character from the search string

    $S[5] =$ 'i'
  - The corresponding bit mask

    $B[S[5]] = B[i] = 010100$
  - The search status word after processing the character 'i'

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[i]$

    $D \leftarrow ((001000 \ll 1)$ OR $000001)$ AND $010100 = 010000$

Bit-parallelism and exact search

- Example (7)
  - The sixth character from the search string

    $S[6] =$ ‘n’
  - The corresponding bit mask

    $B[S[6]] = B[n] = 100000$
  - The search status word after processing the character ‘n’

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[n]$

    $D \leftarrow ((010000 \ll 1)$ OR $000001)$ AND $100000 = 100000$
    - The MSB of D is 1 – search pattern w found at the position 6 in S

Bit-parallelism and exact search

- Example (8)
  - The seventh character from the search string

    $S[7] =$ ‘a’
  - The corresponding bit mask

    $B[S[7]] = B[*] = 000000$
  - The search status word after processing the character ‘a’

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[*]$

    $D \leftarrow ((100000 \ll 1)$ OR $000001)$ AND $000000 = 000000$

Bit-parallelism and exact search

- Example (9)
  - The eighth character from the search string

    $S[8] =$ 'l'
  - The corresponding bit mask

    $B[S[8]] = B[*] = 000000$
  - The search status word after processing the character 'l'

    $D \leftarrow ((D \ll 1)$ OR $000001)$ AND $B[*]$

    $D \leftarrow ((000000 \ll 1)$ OR $000001)$ AND $000000 = 000000$

Bit-parallelism and exact search

- The Shift-OR algorithm (1)
  - Very similar to Shift-AND
    - Often considered a special implementation of Shift-AND
    - Complements the bit masks for each symbol
    - Complements the search status word D
    - $d_i = 0$ means an active machine
  - In that case, OR-ing with 1 is not necessary
    - More efficient than Shift-AND – fewer logical operations after shifting the search status word

Bit-parallelism and exact search

- The Shift-OR algorithm (2)
  - The search status word update formula is

$$D_j = (D_{j-1} << 1)\ OR\ B[S_j]$$

  - A match is reported if $d_m = 0$ (i.e. MSB=0), for some j

Bit-parallelism and exact search

- Example (1)
  - w = origin, S = original
  - The bit masks

    $B[o] = 111110$

    $B[r] = 111101$

    $B[i] = 101011$

    $B[g] = 110111$

    $B[n] = 011111$
  - The search status word $D = 111111$

Bit-parallelism and exact search

- Example (2)
  - The first character from the search string

    $S[1] =$ ‘o’
  - The corresponding bit mask

    $B[S[1]] = B[o] = 111110$
  - The search status word after processing the character ‘o’

    $D \leftarrow (D \ll 1)$ OR $B[o]$

    $D \leftarrow (111111 \ll 1)$ OR $111110 = 111110$

Bit-parallelism and exact search

- Example (3)
  - The second character from the search string

    $S[2] =$ ‘r’
  - The corresponding bit mask

    $B[S[2]] = B[r] = 111101$
  - The search status word after processing the character ‘r’

    $D \leftarrow (D \ll 1)$ OR $B[r]$

    $D \leftarrow (111110 \ll 1)$ OR $111101 = 111101$

Bit-parallelism and exact search

- Example (4)
  - The third character from the search string

    $S[3] =$ ‘i’
  - The corresponding bit mask

    $B[S[3]] = B[i] = 101011$
  - The search status word after processing the character ‘i’

    $D \leftarrow (D \ll 1)$ OR $B[i]$

    $D \leftarrow (111101 \ll 1)$ OR $101011 = 111011$

Bit-parallelism and exact search

- Example (5)
  - The fourth character from the search string

    $S[4] =$ ‘g’
  - The corresponding bit mask

    $B[S[4]] = B[g] = 110111$
  - The search status word after processing the character ‘g’

    $D \leftarrow (D \ll 1)$ OR $B[g]$

    $D \leftarrow (111011 \ll 1)$ OR $110111 = 110111$

Bit-parallelism and exact search

- Example (6)
  - The fifth character from the search string

    $S[5] =$ ‘i’
  - The corresponding bit mask

    $B[S[5]] = B[i] = 101011$
  - The search status word after processing the character ‘i’

