Optimasi Pencocokan Pola dalam Sistem Deteksi Intrusi Snort Menggunakan GPU

Afrizal Fikri

13513004

Overview

1. Permasalahan
2. Analisis
3. Rancangan dan Implementasi
4. Pengujian
5. Kesimpulan

Permasalahan

Definisi

Intrusi

Serangkaian percobaan yang tidak berhak baik bertujuan untuk merusak maupun tidak terhadap sumber daya

Sistem deteksi intrusi

Mekanisme yang mengotomasi proses deteksi dan pencegahan intrusi terhadap satu atau beberapa sumber daya

Signature

Pola tertentu yang terdapat pada paket seperti *byte payload* atau servis tertentu

Sistem Deteksi Intrusi

Beberapa metode deteksi intrusi:

1. *Signature-based detection*
   Pencocokan dilakukan berdasarkan *signature* yang telah disiapkan
2. *Anomaly-based detection*
   Pencocokan dilakukan dengan membandingkan profil dengan *threshold* tertentu
3. *Stateful protocol analysis*
   Pencocokan dilakukan secara dalam dengan melihat rangkaian protokol dalam *traffic*

Latar Belakang

- Peningkatan kapasitas *traffic* jaringan dan jenis serangan
- Diperlukan sumber daya besar untuk analisis *traffic* jaringan
- Porsi analisis terbesar ada pada bagian pencocokan string
- Perkembangan GPU yang lebih cepat dan murah sebagai alternatif penggunaan CPU
- Kebutuhan akan implementasi pada GPU yang meminimalkan latensi

Rumusan Masalah

1. Bagaimana implementasi pencocokan pola dengan GPU pada IDS Snort?
2. Bagaimana kinerja sistem deteksi intrusi jaringan akibat dari penggunaan GPU?

Tujuan

1. Mengembangkan modul pencocokan pola dengan GPU yang terintegrasi dengan Snort IDS
2. Melakukan perbandingan kinerja sistem deteksi intrusi jaringan sebelum dan setelah menggunakan GPU

Analisis

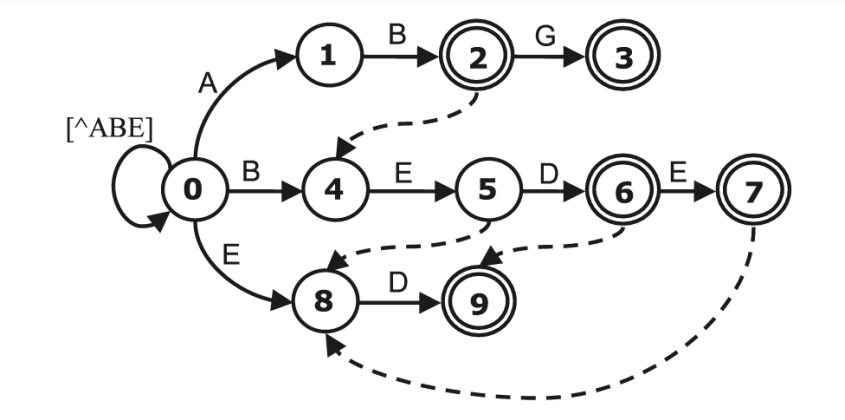
Analisis

- Algoritma pencocokan *signature*
- Struktur penyimpanan kamus
- Alokasi *thread*
- Optimasi latensi pada GPU

Algoritma Pencocokan String

- Pencocokan string menurut sumbernya ada 2 macam: *single pattern* dan *multi pattern*
- Pada *multi pattern string matching*, pola akan dikumpulkan dalam sebuah kamus
- Pencocokan dilakukan dengan pembacaan dari kamus sekaligus

Algoritma Pencocokan Signature



Algoritma akan dikembangkan dengan berbasis algoritma Aho-Corasick karena:

