Studi tentang Trade-off Antara Memori dan Kecepatan pada Varian Algoritma Aho-Corasick dalam Sistem Deteksi Intrusi

Richard Samuel Simanullang - 13524112
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: samsr472@gmail.com , 13524112@std.stei.itb.ac.id

***Abstrak*—Algoritma Aho-Corasick (AC) telah menjadi standar de-facto untuk pencocokan multi-pola pada Network Intrusion Detection System (NIDS) karena waktu pencariannya linear terhadap panjang masukan dan independen terhadap jumlah pola. Namun, AC dikenal boros memori akibat ledakan aturan transisi (transition rule explosion) pada deterministic finite automaton (DFA)-nya. Makalah ini berargumen bahwa seluruh varian AC sebenarnya memakai automaton yang sama; yang berbeda hanyalah cara menyimpan fungsi transisi δ(state, karakter). Karena itu, keberadaan varian merupakan manifestasi langsung dari trade-off ruang–waktu klasik. Untuk membuktikan tesis ini secara kuantitatif, penulis mengimplementasikan satu automaton AC dan menyimpan fungsi transisinya dalam tiga representasi: matriks penuh (DFA), peta jarang (sparse), dan NFA biner padat (BNFA). Pada himpunan 217 signature sintetis (1.535 state) dan payload 2 MB, representasi matriks penuh menempati 1,57 MB dengan tepat satu operasi transisi per karakter, sedangkan representasi jarang hanya 13,8 KB tetapi memerlukan 1,265 operasi per karakter disertai penelusuran failure link. Rasio memori mencapai ≈114×. Hasil ini selaras dengan temuan empiris pada Snort dan menegaskan bahwa pemilihan varian AC pada hakikatnya adalah pemilihan titik pada spektrum ruang–waktu.**

Kata kunci—Aho-Corasick; pencocokan multi-pola; sistem deteksi intrusi; trade-off ruang–waktu; ledakan aturan transisi; automaton.

I. PENDAHULUAN

Volume trafik jaringan dan jumlah signature serangan terus tumbuh seiring meluasnya konektivitas. Sebuah NIDS modern harus memeriksa setiap byte muatan paket terhadap basis aturan yang sangat besar; Snort versi 2.9, misalnya, memuat lebih dari 9.000 pola yang harus dicocokkan terhadap lalu lintas yang masuk [8]. Pemeriksaan pencocokan pola pada muatan paket ini yang menjadi titik sempit (bottleneck) kinerja NIDS. Studi empiris melaporkan bahwa pencocokan string dapat menghabiskan 40–70% dari total waktu pemrosesan Snort [3], sehingga pencocokan pola merupakan uji yang paling mahal secara komputasi pada perangkat ini [8].

Karena jumlah pola mencapai ribuan, pendekatan naif yang menjalankan algoritma pencocokan seperti Knuth-Morris-Pratt atau Boyer-Moore secara berulang untuk setiap pola tidaklah praktis: kompleksitasnya tumbuh sebanding dengan hasil kali panjang teks dan jumlah pola. Yang dibutuhkan adalah algoritma pencocokan multi-pola yang waktunya tidak bergantung pada banyaknya pola. Algoritma Aho-Corasick (AC) [1] memenuhi kebutuhan ini dan telah menjadi standar de-facto pada NIDS berbasis signature.

Keunggulan AC bersumber dari dua sifatnya: waktu pencarian linear terhadap panjang masukan dan independen terhadap jumlah pola [1]. Namun keunggulan ini ditebus dengan kelemahan yang serius pada sisi memori. Biaya utama AC adalah penyimpanan aturan transisi pada DFA yang mendasarinya [6]. Ketika fungsi transisi disimpan sebagai matriks penuh agar setiap langkah memerlukan tepat satu pembacaan tabel, jumlah aturan transisi bertambah drastis, fenomena ini dikenal sebagai transition rule explosion [7].

Untuk mengatasinya, berbagai varian AC dikembangkan. Pengamatan inti makalah ini adalah bahwa varian-varian tersebut tidak mengubah automaton: semuanya beroperasi pada automaton AC yang sama, dengan goto, failure, dan output yang identik. Yang berbeda semata-mata adalah cara menyimpan fungsi transisi δ. Oleh karena itu, keberadaan varian adalah perwujudan langsung dari trade-off ruang–waktu: menukar memori dengan kecepatan, atau sebaliknya.

