PENINGKATAN KINERJA MODUL PENCOCOKAN POLA DALAM SISTEM DETEKSI INTRUSI SNORT MENGGUNAKAN GPU

Laporan Tugas Akhir

Disusun sebagai syarat kelulusan tingkat sarjana

Oleh

AFRIZAL FIKRI

NIM : 13513004

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

Oktober 2018

PENINGKATAN KINERJA MODUL PENCOCOKAN POLA DALAM SISTEM DETEKSI INTRUSI SNORT MENGGUNAKAN GPU

Laporan Tugas Akhir

Oleh

AFRIZAL FIKRI

NIM : 13513004

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Laporan Tugas Akhir

di Bandung, 1 Oktober 2018

Pembimbing,

Achmad Imam Kistijantoro, S.T., M.Sc., Ph.D.

NIP 19730809 200604 1 001

LEMBAR PERNYATAAN

Dengan ini saya menyatakan bahwa:

1. Pengerjaan dan penulisan Laporan Tugas Akhir ini dilakukan tanpa menggunakan bantuan yang tidak dibenarkan.
2. Segala bentuk kutipan dan acuan terhadap tulisan orang lain yang digunakan di dalam penyusunan laporan tugas akhir ini telah dituliskan dengan baik dan benar.
3. Laporan Tugas Akhir ini belum pernah diajukan pada program pendidikan di perguruan tinggi mana pun.

Jika terbukti melanggar hal-hal di atas, saya bersedia dikenakan sanksi sesuai dengan Peraturan Akademik dan Kemahasiswaan Institut Teknologi Bandung bagian Penegakan Norma Akademik dan Kemahasiswaan khususnya Pasal 2.1 dan Pasal 2.2.

Bandung, 1 Oktober 2018

Afrizal Fikri
NIM 13513004

ABSTRAK

PENINGKATAN KINERJA MODUL PENCOCOKAN POLA DALAM SISTEM DETEKSI INTRUSI SNORT MENGGUNAKAN GPU

Oleh

AFRIZAL FIKRI

NIM : 13513004

Saat ini, banyak transaksi penting terjadi dunia maya. Keamanan informasi menjadi jaminan yang sangat penting agar data-data penting tidak bocor dan disalahgunakan. Sistem deteksi dan pencegahan intrusi telah dikembangkan. Namun, kecepatan deteksi intrusi belum secepat peningkatan kecepatan jaringan.

Solusi yang dapat digunakan yaitu penggunaan *multithread* untuk pencocokan yang banyak secara paralel. Solusi *multithreading* telah dikembangkan menggunakan CPU. Namun, CPU memiliki *core* yang sangat terbatas. Alternatif lainnya adalah dengan menggunakan GPU. GPU memiliki kemampuan untuk membangkitkan banyak *thread* sekaligus. GPU sangat cocok untuk melakukan komputasi sederhana dalam jumlah besar.

Salah satu komponen penting dalam sistem deteksi intrusi yaitu pencocokan string. Beban pencocokan string ini seringkali menjadi *bottleneck* dalam analisis paket jaringan. Tugas akhir ini akan mencoba melakukan eksperimen untuk melakukan eksperimen pencocokan string pada sistem deteksi intrusi Snort menggunakan GPU.

Untuk dapat mempercepat proses analisis menggunakan GPU, translasi solusi yang ada tidaklah cukup. Operasi dalam GPU seringkali adalah operasi yang I/O *bound* dan *memory bound*. Diperlukan beberapa pengubahan seperti alokasi *thread* yang berbeda, metode transfer memori *host* dan *device*, skema penampungan paket, dan pengubahan struktur data *state machine*. Berdasarkan pengujian, implementasi sekarang dapat menghasilkan *speedup* yang signifikan hingga 3x lipat solusi *multithreading* dengan CPU yang ada.

Kata kunci: *pattern matching*, deteksi intrusi, GPU, optimasi, paralel.