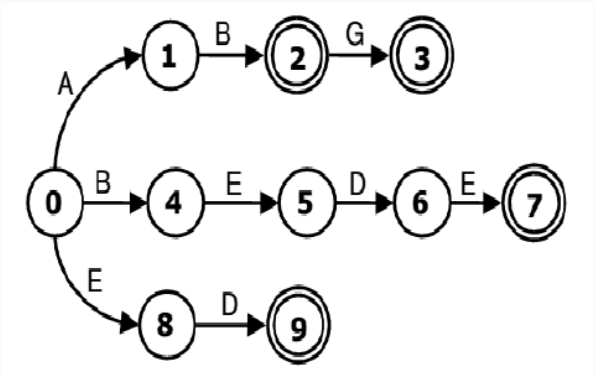
- mampu melakukan pencocokan kamus dalam sekali penelusuran
- stabil terhadap jumlah
- memiliki kinerja paling baik pada IDS Snort

Algoritma Pencocokan Signature

Adaptasi Aho-Corasick untuk memaksimalkan pencocokan secara *multithreading*:

Data Parallel AC
Stream input di partisi menjadi beberapa bagian, kemudian masing-masing akan dilakukan pencocokan dengan satu *thread*

Parallel Failureless AC
Masing-masing *thread* akan memulai pencocokan dari satu karakter dari *stream input* untuk mencari satu pola



Algoritma Pencocokan Signature

DPAC	PFAC
Porsi tiap *thread seimbang*	*Thread* memiliki umur yang pendek
Butuh ekstensi sepanjang pola terpanjang minus satu	Tiap *thread* hanya bertugas mencari satu pola
	Banyak terjadi *overlap* dalam akses *stream input*

Struktur Penyimpanan Kamus

- Pola akan disusun dalam struktur tertentu, seperti *tabel* atau *linked trie*
- Struktur data menentukan operasi pencocokan
- *Spatial locality* dapat berpengaruh pada keseluruhan latensi sistem

Struktur Penyimpanan Kamus

Pendekatan representasi *state machine*:

Linked trie

- Status terenkapsulasi
- Mudah dimodifikasi

Tabel transisi

- Mudah diimplementasi
- Cenderung statik

Alokasi Thread

- *Thread* PFAC akan banyak yang berhenti sangat awal
- Kinerja *warp* menjadi tidak seimbang
- Dapat diatasi dengan skema *assignment* berulang

Optimasi Latensi pada GPU

- Mengurangi transaksi ke memori global
  - Memuat masukan dari memori global ke *shared memory* dengan memanfaatkan *coalescing*
  - Membaca masukan yang *overlap* cukup dari *shared memory*
- Mengurangi latensi ke tabel transisi
  - Mengikat tabel transisi ke memori tekstur
  - Memuat baris pertama ke *shared memory*
- Mengurangi *swappiness* pada *input buffer* dengan *pinned memory*