A. *Rumusan Masalah*

Permasalahan yang dikaji adalah: (1) mengapa varian-varian Aho-Corasick dapat dipandang sebagai satu automaton dengan banyak representasi tabel transisi; (2) bagaimana setiap representasi memosisikan diri pada spektrum ruang–waktu; dan (3) seberapa besar trade-off memori–kecepatan tersebut secara kuantitatif bila diukur langsung melalui implementasi.

B. *Tujuan*

Tujuan makalah ini adalah menganalisis trade-off memori–kecepatan antarvarian Aho-Corasick dari sudut pandang strategi algoritma dan struktur data, serta membuktikannya secara empiris melalui implementasi mandiri atas tiga representasi fungsi transisi dan pengukuran memori serta jumlah operasi transisi yang nyata

C. *Ruang Lingkup*

Pembahasan dibatasi pada analisis algoritmik dan struktur data dari fungsi transisi AC; makalah ini bukan tutorial konfigurasi Snort. Implementasi pengujian dilakukan pada perangkat lunak (Python) dengan himpunan pola dan payload sintetis berskala laboratorium, sementara validasi dilakukan dengan membandingkan kecenderungan hasil terhadap studi empiris pada Snort.

D. *Sistematika*

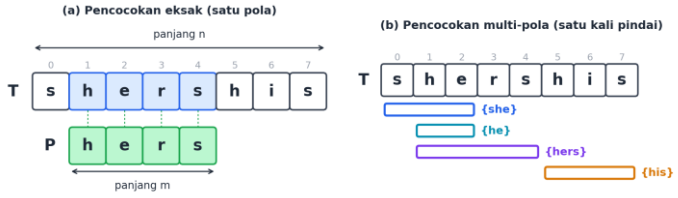
Bab II memaparkan landasan teori pencocokan pola dan algoritma AC. Bab III sebagai bab inti menyajikan kerangka "satu automaton, banyak representasi" beserta posisi tiap varian pada spektrum ruang-waktu. Bab IV menyajikan implementasi dan hasil pengukuran. Bab V kesimpulan dan saran.

II. Landasan Teori

A. *Pencocokan String (String/Pattern Matching)*

Persoalan pencocokan string (string matching) didefinisikan sebagai berikut: diberikan sebuah teks T berupa string panjang berukuran n karakter dan sebuah pola (pattern) P berukuran m karakter dengan m jauh lebih kecil daripada n, carilah lokasi kemunculan P di dalam T [11]. Bila hanya satu pola yang dicari, persoalan ini disebut pencocokan satu-pola; pencocokan eksak menuntut kecocokan karakter demi karakter secara persis, seperti diilustrasikan pada Gambar 1(a) [11]. Pencocokan multi-pola (multi-pattern matching) memperumum persoalan menjadi pencarian seluruh kemunculan dari sebuah himpunan pola $\{P_1, P_2, \ldots, P_k\}$ secara serentak hanya dalam satu kali pemindaian teks, seperti pada Gambar 1(b). Pada konteks NIDS, himpunan pola adalah kumpulan signature serangan dan teks adalah muatan paket, sehingga algoritma yang efisien harus menemukan semua signature dalam sekali jalan, bukan memindai teks berulang kali untuk setiap pola.

Algoritma satu-pola seperti Knuth-Morris-Pratt (KMP) menyelesaikan kasus k = 1 dalam waktu linear dengan memanfaatkan *fungsi failure* yang menghindari pengunduran (backtracking) pada teks [2]. Untuk k pola, menjalankan KMP berulang menghasilkan kompleksitas yang bergantung pada k, sehingga tidak skalabel terhadap basis aturan berukuran ribuan.



Gambar I. Persoalan pencocokan string: (a) pencocokan eksak satu pola P di dalam teks T (panjang n dan m); (b) pencocokan multi-pola yang menemukan seluruh pola {he, she, his, hers} dalam satu kali pemindaian.

B. *Fungsi Failure sebagai Jembatan*

Aho-Corasick dapat dipandang sebagai generalisasi KMP dari satu pola menjadi banyak pola. Bila KMP memakai fungsi failure pada sebuah string tunggal, AC memakai failure link pada sebuah trie yang memuat seluruh pola [1]. Gagasan failure yang sama, "ke mana harus melompat ketika karakter berikutnya tidak cocok", menjadi konsep pemikiran yang sama dari KMP ke AC.