KATA PENGANTAR

Puji dan syukur penulis panjatkan ke hadirat Tuhan yang Maha Esa karena atas berkat dan pertolongan-Nya, penulis dapat menyelesaikan Tugas Akhir dan laporan tugas akhir yang berjudul ”Peningkatan Kinerja Modul Pencocokan Pola dalam Sistem Deteksi Intrusi Snort Menggunakan GPU” ini dengan baik.

Dalam penyusunan Tugas Akhir ini penulis banyak mendapatkan masukan, kritik, dorongan, bantuan, bimbingan, serta dukungan baik secara fisik maupun moral dari berbagai pihak yang merupakan pengalaman yang berharga yang tidak dapat diukur secara materi dan dapat menjadi pembelajaran yang berharga dikemudian hari. Oleh karena itu dengan segala hormat dan kerendahan hati perkenankanlah penulis mengucapkan terima kasih kepada:

1. Bapak Achmad Imam Kistijantoro, S.T., M.Sc., Ph.D. selaku pembimbing penulis atas ilmu yang diberikan selama bimbingan maupun perkuliahan, kritik dan saran, serta dukungan moral yang diberikan selama proses pengerjaan tugas akhir.
2. Bapak Yudistira Dwi Wardhana Asnar, S.T., Ph.D. dan Riza Satria Perdana, S.T., M.T. selaku penguji atas kritik dan sarannya sehingga membuat tugas akhir ini menjadi lebih baik.
3. Segenap keluarga penulis. Terima kasih atas dukungan baik secara moral dan material sehingga penulis dapat melaksanakan Tugas Akhir ini hingga selesai dengan baik.
4. Ibu Dr. Fazat Nur Azizah S.T, M.Sc. dan Tricya Esterina Widagdo, S.T., M.Sc. selaku para pengajar matakuliah Tugas Akhir yang telah memotivasi penulis dan memberikan arahan dalam penyelesaian dan pengerjaan tugas akhir ini.
5. Bapak Dr.techn. Saiful Akbar S.T., M.T. dan Achmad Imam Kistijantoro, S.T., M.Sc., Ph.D. selaku Ketua Program Studi dari Teknik Informatika dan Sistem dan Teknologi Informasi yang ikut mendukung penulis dalam menyelesaikan tugas akhir ini.
6. Seluruh dosen, karyawan dan civitas program studi Teknik Informatika, Institut

Teknologi Bandung.

7. Para penduduk Sekeloagakure yang selalu siap mendengar keluh kesah dalam pengerjaan tugas akhir ini; Luqman Arifin Siswanto, Gazandi Cahyadarma, Muhammad Azam Izzuhri, dan Wiwit Rifai.
8. Khalil Ambiya, Feryandi Nurdiantoro, Nitho Alif Ibadurrahman, dan Muhamad Visat Sutarno yang sempat menjadi teman seperjuangan dalam mengerjakan tugas akhir ini.
9. Ibrohim Kholilul Islam, Wisnu, dan Aufar Gilbran yang menarik saya pada dunia kompetisi pemrograman sekaligus sebagian dasar-dasar ilmu komputer.
10. Rekan-rekan seperjuangan Teknik Informatika yang menamakan dirinya Happy Anti Wacana yang selalu mendukung penulis untuk mengerjakan Tugas Akhir.
11. Rekan-rekan dari Binary 2013 dan HMIF yang telah memberikan dukungan dan bantuan dalam pengerjaan Tugas Akhir.
12. Pihak-pihak lain yang tidak dapat disebutkan satu-persatu atas segenap bantuannya baik secara langsung maupun tidak langsung.

Penulis menyadari bahwa Tugas Akhir ini masih jauh dari sempurna serta memiliki banyak kekurangan. Oleh karena itu, penulis sangat terbuka dalam menerima kritik dan saran yang membangun untuk Tugas Akhir ini. Semoga Tugas Akhir ini dapat bermanfaat bagi pembaca.