Rancangan dan Implementasi

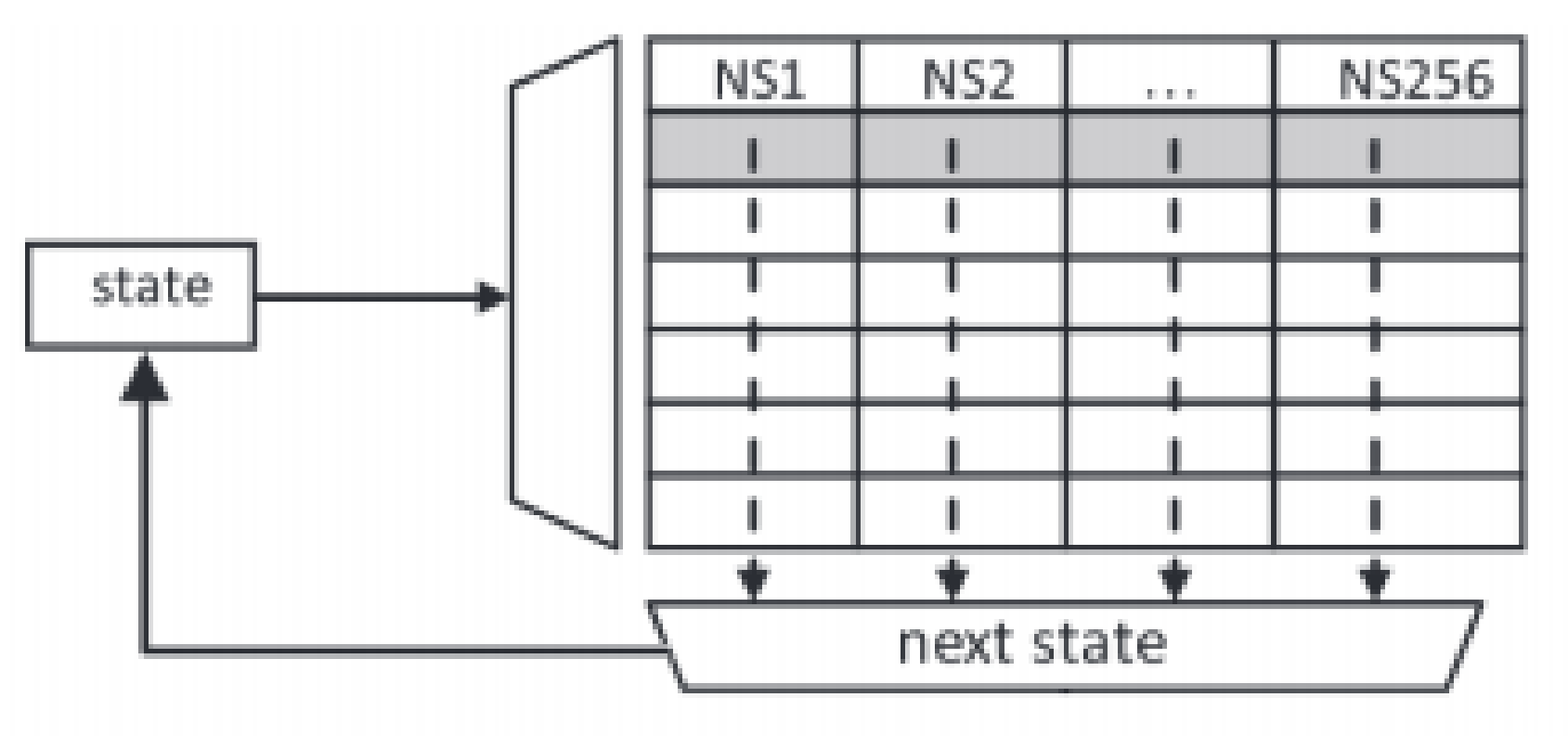
Rancangan Solusi

Pencocokan akan menggunakan implementasi PFAC dengan optimasi dengan memori tekstur dan *pinned memory*

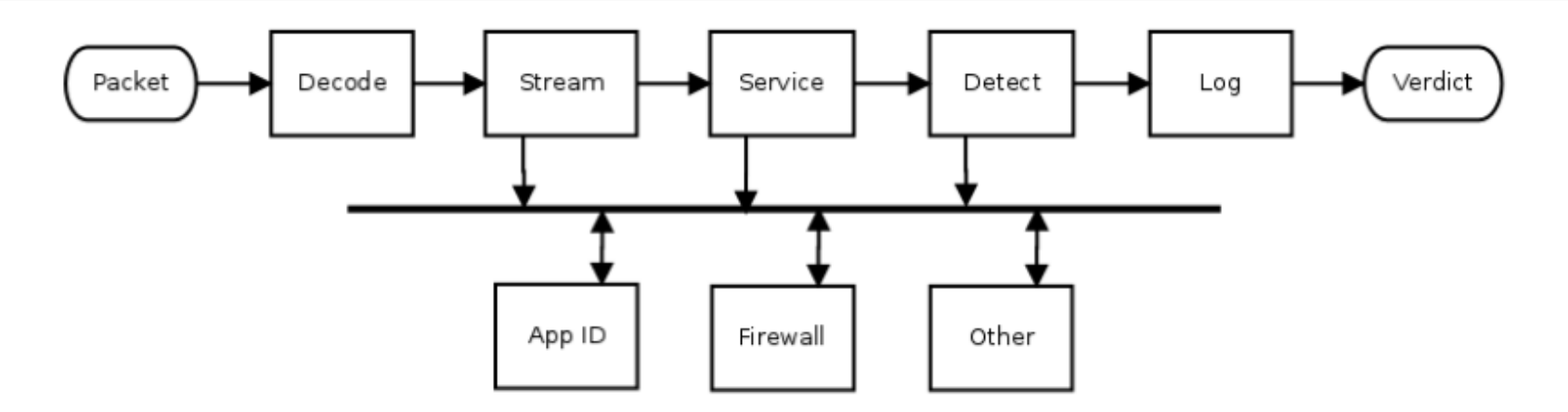
Secara umum, ada 3 tahap yang akan dilakukan oleh modul yang dikembangkan:

