Peningkatan Kinerja Modul Pencocokan Pola dalam Sistem Deteksi Intrusi Snort Menggunakan GPU

*Note: Sub-titles are not captured in Xplore and should not be used

Afrizal Fikri
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
afrizalf96@gmail.com

Achmad Imam Kistijantoro
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
imam@informatika.org

***Ringkasan*—Kemajuan teknologi telah memberikan kontribusi pada pertumbuhan pesat penggunaan data digital. Di era digital ini, banyak data fisik telah diubah menjadi digital. Di antara mereka, banyak data rahasia juga terjadi. Dengan demikian, jaminan keamanan menjadi penting juga. Salah satu titik masuk dari penggunaan berbahaya tersebut berasal dari sisi server. Memastikan otoritas pada server yang menggunakan sistem deteksi intrusi jaringan (NIDS) dapat mengambil sumber daya dan waktu yang sangat besar. Salah satu bagian penting dari NIDS adalah string yang cocok dengan bagian di dalam alat analisa. Berdasarkan alasan ini, makalah ini bertujuan untuk meningkatkan kecepatan deteksi yang cocok untuk memaksimalkan keseluruhan sistem throughput menggunakan GPU bukan CPU. Berdasarkan percobaan dan pengujian, solusi yang diusulkan memiliki waktu operasi yang jauh lebih baik dibandingkan dengan solusi linear state of the art dengan tetap menjaga akurasi.**

***Index Terms*—pattern matching; intrusion detection; GPU; parallel computation; CUDA**

I. Introduction

Salah satu sumber daya yang penting untuk dilindungi yaitu *server*. Banyak aspek security yang terdapat pada *server* dan dapat dengan mudah diserang [**?**]. Solusi untuk pengamanan *server* dari *request* berbahaya adalah dengan menggunakan sistem deteksi intrusi jaringan atau *network intrusion detection system* (NIDS). NIDS berfungsi mengenali kemungkinan serangan dari *request* yang diterima pada *server* atau *client* sebagai tindakan pencegahan. Ketika *request* dianggap berbahaya, maka dapat dilakukan tindakan lanjutan untuk mencegah kerusakan lebih lanjut pada aset. NIDS yang melakukan tindakan preventif seperti ini disebut juga NIPS (*network intrusion prevention system*).

Salah satu contoh NIDS yang banyak digunakan adalah NIDS Snort. Snort adalah salah satu NIDS berbasis *signature* yang mencocokan paket dengan *rule* yang telah dikumpulkan. *Rule* Snort terdiri dari beberapa komponen yang bertujuan mengenali *host* asal, *host* tujuan, dan isi paket jaringan. *Rule* yang digunakan berasal dari riset para pakar keamanan dan terus diperbaharui mengikuti pola-pola serangan terbaru.

Identify applicable funding agency here. If none, delete this.

Karena adanya peningkatan kecepatan *traffic* internet dan banyaknya serangan yang terjadi, maka dibutuhkan NIDS yang mampu melakukan deteksi dengan lebih cepat. Salah satu *bottleneck* dalam pengecekan paket adalah banyaknya *rule* yang harus dicocokkan [**?**]. Sehingga, peningkatan secara signifikan dapat dicapai salah satunya dengan meningkatkan kemampuan komputasi untuk pencocokan banyak paket secara paralel. Pemanfaatan *multithreading* pada *multicore processor* telah dikembangkan untuk mempercepat kinerja NIDS [**?**].

Selain mengoptimalkan penggunaan CPU, alternatif yang dapat digunakan yaitu penggunaan *general purpose* GPU (GPGPU) [**?**]. Dapat juga menggunakan prosesor yang mudah dikustomisasi seperti ASIC atau FPGA [**?**]. GPGPU banyak digunakan karena perangkat yang mudah didapatkan, multiguna, dan hanya membutuhkan lebih sedikit kustomisasi daripada ASIC dan FPGA. Selain itu, perbandingan kinerja antara GPGPU dan ASIC atau FPGA tidak terlalu jauh tanpa melakukan kustomisasi lebih lanjut di tingkat *low level* [**?**]. Maka, Tugas Akhir ini akan fokus untuk melakukan eksperimen terhadap metode pencocokan *string* pada NIDS yang berbasis GPGPU.