Bandung, Oktober 2018

Penulis

Daftar Isi

Daftar Gambar

Daftar Tabel

BAB I

PENDAHULUAN

I.1 Latar Belakang

Penjaminan keamanan informasi merupakan cara-cara untuk mencegah pengaksesan, penggunaan, penyebaran, pengubahan, penyalinan, atau perusakan data yang tidak berhak. Beberapa aspek yang penting untuk dijamin dalam sebuah informasi adalah *confidentiality*, *integrity* dan *availability*. Ketiganya biasa disebut CIA Triad dan telah menjadi pedoman prinsip keamanan informasi yang dikenal luas (Evans et al., 2003). Pada tahun 1998, Donn Parker mengusulkan aspek tambahan penjaminan informasi yaitu *possession*, *authenticity*, dan *utility* (Parker, 1998). Parker berpendapat bahwa model CIA hanya berfokus pada aset saja tidak pada kontrol pengguna.

Pengamanan dapat dilakukan pada 2 sisi sumber daya, yaitu *tangible resources* dan *intangible resources*. *Tangible resources* yaitu aset yang berupa barang kasar (yang memiliki rupa). Contoh *tangible resources* misal *client*, *server*, dan *channel* komunikasi (komunikasi kabel dan nirkabel). *Intangible resources* yaitu aset yang tidak memiliki rupa, yaitu isi dari informasi yang dikirimkan. Baik keamanan dari *tangible resources* maupun *intangible resources* saling terkait dan memiliki peran penting dalam menjamin aset yang lain (Kizza, 2015).

Salah satu sumber daya yang penting untuk dilindungi yaitu *server*. Banyak aspek security yang terdapat pada *server* dan dapat dengan mudah diserang (OWASP, 2017). 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.

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 (Mitra et al., 2007). 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 (Haagdorens et al., 2005).

Selain mengoptimalkan penggunaan CPU, alternatif yang dapat digunakan yaitu penggunaan *general purpose* GPU (GPGPU) (Huang et al., 2008). Dapat juga menggunakan prosesor yang mudah dikustomisasi seperti ASIC atau FPGA (Das et al., 2008). 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* (Vasiliadis et al., 2008). Maka, Tugas Akhir ini akan fokus untuk melakukan eksperimen terhadap metode pencocokan *string* pada NIDS yang berbasis GPGPU.

I.2 Rumusan Masalah

Melakukan pencocokan pola secara paralel memiliki beberapa masalah yaitu algoritma yang digunakan, penyimpanan *dictionary*, dan metode *streaming* paket. Dan juga hasil yang diharapkan dapat menangani kasus dengan kapasitas aliran data yang besar. Maka dari itu, Tugas Akhir ini akan difokuskan pada eksperimen terhadap komponen pencocokan *string* paralel pada NIDS dengan GPU. Dalam rangka eksperimen tersebut, terdapat beberapa permasalahan yang menjadi fokus perhatian dari penelitian ini, yaitu:

1. Bagaimana implementasi algoritma pencocokan pola dengan GPU pada NIDS Snort?

2. Bagaimana optimasi yang digunakan untuk mengurangi latensi akibat penggunaan GPU?
3. Bagaimana kinerja modul pencocokan dibandingkan implementasi pada NIDS Snort?

I.3 Tujuan

Tujuan dari pelaksanaan Tugas Akhir ini adalah sebagai berikut:

1. Mengembangkan modul pencocokan pola dengan GPU yang dapat terintegrasi dengan Snort IDS
2. Melakukan optimasi pada struktur memori yang digunakan pada GPU
3. Melakukan perbandingan kinerja modul pencocokan sebelum dan setelah menggunakan GPU

I.4 Batasan Masalah

Merancang modul pencocokan string dengan menggunakan GPU diharapkan dapat mempercepat NIDS Snort. Dalam rangka pembangunan modul tersebut, terdapat beberapa batasan yang digunakan dalam penelitian ini, yaitu:

1. Modul pencocokan akan mengimplementasi *interface* MPSE *multi-pattern search engine* milik Snort
2. Modul dikembangkan menggunakan *platform* GPGPU CUDA
3. Masukan disuplai akan diambil dari *traffic log* jaringan berupa berkas PCAP
4. Pola didapat dari *rule* milik Snort VRT yang disesuaikan