C. *Algoritma Aho-Corasick*

Automaton AC dibangun dari tiga fungsi [1]. Pertama, fungsi goto g(s, c) membentuk sebuah trie dari seluruh pola: dari state s, karakter c menuntun ke state anak bila ada cabang berlabel c. Kedua, fungsi failure f(s) dihitung melalui penelusuran lebar (BFS) dan menunjuk ke state terpanjang yang merupakan sufiks sejati dari string yang membawa ke s. Ketiga, fungsi output mencatat pola apa saja yang berakhir pada s (termasuk yang diwarisi melalui failure).

Pada saat pencarian, automaton membaca teks satu karakter setiap langkah. Bila goto pada state kini tidak terdefinisi untuk karakter masukan, automaton mengikuti failure link berulang hingga menemukan goto yang sesuai atau kembali ke akar. Setiap kali state output dimasuki, seluruh pola yang tercatat padanya dilaporkan sebagai kecocokan.

D. *Sifat Linear dan Analisis Kompleksitas*

Konstruksi automaton memerlukan waktu sebanding dengan total panjang seluruh pola, yakni $O(\Sigma|P_i|)$ [1]. Pada tahap pencarian, jumlah transisi state yang dilakukan automaton dalam memproses teks bersifat independen terhadap jumlah pola [1]; pencarian berjalan dalam O(n + z), dengan n panjang teks dan z banyaknya kecocokan yang dilaporkan. Inilah sumber keunggulan AC: berapa pun banyak signature yang dimuat, biaya pemindaian per karakter tetap terkendali.

E. *Representasi Fungsi Transisi*

Sifat linear di atas mengasumsikan transisi δ(s, c) dapat diperoleh dalam waktu konstan. Cara termudah mewujudkannya adalah mengubah goto+failure menjadi DFA penuh dan menyimpan δ sebagai matriks: untuk setiap state, sebuah baris berisi |Σ| entri (256 untuk alfabet byte). Namun penyimpanan setiap state sebagai larik 256 penunjuk sangat boros [4]. Terdapat variasi besar pada banyaknya transisi yang benar-benar terpakai: state dekat akar membutuhkan lebih dari 200 penunjuk, sementara state dekat daun hanya 1–2 [4]. Sebagian besar sel matriks, dengan demikian, hanyalah hasil pelipatan failure yang berulang.

Persoalan representasi fungsi transisi inilah yang menjadi pusat seluruh perdebatan kinerja AC [5]. Nieminen dan Kilpeläinen membandingkan delapan representasi yaitu larik, senarai berantai, hashing, pohon seimbang, perfect hashing, hibrida, triple-array, dan double-array serta menunjukkan bahwa praktik unjuk kerja AC sangat ditentukan oleh struktur data yang dipilih untuk menyimpan δ [5]. Dengan kata lain, automaton-nya tetap, representasinyalah yang menentukan posisi pada spektrum ruang–waktu.

F. *Konteks Sistem Deteksi Intrusi (IDS)*

The Sistem Deteksi Intrusi (Intrusion Detection System, IDS) adalah perangkat lunak atau perangkat keras yang memantau aktivitas pada jaringan atau host untuk mengenali tanda-tanda serangan maupun pelanggaran kebijakan keamanan [10]. Berdasarkan cakupan pemantauannya, IDS lazim dibedakan menjadi berbasis jaringan (network-based IDS, NIDS) yang memeriksa lalu lintas paket, dan berbasis host (host-based IDS, HIDS) yang memeriksa aktivitas pada satu mesin [10]. Berdasarkan metode deteksinya, terdapat dua pendekatan utama: deteksi berbasis anomali, yang membangun profil perilaku "normal" lalu menandai penyimpangan terhadapnya, dan deteksi berbasis signature, yang membandingkan kejadian teramati terhadap pustaka pola serangan yang telah dikenal [10]. Sebuah signature pada dasarnya adalah pola yang berkorespondensi dengan ancaman yang diketahui [10]. Pendekatan berbasis signature sangat akurat terhadap ancaman yang sudah dikenal, tetapi secara arsitektural tidak mampu mengenali serangan zero-day yang belum memiliki signature [10].