Perkembangan teknologi telah begitu cepat

II. String Matching

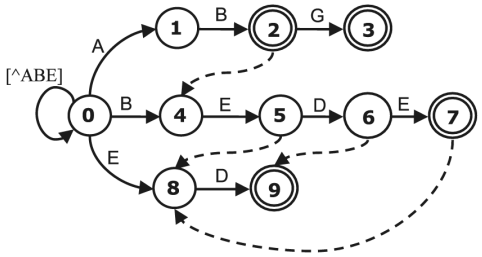
Secara umum, ada dua jenis pencocokan string: tunggal pencocokan pola string dan pencocokan pola string. Kami akan fokus ke pencocokan string pola multi. Ada beberapa algoritma terkenal untuk pencocokan string pola multi: Algoritma Aho-Corasick [**?**], Commentz-Walter [**?**], dan Wu-Manber [**?**]. Tulisan ini akan mencoba menerapkan variasi algoritma Aho-Corasick.

A. Aho-Corasick Algorithm

Algoritma Aho-Corasick adalah jenis algoritma pencocokan string yang bekerja dengan mencocokkan tiap string ke kamus. Kamus akan berisi semua pola untuk dicocokkan dalam bentuk *state machine*.

Dalam algoritma ini, operasi pencocokan dilakukan dengan menelusuri *state machine* untuk tiap karakter dalam string input. Jika sampai pada *final state*, maka string input dianggap

cocok dengan salah satu pola. Pencocokan berlanjut hingga string input berakhir.



Gambar 1. Example state machine of Aho-Corasick dictionary.

Aho-Corasick menggunakan *failure function* untuk melakukan *backtrack* setelah berhenti di salah satu *final state*. *Failure function* akan menunjuk dari *final state* sekarang ke prefiks pola terpanjang yang cocok dengan sufiks dari pola yang saat ini dicocokkan [**?**]. *Failure function* mengurangi redundansi untuk sekuens yang telah dilewati. Dengan metode ini, semua pola yang ada dalam kalimat dapat dicocokkan dengan sekali penelusuran dan mengurangi kompleksitas dari $O(mn)$ to $O(n)$.

Dalam algoritma ini, akses memori menjadi *bottleneck*. Setiap operasi penelusuran akan mengambil satu entri dari tabel. Karena akses memori lebih mahal daripada komputasi, algoritma ini terbatas dengan memori (*memory bound*) [**?**].

B. Data Parallel Aho-Corasick

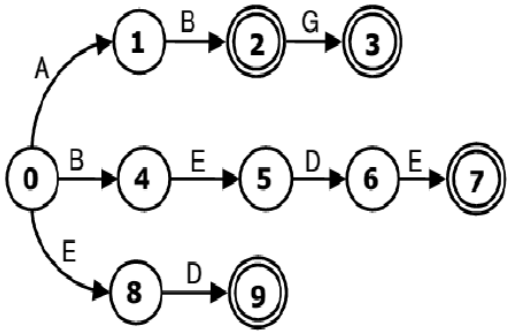
Untuk membuat Aho-Corasick menjadi *multithreading*, beban pencocokan perlu dibagi ke setiap *thread*. Salah satu caranya adalah dengan mempartisi input menjadi beberapa bagian dan kemudian setiap *thread* akan mencocokkan tiap bagian yang berkorespondensi terhadap kamus. Namun, jika ada pola yang menjangkau beberapa bagian sekaligus, pola tidak akan dikenali pada kedua *thread*. Masalah ini dikenal sebagai *boundary matching problem*. Sehingga perlu memperpanjang sesuai dengan panjang pola terpanjang. Akses memori meningkat menjadi $O((n/s + m) * s) = O(n + ms)$ dengan s adalah banyak bagian.

Gambar 2. Pattern **AB** not recognized by both thread 3 and 4

C. Parallel Failureless Aho-Corasick

Metode lain untuk mengadaptasi Aho-Corasick dengan *multithreading* adalah membagi *thread* ke semua *byte* karakter. Setiap *thread* akan mencocokkan pola yang dimulai dengan karakter yang berkorespondensi dan selesai ketika ada *final state* atau tidak ada transisi yang valid pada karakter [**?**]. Konsekuensinya adalah setiap *thread* paling banyak hanya akan cocok dengan satu pola.

Dengan demikian, *failure function* tidak diperlukan lagi. Selain itu, *boundary matching problem* tidak akan terjadi dengan menggunakan pendekatan ini. Pendekatan ini dinamakan juga sebagai *Parallel Failureless Aho-Corasick* (PFAC).