I.5 Metodologi

Tahapan yang akan dilakukan selama pelaksanaan Tugas Akhir ini adalah sebagai berikut:

1. Analisis Permasalahan

   Pengerjaan Tugas Akhir ini diawali dengan analisis terhadap permasalahan yang

ingin diteliti lebih lanjut. Pada tahap ini, studi literatur perlu dilakukan dalam rangka mempelajari hasil-hasil penelitian terkait yang telah dilakukan, berikut metodologi penelitian serta ketercapaian yang berhasil didapatkan. Selain itu, tahap ini juga menjadi langkah awal untuk merumuskan landasan teori, penarikan hipotesis, hingga akhirnya dicapai rancangan solusi yang akan digunakan untuk membangun modul pencocokan.

2. Analisis Metode

   Pada tahap analisis metode, dilakukan eksplorasi yang bertujuan untuk mendapatkan metode-metode terbaik yang dapat digunakan sebagai komponen dalam pencocokan *signature*. Analisis yang dilakukan mencakup memori *layout* untuk *dictionary*, dan algoritma pencocokan *signature*. Selain dilakukan eksperimen secara umum dengan mengacu pada penelitian-penelitian serupa yang pernah dilakukan sebelumnya, akan dilakukan pula eksperimen dengan beberapa modifikasi yang dianggap perlu.

3. Implementasi Modul

   Pembangunan modul pencocokan dilakukan dengan memanfaatkan hasil yang didapatkan dari tahapan sebelumnya, yakni metode-metode dan algoritma pencocokan pola.

4. Evaluasi dan Penarikan Kesimpulan

   Pada tahap ini dilakukan evaluasi terhadap modul yang telah dibangun. Selain itu, dilakukan juga penarikan kesimpulan yang didasari oleh hasil evaluasi pengujian modul.

BAB II

STUDI LITERATUR

Pada bab ini akan dipaparkan mengenai beberapa penelitian terkait deteksi intrusi paralel berbasis GPU yang telah dilakukan sebelumnya berikut metodologi serta ketercapaian yang didapatkan. Selain itu, pada bab ini juga akan dijelaskan lebih lanjut mengenai landasan teori dari pengerjaan tugas ini.

II.1 Deteksi dan Pencegahan Intrusi

II.1.1 Deteksi Intrusi

Intrusi merupakan serangkaian percobaan untuk menyusup masuk, mengakses, mengubah, atau menyalahgunakan sumber daya yang berharga yang tidak berhak, baik sukses maupun tidak sehingga sumber daya menjadi tidak dapat dipercaya atau bahkan digunakan (Kizza, 2015). Proses intrusi sistem terbagi menjadi beberapa tahap. Proses dimulai dengan identifikasi target. Lalu target akan diperiksa secara menyeluruh dan dicari celah keamanannya. Selanjutnya dilakukan pengambilalihan akses ke sistem. Dan terakhir, sumber daya dalam sistem dapat dibaca, digunakan dan dimodifikasi.

Deteksi intrusi adalah kegiatan merekam, menganalisa, dan mendeteksi kemungkinan sebuah percobaan akses yang tidak berhak terhadap sebuah sistem (Kizza, 2015). Adanya percobaan akses ilegal dapat mengindikasikan adanya serangan dari luar (*hijacking*), maupun dari dalam (seperti *malware*). Setelah serangan ditemukan, aktivitas akan dicatat ke log untuk ditindak lebih lanjut.

Selain deteksi intrusi, kegiatan yang berkaitan adalah pencegahan intrusi. Pencegahan intrusi merupakan deteksi intrusi yang secara aktif melakukan penyaringan terhadap percobaan akses ke sistem. Jika terindikasi sistem terkena percobaan serangan, maka semua percobaan akses yang terkait akan dihentikan. Tindakan lebih lanjut yaitu aliran paket bisa diblok baik untuk sementara waktu maupun secara permanen.