    $D \leftarrow (D \ll 1)$ OR $B[i]$

    $D \leftarrow (110111 \ll 1)$ OR $101011 = 101111$

Bit-parallelism and exact search

- Example (7)
  - The sixth character from the search string

    $S[6] =$ ‘n’
  - The corresponding bit mask

    $B[S[6]] = B[n] = 011111$
  - The search status word after processing the character ‘n’

    $D \leftarrow (D \ll 1)$ OR $B[n]$

    $D \leftarrow (101111 \ll 1)$ OR $011111 = 011111$
    - The MSB of D is 0 – search pattern w found at the position 6 in S

Bit-parallelism and exact search

- Example (8)
  - The seventh character from the search string

    $S[7] =$ ‘a’
  - The corresponding bit mask

    $B[S[7]] = B[*] = 111111$
  - The search status word after processing the character ‘a’

    $D \leftarrow (D \ll 1)$ OR $B[*]$

    $D \leftarrow (011111 \ll 1)$ OR $111111 = 111111$

Bit-parallelism and exact search

- Example (9)
  - The eighth character from the search string

    $S[8] =$ 'l'
  - The corresponding bit mask

    $B[S[8]] = B[*] = 111111$
  - The search status word after processing the character 'l'

    $D \leftarrow (D \ll 1)$ OR $B[*]$

    $D \leftarrow (111111 \ll 1)$ OR $111111 = 111111$

Bit-parallelism and exact search

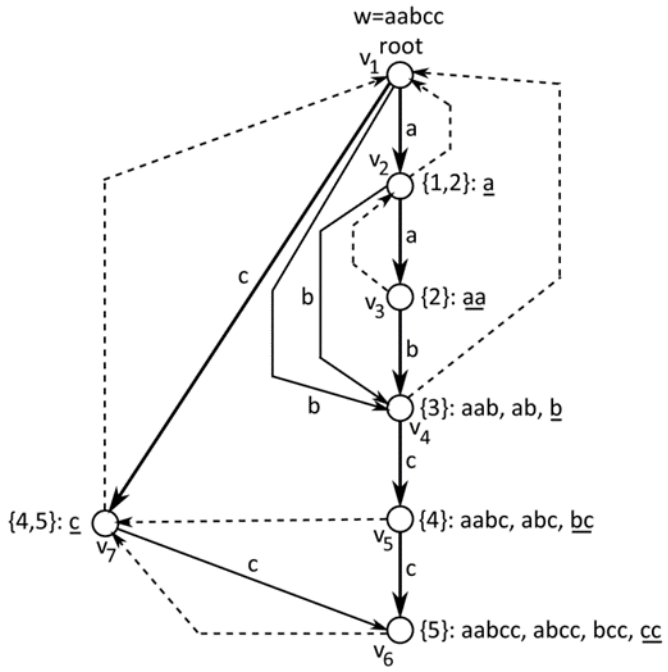
- The worst-case and the average-case time complexities of the algorithm Shift-AND / Shift-OR are the same
  - Resistant to algorithmic attacks
  - Slower on average than some other algorithms from the category of bit-parallel search algorithms (e.g. BNDM, BNDMq)

Bit-parallelism and exact search

- BNDM (Backward Non-deterministic DAWG Matching)
- DAWG – Directed Acyclic Word Graph
  - A graph that can be assigned to any string
  - It can be shown that a DAWG is equivalent to the DFA corresponding to the NFA assigned to the same string
    - DAWG is sometimes called *suffix automaton*

Bit-parallelism and exact search

- Example – the DAWG corresponding to w = aabcc



Bit-parallelism and exact search

- BNDMq – a small modification of BNDM
  - Faster than BNDM since it processes more (q) input characters at a time
- BNDM and BNDMq are so-called *skip algorithms*
  - Average-case time complexity better than the worst-case time complexity
  - Consequently, sensitive to algorithmic attacks
    - Not good for application in IDS
    - Tried in Suricata and later abandoned