Gambar 3. State machine without failure function

III. Related Works

A. Utilization of Double Buffering Scheme, Texture Memory and Pinned Memory

Dalam tulisan ini, implementasi pencocokan berbasis GPU telah diusulkan menggunakan algoritma Aho-Corasick. Selain pemanfaatan GPU multithreading, ada beberapa penyesuaian untuk mengimpor Aho-Corasick ke GPGPU. Mesin negara dibangun menggunakan tabel transisi pada tabel 2D pada memori tekstur. Berdasarkan eksperimen, penerapan ini meningkatkan kinerja sebesar 19% [**?**].

Untuk skema transfer memori, skema *double buffering* diusulkan. Paket akan di-batch dalam satu *buffer*. Setiap kali *buffer* penuh, semua paket ditransfer ke GPU dalam satu operasi. Mentransfer dari perangkat ke host juga dilakukan dengan cara yang sama. Semua hasil akan dikumpulkan di *buffer* kedua dan ditransfer ketika penuh.

Trik optimasi lain yang digunakan adalah menggunakan *pinned memory*. Kedua *buffer* in host akan manggunanakan *pinned memory* untuk mengurangi swappiness. Dan juga *pinned memory* memungkinkan transfer secara langsung tanpa melalui CPU atau *direct memory access* (DMA) dan mengurangi kebutuhan komputasi dan latensi. Desain ini dapat meningkatkan kinerja sistem hingga 3,2 kali lebih tinggi daripada implementasi menggunakan *multithreading* pada CPU.

B. Scalable Architecture for Maximizing GPU Utilization

Dengan profiling, disimpulkan bahwa hambatan terjadi pada 3 komponen: akuisisi paket, pencocokan string pola multi, dan pencocokan aturan opsi oleh ekspresi reguler. Selain itu, di dalam komponen senar string saja, rata-rata utilisasi GPU masih rendah. Makalah ini menerapkan perbaikan arsitektur di sekitar NIDS. textcolor red Metode yang diusulkan adalah pipelining dan payload batching. Pipelining digunakan untuk memisahkan penggunaan benang dalam transfer dan pencocokan. Dan dengan arsitektur ini, NIDS dapat diskalakan ke beberapa GPU dan CPU contoh inti cite kargus2012.

Perbaikan yang dicapai dengan metode ini adalah sekitar 1,5 hingga 4 kali lebih tinggi dibandingkan dengan garis dasar Snort. Evaluasi dilakukan menggunakan CPU ganda Intel X5680 dengan masing-masing 12 core dan GPU ganda NVIDIA GTX 580. NIDS dijalankan di bawah lalu lintas jaringan dengan kapasitas sekitar 40 Gbps dan bisa mencapai throughput 25,2 Gbps.

C. Failureless Aho-Corasick by Maximizing Shared Memory Usage

Salah satu cara untuk melakukan pencocokan Aho-Corasick adalah dengan menetapkan setiap byte input ke setiap utas secara bersamaan. Kemudian setiap utas akan melakukan pencocokan textcolor red begin dengan byte assined. Pencocokan berhenti setiap kali keadaan akhir atau transisi tidak valid terjadi. Pendekatan ini tidak memerlukan fungsi kegagalan. Sehingga setiap utas hanya bertanggung jawab untuk mencocokkan paling banyak satu pola cite lin2013.

Pendekatan baru ini memanfaatkan penggabungan memori dalam GPU sehingga byte yang berurutan dapat dimuat sekaligus. Untuk lebih meningkatkan kinerja, transisi dari keadaan awal akan dimuat ke memori bersama untuk mengurangi transaksi ke memori global.

Untuk tabel transisi, kami tidak dapat memastikannya cocok dengan memori bersama. Jadi, sebagai alternatif untuk memori bersama, tabel transisi akan terikat ke memori tekstur sebagai gantinya. Memori tekstur memiliki beberapa kelebihan. Tekstur memori memiliki cache lebih besar dan cocok untuk melakukan pengambilan dengan pola akses yang sulit diprediksi. Pengujian menggunakan dataset PCAP dari DEFCON menunjukkan peningkatan kinerja yang signifikan. Menggunakan CPU Intel Core i7-950 dan GPU NVIDIA GTX 580, throughput sistem bisa menjadi 3 kali dibandingkan dengan Snort baseline.

IV. Proposed Solution

A. Matching Algorithm

Untuk mengimplementasikan Aho-Corasick ke GPU, kita perlu mengeksploitasi SIMD (instruksi tunggal banyak data) secara optimal. Pendekatan paralel data adalah salah satu pendekatan yang paling mudah untuk diterapkan. Setelah memisahkan potongan, masing-masing potongan menugaskan ke benang berbeda dengan ekstensi sepanjang pola terpanjang. Pendekatan ini memiliki 2 masalah: ekstensi meningkatkan transaksi data secara drastis, dan tidak optimal jika masukan dipisah menjadi banyak potongan. textcolor red GPU dapat menelurkan ratusan benang dan semoga semua utas dapat dimanfaatkan secara optimal.