II.1.2 Sistem Deteksi dan Pencegahan Intrusi

Sistem deteksi dan pencegahan intrusi atau *intrusion detection and prevention system* (IDPS) adalah mekanisme yang mengotomasi proses deteksi dan pencegahan intrusi kepada satu atau beberapa node. Sistem mengambil paket yang masuk sambil melakukan pengecekan kemudian membuat laporan hasil inspeksi. Laporan hasil inspeksi dapat digunakan untuk menentukan aksi yang akan dilakukan pada paket (Scarfone and Mell, 2007).

Ada dua macam NIDPS dilihat dari cakupan deteksinya:

1. *Network Based IDPS (NIDPS)*

   IDPS tipe ini dapat melakukan analisis aliran paket dalam suatu subnet jaringan. IDPS ini biasanya diletakkan di sebuah segment jaringan seperti *gateway*, *hub / switch*, *router*, dsb. Cara kerja NIDPS mirip dengan *network sniffer*, yaitu mengambil seluruh paket yang berjalan pada jaringan. IDPS jenis ini digunakan untuk mengantisipasi serangan jenis DoS (*denial of service*) dan *network probing*.

2. *Host Based IDPS (HIDPS)*

   IDPS tipe ini dapat mengambil seluruh informasi pada sistem operasi seperti *audit trails*, *logs*, *system logs*, dsb serta melakukan analisis terhadap perilaku sistem. IDPS jenis ini digunakan untuk mengantisipasi serangan yang menargetkan sistem operasi, seperti *buffer overflow*, *malware*, dll.

Fokus tugas akhir ini akan membahas tentang NIDPS.

II.1.3 Metode Deteksi Intrusi

Ada tiga macam metode deteksi yang biasa dipakai dalam NIDPS:

1. *Signature-based Detection*

   Signature adalah pola aktivitas tertentu yang mengindikasikan adanya ancaman. IDS berbasis *signature* melakukan pencocokan *signature* dengan aktivitas sistem untuk mengidentifikasi insiden yang mungkin. Pengenalan berbasis *signature* sangat efektif dan cepat untuk mendeteksi ancaman yang telah dikenal. Tapi metode ini tidak efektif untuk mendeteksi ancaman yang belum pernah dikenal sebelumnya atau ancaman yang disamarkan dengan teknik-teknik tertentu.

Metode ini adalah metode termudah karena hanya membandingkan unit aktivitas saat ini, seperti paket atau *log entry*, dengan daftar *signature* menggunakan operasi perbandingan string. Selain itu jumlah *false alarm* yang dihasilkan juga relatif sedikit. Metode berbasis *signature* membutuhkan sedikit pengetahuan tentang protokol jaringan dan aplikasi, dan tidak dapat melacak atau mendapat *state* dari jaringan komunikasi yang kompleks.

2. *Anomaly-based Detection*

   IDS berbasis anomali melakukan analisis parameter aktivitas yang dianggap normal dengan aktivitas sistem sekarang untuk melihat penyimpangan yang signifikan. Sistem menggunakan profil untuk menentukan parameter aktivitas normal seperti user, host, koneksi jaringan, atau aplikasi. Profil dikembangkan menggunakan pembelajaran mesin dengan memantau karakteristik dari aktivitas sejenis selama selang waktu tertentu.

   Sistem akan menggunakan metode statistik untuk membandingkan karakteristik aktivitas sekarang terhadap batas yang didefinisikan berdasarkan profil. Ketika ada aktivitas yang menyimpang cukup signifikan dari batas profil, aktivitas tersebut akan dicatat ke log dan dilaporkan ke *administrator* / pengelola. Profil bisa dikembangkan dari berbagai atribut, seperti jumlah email yang dikirim oleh pengguna, jumlah percobaan login yang gagal, dan tingkat penggunaan CPU *host* waktu tertentu.

   Keuntungan dari metode berbasis anomali ini adalah dapat mendeteksi ancaman yang belum pernah dikenal sebelumnya. Contoh, sebuah proses yang menggunakan sumber daya dalam jumlah besar, mengirim banyak email, menjalankan banyak koneksi, dan melakukan kegiatan lain yang cukup berbeda dari profil sistem normal akan terdeteksi sebagai *malware*.