1. Pembentukan tabel transisi
2. Pemuatan tabel transisi dalam *device memory*
3. Penelusuran otomata

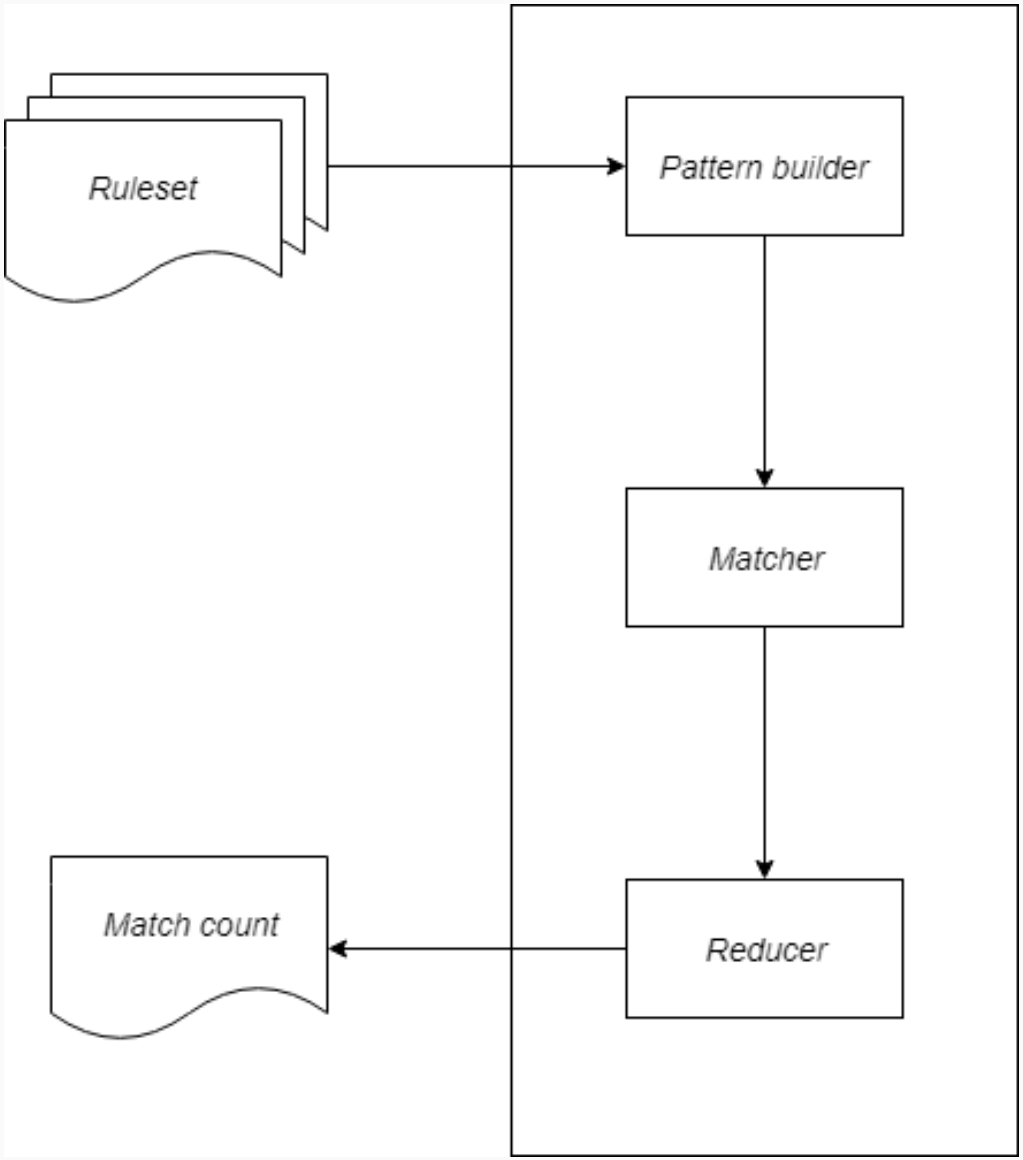
Tabel Transisi



Arsitektur Snort



Arsitektur Modul



Spesifikasi Komponen

No	Komponen	Spesifikasi
1.	*Handle*	Struktur data yang membungkus fungsionalitas pencocokan untuk memanggil komponen terkait
2.	*State machine*	Kamus yang digunakan dalam pencocokan
3.	*Kernel wrapper*	*Wrapper* yang menyiapkan *payload* sebelum kode *kernel* dijalankan
4.	*Matcher kernel*	Kode GPU yang melakukan penelusuran *state machine*
5.	*Reducer*	Komponen yang menggabungkan hasil pencocokan untuk dikembalikan ke Snort

Pengujian

Skenario Pengujian

Pengujian dilakukan dengan melakukan analisis paket dalam berkas PCAP. Berkas didapatkan dari arsip DEFCON sebesar 2,6 GB

Pengujian dilakukan dengan besar *buffer* berbeda dan 5 skenario:

1. *Baseline* (AC *multithread* CPU)
2. Skenario 1: PFAC dengan *global memory*
3. Skenario 2: PFAC dengan *shared memory*
4. Skenario 3: PFAC dengan *shared memory* dan *pinned memory*
5. Skenario 4: PFAC dengan *shared memory*, *pinned memory* dan *texture memory*

Hasil Pengujian