Approximate search in IDS

- Most misuse-based IDS employ *exact search*
  - Fast
  - Reliable
  - However, useless for detection of zero-day attacks
    - A single bit of change is enough to evade such an IDS

Approximate search in IDS

- Example – A Snort rule

  alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (msg: "NETBIOS SMB CD.."; flow: to_server,established; *content*: "|5C|../|00 00 00|";)

  - The *content* field defines the search pattern
    - In this case "|5C|../|00 00 00|"
    - If attack traffic is changed to contain "|5C|../|20 00 00|"
      - Only one bit changed
      - Same attack, but not detected with this rule

Approximate search in IDS

- Possible solutions to this problem
  - Make a new signature for every variant of an old attack
    - Current situation in IDS
    - Inefficient, impossible for zero-day attacks
  - Use *approximate search*
    - Problem – false positives and false negatives due to not taking into account the fact that only small (limited) changes with specific distribution on the old attack patterns are possible
      - If big changes, traffic could become harmless – no attack
      - For taking this into account – *constrained approximate search*

Approximate search in IDS

- Definition – *approximate matching* (1)
  - String matching allowing up to k errors (substituted, inserted or deleted characters)
  - Edit distance (Levenshtein 1965)
    - Minimum number of elementary edit operations (substitutions, deletions, and insertions) needed to transform one string into another

Approximate search in IDS

- Definition – *approximate matching* (2)
  - Then, approximate matching is reduced to finding all occurrences of the distorted search pattern w' in the search string S such that the edit distance $ed(w, w') \leq k$
  - The distortion of w' is carried out by deleting, inserting, or substituting characters

Approximate search in IDS

- Definition – *approximate matching* (3)
  - Substitution of a character by itself (a *match*) does not usually contribute to the increase of edit distance
    - In most applications (except in computational biology), it is enough to assign the elementary distance equal to 1 to an ordinary substitution and 0 to a character match

Approximate search in IDS

- Example – edit distance

  $X =$ surgery, $Y =$ survey, $ed(X, Y) = 2$

  - Edit sequence

s	u	r	g	e	r	y
s	u	r	v	e	ϕ	y

  - ϕ is called the *empty symbol*
    - Used for presenting deletions and insertions

Approximate search in IDS

- Example – approximate matching

  $w =$ surv, $S =$ xxxxxsurgery, $k = 2$

  - Edit sequence ($w' =$ surg, $ed(w', w) = 1 < k$)

x	x	x	x	x	s	u	r	g	e	r	y
ϕ	ϕ	ϕ	ϕ	ϕ	s	u	r	v	ϕ	ϕ	ϕ

  - The elementary edit operations to the left of the first vertical line and to the right of the second vertical line do not count
    - We are looking for the pattern anywhere in the search string

Approximate search in IDS

- Two approaches to approximate matching
  - Dynamic programming
  - Simulation of an NFA (bit-parallelism)

Approximate search in IDS

- Dynamic programming approach (1)
  - A matrix of *partial edit distances* is used
  - The cell $[i, j]$ contains edit distance between the prefix of the string X of length i and the prefix of the string Y of length j
  - The value in a cell is computed on the basis of the previously computed values in other cells

Approximate search in IDS

- Dynamic programming approach (2)
  - Computation of edit distance between X and Y

$$d_{i,j} = \begin{cases} d_{i-1,j-1}, & x_i = y_j \\ 1 + \min\{d_{i-1,j-1}, d_{i-1,j}, d_{i,j-1}\}, & \text{otherwise} \end{cases}$$

Approximate search in IDS

- Dynamic programming approach (3)
  - Example, $X =$ annual, $Y =$ annealing, $ed(X, Y) = 4$

		a	n	n	e	a	l	i	n	g
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
n	2	1	0	1	2	3	4	5	6	7
n	3	2	1	0	1	2	3	4	5	6
u	4	3	2	1	1	2	3	4	5	6
a	5	4	3	2	2	1	2	3	4	5
l	6	5	4	3	3	2	1	2	3	4

a n n ø | u	a n n e | a	$d = 2$
a n n ø ø | u	a n n e a | ø	$d = 3$
a n n u | ø	a n n e | a	$d = 2$