   Profil awal dibangkitkan selama rentang waktu tertentu yang disebut *training period*. Kemudian profil dapat diperbarui secara statis atau dinamis. Metode statis yaitu jika profil diperbarui secara manual. Sedangkan metode dinamis yaitu ketika profil diperbarui menggunakan data latih dari log aktivitas sistem.

3. *Stateful Protocol Analysis*

   Stateful protocol analysis atau analisis protokol dengan *state* adalah proses membandingkan profil yang telah ditentukan sebelumnya terhadap definisi aktivitas protokol yang dianggap tidak berbahaya dan berlaku umum untuk setiap protokol. Profil diukur terhadap batas tertentu untuk mengidentifikasi penyimpangan.

   Tidak seperti deteksi berbasis anomali, yang menggunakan profil khusus host atau jaringan, analisis protokol dengan *state* bergantung pada profil universal yang dikembangkan vendor yang menentukan bagaimana protokol tertentu seharusnya dan tidak boleh digunakan. Contoh, saat pengguna memulai sesi *File Transfer Protocol* (FTP), sesi awalnya dalam *state* yang tidak terotorisasi. Pengguna yang tidak terotorisasi hanya dapat melakukan beberapa perintah di *state* ini, seperti melihat informasi umum dan memberikan nama pengguna dan kata sandi.

   Bagian penting dari pengenalan *state* adalah mencocokkan *request* dengan respons, jadi ketika upaya otentikasi FTP terjadi, IDS dapat menentukan apakah berhasil dengan menemukan kode *state* dalam respons yang sesuai. Setelah pengguna berhasil dikonfirmasi, sesi diset berada dalam keadaan terotentikasi, dan pengguna dapat melakukan lebih banyak perintah. Contohnya adalah jika terlalu banyak perintah dalam suatu *request* dilakukan dalam *state* yang tidak diotentikasi, maka akan dianggap mencurigakan.

Selain ketiga kategori di atas, penggunaan metode deteksi *hybrid signature-based* dan *anomaly-based* menjadi populer karena fleksibilitas dan kinerjanya. Cakupan dari *signature-based* NIDPS dapat ditingkatkan dengan *rule* baru hasil pembelajaran dari model *anomaly-based*.

II.2 Snort NIDPS

Snort NIDPS adalah sistem deteksi dan pencegahan intrusi yang mampu melakukan analisis lalu lintas secara *real-time* dan *logging* paket dalam jaringan. Snort mampu melakukan eksekusi *rule* untuk deteksi lalu lintas yang berbahaya dan melaporkan ke pengelola. *Rule* dibentuk berdasarkan perbedaan protokol paket dalam jaringan. *Rule* ini kemudian digunakan untuk melakukan analisis protokol dan pencocokan. Kumpulan *rule*

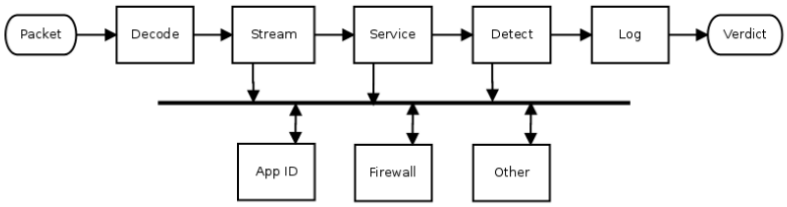
ini kemudian disebut juga kebijakan (*policies*).

II.2.1 Modus Snort NIDPS

Snort memiliki 3 buah modus; modus pengendus, modus pencatat, dan modus deteksi intrusi. Modus pengendus hanya menampilkan informasi tentang *header* paket. Perintah yang berbeda dapat digunakan menampilkan *header* untuk protokol berbeda. Modus pencatat hanya mencatat paket ke dalam berkas. Modus ini dapat digunakan untuk menggolongkan payload berdasarkan protokol dalam direktori yang berbeda. Dan terakhir, modus deteksi intrusi akan melakukan deteksi intrusi dari paket yang disimpan maupun analisis lalu lintas jaringan secara *real-time*. Modus ini dikonfigurasi menggunakan sebuah file konfigurasi. Melalui modus ini, Snort juga dapat dikonfigurasi agar menjadi sistem pencegah intrusi (NIPS) untuk menghalangi jaringan dengan membuat sebuah *bridge*.