Buffer (kB)	*Baseline*	Skenario 1		Skenario 2	
	Runtime (detik)	*Runtime* (detik)	*Speedup*	*Runtime* (detik)	*Speedup*
128	94.632	450.629	0.21	29.207	3.4
512	92.656	463.281	0.2	39.321	3.16
1024	90.118	500.656	0.18	31.29	2.88
2048	96.097	436.805	0.22	31.302	3.07

Hasil Pengujian (bagian 2)

Buffer (kB)	*Baseline*	Skenario 3		Skenario 4	
	Runtime (detik)	*Runtime* (detik)	*Speedup*	*Runtime* (detik)	*Speedup*
128	94.632	31.44	3.01	30.825	3.07
512	92.656	38.446	2.41	29.137	3.18
1024	90.118	35.202	2.56	22.417	4.02
2048	96.097	31.404	3.06	20.446	4.7

Analisis

- Skenario 1 sangat lambat karena seringnya akses ke *global memory*
- Penggunaan *shared memory* pada skenario 2 mengurangi akses ke *global memory*
- Skenario 3 cenderung melambat dibandingkan skenario 2 karena *overhead* alokasi *pinned memory*
- Efektivitas *cache* milik *texture memory* pada skenario 4 selaras dengan besar *buffer*

Kesimpulan

Kesimpulan

1. Penggunaan GPU mampu meningkatkan kinerja hingga 3-4 kali lipat
2. Implementasi menggunakan algoritma Aho-Corasick tanpa *failuress* dengan memanfaatkan *thread* GPU
3. Digunakan beberapa teknik untuk mengurangi latensi akibat akses ke *global memory*

Terima kasih