Approximate search in IDS

- Dynamic programming approach (4)
  - Approximate search
    - We use the same computation formula for edit distance

$$d_{i,j} = \begin{cases} d_{i-1,j-1}, & x_i = y_j \\ 1 + \min\{d_{i-1,j-1}, d_{i-1,j}, d_{i,j-1}\}, & \text{otherwise} \end{cases}$$

    - The *initialization* is different
      - We fill the first row with zeros, which reflects the fact that the computation of the edit distance can start at any position in the search string

Approximate search in IDS

- Dynamic programming approach (5)
  - $w =$ annual, $S =$ annealing, $k = 2$

		a	n	n	e	a	l	i	n	g
	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	1	0	1	1	1	1
n	2	1	0	1	2	1	1	2	1	2
n	3	2	1	0	1	2	2	2	2	2
u	4	3	2	1	1	2	3	3	3	3
a	5	4	3	2	2	1	2	3	4	4
l	6	5	4	3	3	**2**	**1**	**2**	3	4

Approximate search in IDS

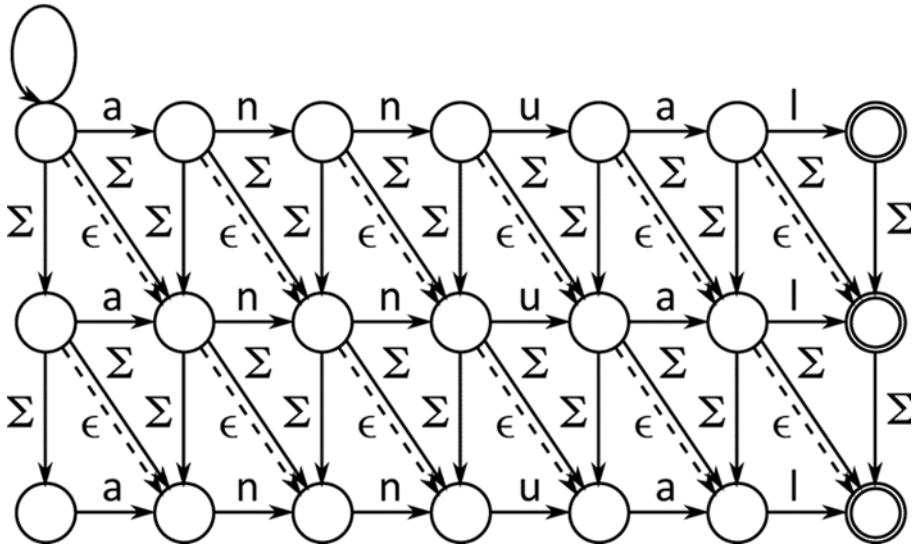
- Simulation of an NFA (bit-parallelism) (1)
  - The NFA (1)
    - Has a matrix form (2-dimensional)
    - The state (0,0) has a self-loop
      - Reflects the fact that the search pattern can start at any position in the search string
    - Horizontal transition means a *match* (we advance 1 character in the search pattern and in the search string)
    - Vertical transition means an *insertion* (we advance 1 character in the search string but we do not advance in the search pattern)

Approximate search in IDS

- Simulation of an NFA (bit-parallelism) (2)
  - The NFA (2)
    - Diagonal transition with a solid line means an *ordinary substitution* (we advance 1 character in the search pattern and in the search string)
    - Diagonal transition with a dashed line means a *deletion* (we advance 1 character in the search pattern but we do not advance in the search string)
      - Dashed lines represent *ε transitions*

Approximate search in IDS

- Simulation of an NFA (bit-parallelism) (3)
  - Example, $w =$ annual, $S =$ anneal (1)



Approximate search in IDS

- Two groups of matching algorithms
  - Row-based Bit-Parallelism (RBP)
  - Diagonal-based Bit-Parallelism (DBP)

Approximate search in IDS

- The Row-based Bit-Parallelism (RBP) algorithm (1)
  - Wu, Manber, 1992
  - Unconstrained approximate search allowing up to k errors
  - The algorithm updates the matrix associated with the search pattern, row by row, after processing each character of the search string
  - If a final state becomes active in any row, an *occurrence* is reported