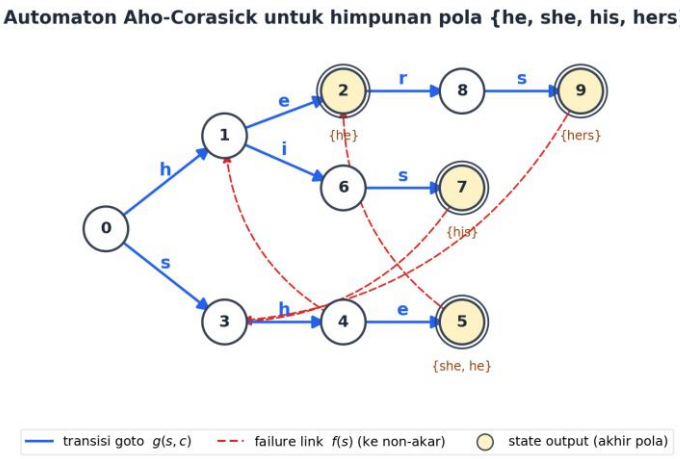
Di sinilah pencocokan pola menjadi inti operasi NIDS berbasis signature: mendeteksi serangan berarti mencari kemunculan ribuan signature pada muatan setiap paket secara serentak. Snort, NIDS sumber terbuka yang banyak dipakai, menyandarkan deteksinya pada mesin pencarian multi-pola Aho-Corasick—ia memindai lalu lintas dengan mencari nilai tertentu pada header serta pola yang dikenal pada data lapisan aplikasi [9]. Konsekuensinya, kebutuhan akan memori yang ramping menjadi krusial karena automaton yang dihasilkan sangat besar. Himpunan baku Snort yang memuat lebih dari 1.000 string menghasilkan automaton AC dengan sekitar 10.000 state [4]. Bila setiap state menyimpan matriks 256 entri, kebutuhan memori melonjak hingga orde megabyte, beban berdampak langsung pada lokalitas cache dan, pada akhirnya, throughput pemindaian. Karena itu seluruh varian AC berlomba menekan biaya representasi δ tanpa mengorbankan terlalu banyak kecepatan.

III. Varian Aho-Corasick dan Strategi Trade-Off

Bab ini merupakan inti dari makalah. Kerangka analisis yang dipakai terdiri atas 1 automaton Aho-Corasick, tetapi banyak cara menyimpan tabel transisinya, dan setiap cara menempati titik berbeda pada spektrum ruang-waktu.

A. *Satu Automaton, Banyak Representasi*

Ilustrasi klasik pada Gambar 1 memperlihatkan automaton AC untuk himpunan pola {he, she, his, hers} [1]. Panah penuh menyatakan transisi goto, panah putus-putus menyatakan failure link (yang menuju selain akar), dan state berlingkar ganda adalah state output. Struktur inilah—goto, failure, output—yang dipakai bersama oleh semua varian. Perhatikan bahwa state 5 (akhir pola "she") sekaligus melaporkan "he" karena failure-nya menunjuk ke state 2; warisan output semacam ini adalah bagian dari automaton dan tidak berubah antarrepresentasi.



Gambar II. Automaton Aho-Corasick untuk himpunan pola {he, she, his, hers}. Goto (penuh), failure (putus-putus), dan state output (lingkaran ganda) dipakai Bersama oleh semua varian.

Yang berubah hanyalah bagaimana δ(s, c) disimpan dan dievaluasi. Pada salah satu ujung spektrum, kita dapat menghitung penuh DFA dan menyimpan δ sebagai matriks 256 kolom per state. Evaluasi transisi menjadi satu pembacaan larik tetapi memori membengkak. Pada ujung lain, kita menyimpan hanya goto eksplisit (yang jumlahnya sebanding dengan jumlah state) ditambah failure link, lalu mengevaluasi δ dengan menelusuri failure saat goto kosong. Memori menjadi minimal, tetapi setiap karakter mungkin memerlukan lebih dari satu langkah. Sebagaimana ditegaskan Norton, justru pilihan representasi tabel state inilah yang menentukan trade-off memori dan kinerja sebuah varian [9].

B. *Posisi Varian pada Spektrum*

Snort menyediakan beberapa metode pencarian yang seluruhnya berbasis automaton AC yang sama, namun berbeda representasi tabel transisinya [8], [9]. Tabel I merangkum posisinya pada spektrum ruang–waktu, dan subbab berikut menguraikan tiap varian satu per satu menurut cara penyimpanan δ-nya.