Solusi alternatif adalah dengan menggunakan PFAC. Ide mengalokasikan thread individu ke setiap byte dari aliran input memiliki implikasi penting pada efisiensi mesin negara PFAC.

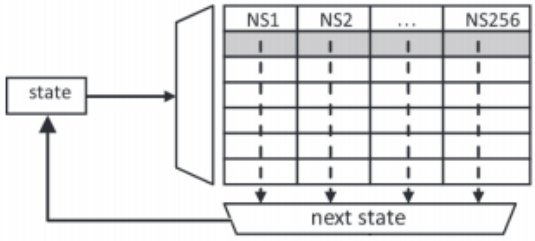
1) Tiap *thread* hanya bertanggung jawab dengan string yang dimulai dengan karakter pada *thread* tersebut. Ketika tidak ada pola yang cocok dengan huruf pada *thread*, pencarian langsung berhenti pada *thread*. Akibatnya, umur *thread* juga lebih singkat.
2) Dalam mesin PFAC, tiap 32 *thread* dalam *warp* akan mengakses memori yang berurutan dari memori global. Dengan demikian, akses tabel transisi fitur *memory coalescing* dapat dimanfaatkan.
3) Tidak memerlukan penyimpanan tambahan untuk *failure transition*. Sehingga konsumsi memori akan lebih sedikit dan kemungkinan *cache hit* lebih besar.

B. *Implementasi* State Machine

Ada 2 alternatif untuk implementasi mesin negara: tabel dua dimensi atau trie dengan pointer. Struktur trie mendukung modifikasi dinamis dan link-unlinking. Tetapi terlalu rumit untuk disimpan di GPU. Dan juga tempat ingatan bisa menjadi terfragmentasi dan tidak dapat mengeksploitasi lokalitas spasial. Belum lagi biaya mengalokasikan memori di GPU terlalu menuntut. Jadi, opsi ini kurang disukai.

Desain lain dan yang lebih mudah adalah meja dua dimensi. Untuk masing-masing negara bagian, akan ada transisi ke, katakanlah 256 karakter (jika input adalah unsigned ASCII char). Model ini sederhana dan mendukung penggabungan memori. Jika kita menggunakan memori tekstur, juga akan ada pola akses peluang yang akan di-cache.

Setelah semua pola terdaftar, tabel akan dikompilasi di host. Dan kemudian, meja dipindahkan ke perangkat seperti ukuran tabel yang terisi. Gambar. 4 di bawah ini menunjukkan desain tabel transisi.



Gambar 4. Transition table representation

Untuk mengurangi kebutuhan terhadap penanda status akhir, nomor status akan disusun ulang. Status akhir akan diletakkan pada status pertama hingga ke-N. Sedangkan status lainnya termasuk status awal dimulai dari N+1. Dengan demikian, kebutuhan memori untuk penanda akan berkurang. Selain itu, operasi pengecekan dalam kode *kernel* lebih sederhana dan ringan.

C. *Alokasi* Thread

Meskipun *thread* dibangkitkan sangat banyak, banyak juga diantara *thread* yang akan berhenti sangat awal. Sehingga, muatan pencocokan tiap *thread* dapat timpang cukup jauh dan utilisasi GPU rata-rata menjadi rendah. Untuk menunjang utilisasi GPU, [**?**] mengusulkan alokasi memori yang berulang.

Dalam satu blok, satu atau beberapa *byte stream* akan ditempatkan sebagai lokasi awal satu *thread*. Misal, dalam blok berisi 4.096 *byte* dengan 512 *thread*, *thread* pertama akan memulai pencocokan pada posisi kelipatan 512. Akibat dari

penugasan yang berulang, secara statistik kemungkinan ketimpangan muatan dari masing-masing *thread* akan berkurang [**?**].

D. Optimasi pada Memori GPU

Untuk memaksimalkan utilisasi GPU dan mengurangi latensi akibat transfer data, beberapa metode dapat diterapkan. Salah satu metode yang sangat umum adalah penggunaan *shared memory*. *Shared memory* memiliki latensi yang lebih kecil beberapa kali lipat dibandingkan *global memory*. Namun, ukuran *shared memory* sangat terbatas. Selain itu, *shared memory* harus di-"replikasi" ke setiap *block*.