Approximate search in IDS

- The Row-based Bit-Parallelism (RBP) algorithm (2)
  - We only consider the Shift-OR variant (more efficient)
  - The row-based matrix update formula (R_i denotes the i-th row of the matrix, j is the index of the current processed character from the search string S, $i = 1, \ldots, k$ is the number of errors)

Approximate search in IDS

- The Row-based Bit-Parallelism (RBP) algorithm (3)

$$R'_0 = (R_0 \ll 1)\ OR\ B[S_j]$$ No errors

$$R'_i = \left((R_i \ll 1)\ OR\ B[S_j]\right)\ AND$$ *match*

$$R_{i-1}\ AND$$ *ins*

$$\left((R_{i-1} \ll 1)\ OR\ NOT\ B[S_j]\right)\ AND$$ *sub*

$$(R'_{i-1} \ll 1)$$ *del*

Initialization: R_0 all zeros, R_1 one 1 rightmost, R_2 two 1s rightmost, …, R_k k 1s rightmost

Approximate search in IDS

- The attack scenario (1)
  - To modify the old attack pattern, the attacker uses a tool (1)
    - Cannot delete more than *e* characters in total
    - Cannot insert more than *i* characters in total
    - Cannot substitute more than *s* characters in total

Approximate search in IDS

- The attack scenario (2)
  - To modify the old attack pattern, the attacker uses a tool (2)

    Or

    - Cannot have more than *d* indels (insertions+deletions) in total

Approximate search in IDS

- The attack scenario (3)
  - To modify the old attack pattern, the attacker uses a tool (3)

    Or

    - Cannot delete more than *E* characters *at a time*
    - Cannot insert more than *I* characters *at a time*

Approximate search in IDS

- With such an attack scenario
  - Small changes in original attack traffic
    - If big changes, traffic could become harmless
    - Also distribution of changes matters (e.g. $< E$ deletions at a time)
  - Unconstrained approximate search cares only about the number of changes
    - Consequence – *false positives*

Approximate search in IDS

- Constrained approximate search (1)
  - To reduce false positive rate, introduce *constraints*
    - Related to a-priori knowledge about the parameters of the traffic modification tool used by the attacker
      - If guessed well, reduces the false positive rate
    - Realization in the search algorithm – depends on constraint type
      - Introduce various counters and additional bit masks
      - Small changes in the original traffic –> small overhead
      - Counter assigned to each cell of the NFA array

Approximate search in IDS

- Constrained approximate search (2)
  - Constrained string editing (1)
    - Dynamic programming (Wagner, Fisher, 1974) – original algorithm
      - Quadratic time/space complexity
      - Example
        - Transform the string X of length n to the string Y of length m
        - Elementary edit operations - substitutions, insertions, deletions
        - Then, the prefix X_i is transformed to the prefix Y_j
        - $i = 1, \dots, n, j = 1, \dots, m$ are the coordinates
        - Complicated to introduce various constraints

Approximate search in IDS

- Constrained approximate search (3)
  - Constrained string editing (2)
    - The formula with the coordinates i, j (Wagner, Fisher, 1974)

$$d_{i,j} = \begin{cases} d_{i-1,j-1}, & x_i = y_j \\ 1 + \min\{d_{i-1,j-1}, d_{i-1,j}, d_{i,j-1}\}, & \text{otherwise} \end{cases}$$

    - Alternative formulation

$$d_{i,j} = \min\{d_{i-1,j-1} + d_s, d_{i-1,j} + 1, d_{i,j-1} + 1\},$$

      where $d_s = 0$ if $x_i = y_j$, otherwise $d_s = 1$

    - *Unconstrained*

Approximate search in IDS

- Constrained approximate search (4)
  - Constrained string editing (3)
    - Changing the coordinates (Oommen, 1986)
      - The new coordinates are the numbers of elementary edit operations, not the prefix lengths
      - The number of insertions i, the number of substitutions s, the number of deletions e
      - Three-dimensional dynamic programming array
        - More complicated than the original algorithm (two-dimensional)
      - But, easier to define various constraints