Tabel I. Posisi Varian Aho-Corasick Snort pada spektrum ruang-waktu

Varian (Snort)	Representasi δ	Memori	Kecepatan
ac/ac-q (full)	Matriks penuh (DFA)	Tinggi	Terbaik
ac-bnfa	NFA biner (sparse, padat)	Rendah	Tinggi
ac-split	Pola dibagi menjadi blok	Sedang	Tinggi
ac-banded	Matriks banded	Kecil	Sedang
ac-sparsebands	Matriks sparse-banded	Kecil	Sedang–tinggi

lowmem	Keyword trie + failure	Kecil	Rendah

C. *Varian Matriks Penuh (ac/ac-q)*

Varian matriks penuh menyimpan fungsi transisi sebagai tabel state penuh: setiap state memiliki satu baris berisi 256 entri yang seluruhnya telah dipraproses menjadi DFA. Akibatnya, setiap karakter masukan dilayani dengan tepat satu pembacaan tabel tanpa penelusuran failure [9]. Sifat DFA inilah yang menjadikannya tercepat; Aho-Corasick menunjukkan bahwa sebuah NFA dapat memerlukan hingga dua kali jumlah transisi DFA untuk memindai aliran data yang sama [1], [9]. Implementasi vektor penuh yang dioptimalkan pada Snort dilaporkan 1,5–2,5 kali lebih cepat daripada implementasi awal [9]. Harga yang dibayar adalah memori: karena seluruh 256 kolom disimpan untuk setiap state tanpa memandang transisi yang benar-benar aktif, varian ini menempati ujung boros-memori spektrum dan menjadi pilihan ketika throughput diutamakan serta memori berlimpah [8].

D. *Varian NFA Biner (ac-bnfa)*

Varian ac-bnfa menyimpan automaton sebagai NFA biner yang padat: hanya transisi goto eksplisit yang disimpan, sedangkan failure link ditelusuri ketika transisi langsung tidak ditemukan [8]. Karena hanya entri tak-nol yang disimpan, kebutuhan memorinya jauh lebih kecil daripada matriks penuh; studi evaluasi Snort menempatkan ac-bnfa sebagai yang terbaik dalam utilisasi memori sekaligus termasuk terbaik dalam utilisasi CPU [8]. Konsekuensinya, sebuah NFA dapat memerlukan lebih banyak langkah transisi daripada DFA untuk karakter yang sama [1], [9]—persis pola yang terukur pada eksperimen Bab IV. Keseimbangan memori-rendah dengan kecepatan-tinggi inilah yang menjadikan ac-bnfa sebagai metode pencarian baku pada Snort.

E. *Varian Banded (ac-banded)*

Varian banded memanfaatkan pengamatan bahwa transisi aktif pada sebuah baris state cenderung mengelompok pada rentang kolom yang berdekatan [4]. Alih-alih menyimpan 256 entri, varian ini hanya menyimpan "pita" (band) kolom dari transisi tak-nol pertama hingga terakhir beserta indeks awal pita [9]. Semakin sempit pita, semakin besar penghematan memorinya [9]. Akses transisi tetap berupa pengindeksan langsung dalam pita sehingga relatif cepat, namun besar penghematan bergantung pada pola: bila transisi tersebar lebar, penghematan mengecil. Varian ini menempati posisi tengah spektrum—memori kecil dengan kecepatan sedang [8].

F. *Varian Sparse-Banded (ac-sparsebands)*

Varian sparse-banded menggabungkan dua gagasan sekaligus: penyimpanan jarang yang hanya menyimpan entri tak-nol dan pembagian baris atas beberapa pita [8], [9]. Setiap baris state dipecah menjadi pita-pita kecil yang masing-masing disimpan secara jarang, sehingga celah kosong yang lebar di antara kelompok transisi tidak ikut memakan memori seperti pada banded tunggal. Hasilnya adalah memori yang kecil dengan kecepatan sedang hingga tinggi, bergantung pada sebaran transisi [8]. Varian ini efektif ketika sebuah baris state memiliki beberapa gugus transisi yang terpisah jauh.

G. *Varian Split (ac-split)*

Varian ac-split membagi pola secara dinamis menjadi blok-blok agar pencocokan aturan lebih efisien [8]. Dengan memecah automaton menjadi bagian-bagian yang lebih kecil dan teratur, distribusi transisi per state menjadi lebih seragam sehingga representasi tabelnya lebih ringkas tanpa kehilangan banyak kecepatan. Studi evaluasi Snort menempatkan ac-split, bersama ac-bnfa, sebagai yang terbaik dalam utilisasi CPU [8]. Posisi spektrumnya adalah memori sedang dengan kecepatan tinggi.

H. *Varian Lowmem ()*

Varian lowmem menyimpan automaton sebagai trie kata-kunci mentah beserta failure link, tanpa membangun tabel transisi penuh maupun pita. Pendekatan berbasis trie semacam ini bertujuan memberikan representasi tabel state dengan memori paling minimum, namun lazimnya disertai penurunan kinerja yang terukur dibandingkan representasi matriks penuh [9]. Karena itu lowmem menempati ujung paling hemat-memori sekaligus paling lambat pada spektrum, dan dipilih hanya ketika anggaran memori sangat terbatas.