Dengan memanfaatkan sifat *early terminate* pada PFAC, penggunaan *shared memory* dapat dihemat. Transisi dari *state* awal akan selalu diakses namun kemungkinan pencocokan berhenti sangat tinggi [**?**]. Sehingga transisi ini akan dimuat ke *shared memory* untuk menurunkan latensi ketika pembacaan tabel transisi.

Sisa tabel transisi yang masih ada di *global memory* akan ditingkatkan dengan *texture memory*. Diantara keuntungan *texture memory* adalah memiliki *cache* yang lebih besar dan mampu melakukan transpolasi sehingga dapat memanfaatkan *cache* untuk pola yang susah ditebak [**?**]. Kelemahan *texture memory* data ditulis hanya dari host, cocok dengan skema pencocokan yang melakukan kompilasi kamus sekali dalam satu *rule group*.

Metode lainnya yaitu dengan melakukan batching dan menyimpan batch dalam pinned memory. Pinned memory adalah memori yang dialokasi di host dan tidak dapat diswap ke disk. Penggunaan pinned memory untuk buffer input dapat menurunkan kemungkinan swappiness untuk ukuran buffer yang besar. Selain itu, pinned memory juga memungkinkan adanya DMA dari host ke device maupun sebaliknya [**?**].

V. Implementation pada Snort

A. Interface *MPSE*

Dalam mengimplementasi modul pencocokan string, akan digunakan *interface* MPSE (*multi pattern search engine*). Modul deteksi pada Snort akan memuat semua kelas yang mengimplementasi *interface* MPSE. Pilihan metode pencocokan akan diberikan dari konfigurasi sesuai definisi pada file header.

Interface MPSE sendiri berisi beberapa fungsi yang digunakan untuk analisis sebagaimana pada tabel

B. Modules

VI. Pengujian

A. Strategi Pengujian

Pengujian dilakukan untuk mendapatkan perbandingan kinerja antara modul yang dikembangkan dan modul yang ada dalam NIDS Snort. Pengujian dibatasi dengan menggunakan konfigurasi *data acquisition* PCAP dengan masukan *file* tcpdump saja. Berkas didapatkan dari arsip DEFCON tahun 2017. *Live testing* tidak diujikan dalam Tugas Akhir ini mengingat kapasitas *network interface* dapat menjadi *bottleneck* sehingga hasil perbandingan tidak akurat.

Pengujian dilakukan pada komputer dengan spesifikasi seperti pada Tabel I berikut.

Tabel I
Spesifikasi Lingkungan Pengujian

Komponen	Spesifikasi
Sistem Operasi	Ubuntu 16.04.5 amd64
CPU	Intel Core i7-7500U CPU @ 2.70GHz × 4
Host RAM	16 GB
Media Penyimpanan	SSD SATA III 6 Gbps
GPU	NVIDIA GeForce 940MX
Arsitektur GPU	Maxwell
Compute Capability	5.0
Dedicated Video RAM	2 GB
CUDA *Runtime*	CUDA 8.0 r2
C/C++ *Compiler*	GCC 5.4.0

Ada 5 skenario yang akan diujikan dalam penelitian ini:

1) *Baseline* (Aho-Corasick dengan *multithreading* CPU)
   Baseline akan menggunakan konfigurasi default Snort. Ini menjadi dasar pengukuran kinerja dan kebenaran program yang diimplementasikan pada GPU.
2) Skenario 1 (PFAC dengan *global memory*)
   Skenario ini adalah skenario paling dasar. Optimasi hanya dilakukan pada algoritma tanpa melibatkan optimasi pada latensi GPU.
3) Skenario 2 (PFAC dengan *shared memory*)
   Skenario ini memanfaatkan *shared memory* untuk mengurangi akses ke *global memory*. Transisi *state* awal ke tiap huruf serta *stream* masukan ditampung dalam *shared memory*.
4) Skenario 3 (PFAC dengan *shared memory* dan *pinned memory*)
   Skenario ini mirip dengan skenario 2 dengan tambahan *pinned memory* pada *buffer*. Mekanisme ini diharapkan dapat mengurangi *swappiness* dan memungkinkan DMA (*direct memory access*).
5) Skenario 4 (PFAC dengan *shared memory*, *pinned memory*, dan *texture memory*)
   Skenario ini mirip dengan skenario 3 dengan tambahan *texture memory* pada kamus. Dengan *cache* yang lebih besar, penggunaan *texture memory* diharapkan mengurangi transfer memori.

B. Pengujian Kinerja

asd

VII. Kesimpulan