Approximate search in IDS

- Constrained approximate search (5)
  - Constrained string editing (4)
    - With the i, e, s coordinates – *constrained by default*
    - The prefix X_{e+s} is transformed to the prefix Y_{i+s}

$$W[i,e,s] = min \begin{cases} W[i,e,s-1] + d(X_{e+s}, Y_{i+s}), \\ W[i-1,e,s] + d(\phi, Y_{i+s}), \\ W[i,e-1,s] + d(X_{e+s}, \phi) \end{cases}$$

      - ϕ is the empty symbol
      - $d(X_{e+s}, Y_{i+s})$ is the substitution cost, $d(\phi, Y_{i+s})$ is the insertion cost, $d(X_{e+s}, \phi)$ is the deletion cost

Approximate search in IDS

- Constrained approximate search (6)
  - Constrained string editing (5)
    - Constraints – it can be shown that the constraints on the number of deletions and substitutions are related with the constraint on the number of insertions (Oommen, 1986)
    - Let N be the length of X and let M be the length of Y
    - Then, given the number of insertions i
      - The number of deletions must be $N - M + i$
      - The number of substitutions must be $M - i$

Approximate search in IDS

- Constrained approximate search (7)
  - Constrained string editing (6)
    - Thus, given the set $\mathcal{I}$ of the permitted values for i, the constrained edit distance between X and Y is given by

      $$D(X,Y) = \min_{i \in \mathcal{I}} W[i, N-M+i, M-i]$$

    - $\mathcal{I}$ is a subset of the set

      $$\{\max(0, M-N), \ldots, M\}$$

    - If $\mathcal{I} = \{\max(0, M-N), \ldots, M\}$ exactly, then we get the *unconstrained* edit distance

Approximate search in IDS

- Constrained approximate search (8)
  - Constraints on the run lengths of insertions/deletions (1)
    - Cannot delete more than E symbols at a time
    - Cannot insert more than I symbols at a time
    - With the coordinates i, e, s
      - Much easier to introduce these types of constraints

Approximate search in IDS

- Constrained approximate search (9)
  - Constraints on the run lengths of insertions/deletions (2)
    - The formula for computing the elements of the constrained edit distance array W

$$W[i,e,s] = \min\{W[i-i_1, e-e_1, s-1] + e_1 d_E + i_1 d_I + d(x_{e+s}, y_{i+s})\}$$

where

$$\max\{0, e - \min\{n-s, (s-1)E\}\} \leq e_1 \leq \min\{e, E\}$$

$$\max\{0, i - \min\{m-s, (s-1)I\}\} \leq i_1 \leq \min\{i, I\}$$

$$s = 1, \ldots s_{\max},\ e = 0, \ldots \min\{n-s, sE\},\ i = 0, \ldots, \min\{m-s, sI\}$$

Approximate search in IDS

- Constrained approximate search (9)
  - The bit-parallel RBP approach also possible
  - Search status array update formula for any value of E and I not ready yet
    - A special case is ready
      - No insertions
      - $E = 1$
      - Used in cryptanalysis of pseudorandom generator schemes employing irregularly clocked Linear Feedback Shift Registers (LFSRs)

Approximate search in IDS

- Constrained approximate search (10)
  - The search status word update formula, $E = 1$

$$dm_i = 1^m, i = 1, \ldots, k$$

$$R_0' = (R_0 << 1) \text{ OR } B[S_j]$$

$$del = 1^m$$

$$R_i' = \text{match AND sub AND del}$$

$$\begin{aligned} \text{match} &= (R_i << 1) \text{ OR } B[S_j] \\ \text{sub} &= ((R_{i-1} << 1) \text{ OR NOT } B[S_j]) \\ \text{del} &= (R_{i-1} << 1) \text{ OR} \\ &\quad \text{NOT } ((\text{del} << 1) \text{ OR } 0^{m-1}1) \text{ OR} \\ &\quad \text{NOT } ((\text{dm}_{i-1} << 1) \text{ OR } 0^{m-1}1) \text{ OR} \\ &\quad (0^{m-1}1 << (m-1)) \end{aligned}$$

$$dm_i' = \text{del}$$

$$i = 1, \ldots, k$$