I. *Strategi Lanjutan di Luar Snort*

Snort Strategi-strategi yang lebih maju memperhalus titik kerja ini. Tuck dkk. memodifikasi automaton AC dengan kompresi bitmap dan kompresi lintasan (path compression) sehingga seluruh basis signature Snort dapat dimampatkan ke memori yang sangat kecil [4]. Pendekatan berbasis perangkat keras seperti P-AC mereduksi state graph menjadi trie maju saja dan menekan biaya hingga sekitar 18 bit per karakter untuk 6.166 string Snort [6], sementara arsitektur multikarakter memakai reduksi aturan transisi dan perfect hash sehingga turun ke 13,75 byte per karakter pada 2.200 signature [7]. Semua ini, sekali lagi, adalah cara berbeda menyimpan δ atas automaton yang secara konseptual sama.

Pada studi struktur data murni, representasi double-array dan triple-array memberikan kombinasi terbaik antara kecepatan akses state dan keringkasan memori, mengungguli larik penuh maupun senarai berantai [5]. Hal ini memperkuat tesis bahwa pemilihan representasi δ-lah yang menggeser kinerja AC di sepanjang spektrum ruang–waktu, bukan perubahan automaton.

IV. PENGUJIAN DAN IMPLEMENTASI

A. *Metodologi*

Untuk membuktikan tesis secara kuantitatif, penulis mengimplementasikan sendiri satu automaton Aho-Corasick dalam bahasa Python, lalu menyimpan fungsi transisinya dalam tiga representasi yang mewakili tiga titik berbeda pada spektrum: (R1) matriks penuh sebagai DFA, (R2) peta jarang goto+failure, dan (R3) NFA biner padat (BNFA-like). Ketiganya berasal dari automaton yang persis sama sehingga

melaporkan kecocokan yang identik; perbedaan semata pada penyimpanan dan evaluasi δ.

Memori dilaporkan sebagai ukuran tabel transisi dengan asumsi satu sel transisi setara indeks state 32-bit (4 byte), konsisten dengan cara literatur melaporkan "byte per karakter" [4], [6], [7]. Kecepatan diukur dengan dua metrik: (i) jumlah operasi transisi dan lompatan failure—metrik algoritmik yang independen terhadap implementasi—serta (ii) waktu dinding (wall-clock) dan throughput sebagai pelengkap. Cuplikan inti ketiga fungsi transisi diperlihatkan pada Lampiran kode 1.

B. Himpunan Pola dan Payload

Himpunan uji terdiri atas 217 signature sintetis bergaya NIDS (token HTTP, SQL injection, XSS, traversal lintasan, dan perintah shell) ditambah token acak untuk memperbesar automaton. Total panjang pola 1.724 karakter, menghasilkan automaton dengan 1.535 state dan 1.534 goto-edge. Payload uji berupa 2.000.000 byte acak dengan sebagian pola disisipkan pada posisi acak; pemindaian menemukan 4.124 kecocokan yang identik untuk ketiga representasi (diverifikasi melalui asersi).

Lampiran 1. Inti ketiga fungsi transisi δ(state, c)

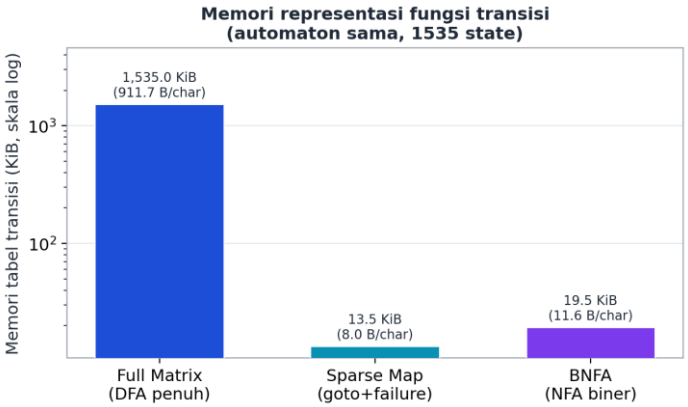
```
# R1 Full Matrix (DFA): tepat 1 pembacaan / karakter
def transition(self, state, c, counter):
    counter[0] += 1
    return self.delta[state*256 + c]

# R2 Sparse Map: ikuti failure link bila goto kosong
def transition(self, state, c, counter):
    while True:
        counter[0] += 1                 # probe goto
        g = self.goto[state].get(c)
        if g is not None: return g
        if state == 0: return 0
        counter[1] += 1                 # lompatan failure
        state = self.fail[state]

# R3 BNFA: goto eksplisit dipak ke array; binary search
def transition(self, state, c, counter):
    while True:
        counter[0] += 1
        g = self._goto(state, c)        # bisect dlm blok
        if g is not None: return g
        if state == 0: return 0
        counter[1] += 1
        state = self.fail[state]
```

Tabel II. Memori Tabel Transisi (Model, 1 sel = 4 byte)

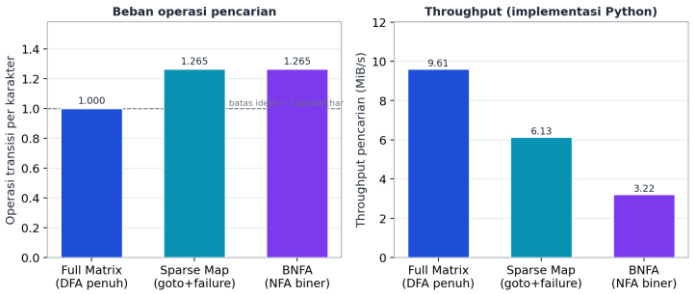
Representasi	Sel	Byte	Byte/char	Rasio
Full Matrix	392.960	1.571.840	911,74	113,8
Sparse Map	3.069	13.810	8,01	1,0
BNFA	3.069	19.950	11,57	1,4

Tabel III. Beban Operasi dan Throughput Pencarian (Payload 2 MB)

Representasi	Probe	Failure	Probe/char	MiB/s
Full Matrix	2.000.000	0	1,0	9, 61
Sparse Map	2.530.510	530.510	1,265	6,13
BNFA	2.530.510	530.510	1,265	3,22



Gambar III. Memori representasi fungsi transisi (skala logaritmik)



Gambar IV. Beban operasi per karakter (kiri) dan throughput (kanan).

C. Hasil dan Analisis

Tabel II dan Gambar 2 menampilkan biaya memori. Representasi matriks penuh menempati 1.571.840 byte (≈1,57 MB; 911,74 byte per karakter pola), sedangkan representasi jarang hanya 13.810 byte (8,01 byte per karakter) dan BNFA 19.950 byte (11,57 byte per karakter). Rasio memori matriks penuh terhadap representasi terkecil mencapai 113,8×. Inilah perwujudan terukur dari transition rule explosion: karena matriks menyimpan 256 sel per state tanpa memandang banyaknya transisi yang benar-benar aktif (rata-rata hanya ≈1 goto-edge per state pada himpunan ini), sebagian besar sel adalah pemborosan.

1) Tabel III dan Gambar 3 menampilkan biaya waktu. Pada metrik algoritmik, matriks penuh melakukan tepat 2.000.000 operasi transisi—satu per karakter—tanpa satu pun lompatan failure. Sebaliknya, representasi jarang dan BNFA sama-sama melakukan 2.530.510 operasi (1,265 per karakter) dengan 530.510 lompatan failure. Selisih inilah harga yang

dibayar untuk hemat memori: ketika goto kosong, automaton harus menelusuri failure link, sehingga jumlah operasi per karakter melampaui batas ideal satu probe per karakter.

Pada metrik waktu dinding, matriks penuh juga tercepat (9,61 MiB/s), diikuti sparse (6,13 MiB/s), lalu BNFA (3,22 MiB/s). Perlu ditegaskan bahwa nilai throughput absolut dipengaruhi oleh overhead interpreter Python; khususnya, binary search pada BNFA berbiaya tinggi dalam Python murni sehingga BNFA tampak paling lambat di sini. Pada implementasi C/perangkat keras yang lebih cache-friendly, representasi padat seperti BNFA justru mempertahankan kecepatan tinggi sambil hemat memori [6], [7]. Karena itu, metrik yang paling sahih untuk perbandingan algoritmik adalah jumlah probe dan lompatan failure (Tabel III), bukan waktu dinding.

D. Validasi terhadap Studi Empiris Snort

Kecenderungan hasil selaras dengan evaluasi varian Snort v2.9.9.1 oleh Mahajan dkk. [8]. Studi tersebut menemukan ac-bnfa terbaik dalam utilisasi memori, sedangkan ac-q (matriks penuh) unggul dalam throughput, dan ac-split serta ac-bnfa terbaik dalam utilisasi CPU [8]. Pola yang sama muncul pada eksperimen ini: representasi jarang/BNFA paling hemat memori, sementara matriks penuh melakukan operasi paling sedikit per karakter. Dengan demikian, pengukuran mandiri ini mereproduksi trade-off yang sama dengan yang teramati pada NIDS produksi.

V. KESIMPULAN

Makalah ini menunjukkan, secara konseptual dan empiris, bahwa varian-varian Aho-Corasick bukanlah algoritma yang berbeda melainkan representasi berbeda atas fungsi transisi dari satu automaton yang sama. Karena itu, keberadaan varian merupakan manifestasi langsung dari trade-off ruang–waktu klasik: matriks penuh membayar memori untuk meraih kecepatan maksimum (satu probe per karakter, tanpa failure), sedangkan representasi jarang dan padat menghemat memori dengan harga operasi tambahan berupa penelusuran failure.

Secara kuantitatif, pada himpunan 217 signature, matriks penuh menempati ≈1,57 MB sementara representasi jarang hanya 13,8 KB—rasio ≈114×—dengan selisih beban operasi 1,000 berbanding 1,265 probe per karakter. Panduan praktis pemilihan varian mengikuti langsung dari spektrum ini: bila memori adalah kendala utama, pilih representasi padat-jarang seperti ac-bnfa; bila throughput adalah prioritas dan memori berlimpah, pilih ac (matriks penuh) [8].

Keterbatasan utama studi ini adalah skala laboratorium, penggunaan Python yang membebani throughput absolut, dan asumsi alfabet penuh 256 simbol. Saran untuk pekerjaan lanjutan mencakup pengujian pada basis signature Snort berukuran penuh, penerapan representasi double-array atau perfect hashing yang terbukti unggul pada studi struktur data [5], serta evaluasi pada implementasi C atau akselerasi perangkat keras [6], [7] untuk mengukur trade-off pada titik kerja yang relevan bagi NIDS berkecepatan tinggi.

VIDEO LINK AT YOUTUBE

https://youtu.be/1xfYixs9-9I?si=MXgsWPy0Ut9XdS3Y

ACKNOWLEDGMENT

Saya ingin mengucapkan terima kasih kepada Tuhan Yang Maha Kuasa atas bimbingan-Nya yang telah memungkinkan saya menyelesaikan makalah ini untuk mata kuliah IF2211 Strategi Algoritma. Secara pribadi, saya juga ingin mengucapkan terima kasih kepada Bapak Rila Mandala, Ir. M.Eng. Ph.D., dosen mata kuliah IF2211 Strategi Algoritma, yang telah mengajarkan materi kuliah dengan jelas dan mendalam, sehingga memudahkan saya dalam menyelesaikan makalah ini.

REFERENCES

[1] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[2] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323–350, 1977.

[3] S. Antonatos, K. G. Anagnostakis, and E. P. Markatos, "Generating realistic workloads for network intrusion detection systems," in *Proc. 4th Int. Workshop on Software and Performance (WOSP)*, 2004, pp. 207–215.

[4] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *Proc. IEEE INFOCOM*, vol. 4, 2004, pp. 2628–2639.

[5] J. Nieminen and P. Kilpeläinen, "Efficient implementation of Aho–Corasick pattern matching automata using Unicode," *Software: Practice and Experience*, vol. 37, no. 6, pp. 669–690, 2007.

[6] D. C. Pao, W. Lin, and B. Liu, "A memory-efficient pipelined implementation of the Aho-Corasick string-matching algorithm," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 2, pp. 10:1–10:27, 2010.

[7] X. Wang and D. Pao, "Memory-based architecture for multicharacter Aho–Corasick string matching," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 143–154, 2018.

[8] A. Mahajan, A. Gupta, and L. S. Sharma, "Performance evaluation of different pattern matching algorithms of Snort," *International Journal of Advanced Networking and Applications*, vol. 10, no. 2, pp. 3776–3781, 2018.

[9] M. Norton, "Optimizing pattern matching for intrusion detection," Sourcefire, Inc., 2004. [Daring]. Tersedia: https://snort.org/documents/optimization-of-pattern-matches-for-ids

[10] K. Scarfone and P. Mell, "Guide to intrusion detection and prevention systems (IDPS)," National Institute of Standards and Technology, NIST Special Publication 800-94, 2007.

[11] R. Munir, "Pencocokan String (String/Pattern Matching)," Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, Institut Teknologi Bandung, 2026.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Richard Samuel Simanullang 13524112