II.2.2 Implementasi Snort NIDPS

Snort memiliki beberapa modul; modul penangkap paket, modul *decoder*, modul preproses, modul detektor, dan modul keluaran. Pada Snort versi 3.0, preproses dibedakan menjadi dua proses, yaitu *stream* dan *service* Alur paket dari mulai akusisi sampai paket berhasil dideteksi dan ditindak kurang lebih seperti Gambar II.1 berikut.



Gambar II.1: Arsitektur Snort versi 3

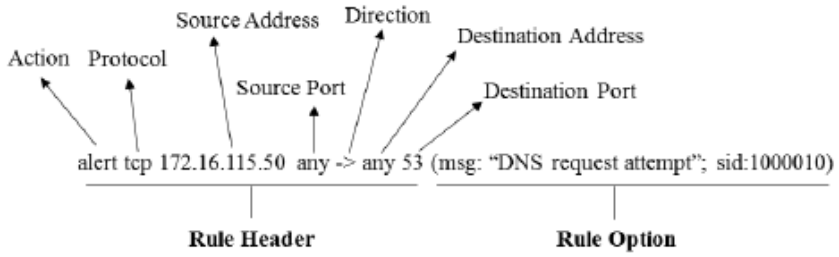
Modul penangkap bertujuan mengambil paket dari lalu lintas jaringan. Lalu paket akan diproses dengan modul *decoder* akan mengidentifikasi protokol paket dan mengklasifikasikannya berdasarkan spesifikasi data-link. Preproses bertujuan mengolah paket berdasarkan jenis protokol paket untuk kemudian siap dideteksi. Ini sekarang dilakukan dalam komponen *service*. Preproses juga bertujuan menyusun fragmen IP dan

stream TCP yang dilakukan pada komponen *stream*. Selain itu paket juga dicek jika berada diluar TCP *window*, dan akan disusun ulang.

Kemudian, paket akan masuk modul pencocokan. Modul ini akan mengelompokkan *rule* dan paket berdasarkan grup yang disusun berdasarkan protokol, port, dan servis. Lalu terakhir, paket akan dicek oleh modul detektor menggunakan *rule* yang tersedia dan mengirimkan hasilnya ke modul keluaran.

II.2.3 *Rule* Snort NIDPS

Rule adalah kumpulan kata kunci yang digunakan dalam mendeteksi kemungkinan pelanggaran kebijakan keamanan. Snort akan melakukan pencocokan paket berdasarkan kondisi yang ditentukan dalam tiap *rule*. *Rule* memiliki struktur dasar berupa *header* dan *body* seperti pada Gambar II.2 berikut.



Gambar II.2: Struktur *rule* Snort NIDPS

Header berisi spesifikasi dari berupa aksi, protokol, IP sumber, port sumber, operator, IP tujuan, port tujuan. Operator menandai arah dari paket. Arah paket bisa satu atau dua arah. Aksi akan dijalankan jika signature paket cocok dengan *rule* (Azimi et al., 2009). Sedangkan *body rule* berisi kumpulan *option* yang digunakan untuk pencocokan paket.

Beberapa *option* yang biasa digunakan dalam Snort *Rule* diantaranya:

1. *sid*, ID *rule*
2. *msg*, pesan kesalahan yang ditampilkan ke pengelola atau *log*
3. *flow*, arah aliran paket dari atau ke server
4. *flags*, flag TCP yang dicari
5. *content*, konten yang dicocokkan berupa teks

6. *reference*, acuan dari *vulnerability* yang mungkin

II.3 *Pattern Matching*

Pattern matching adalah metode mencari kemunculan sebuah string atau substring terhadap string lain. *Pattern matching* menjadi salah satu komponen inti dalam NIDPS. Secara umum ada dua jenis utama pencocokan pola, yaitu *single pattern matching* dan *multi-pattern matching*.

Single pattern matching yaitu pencocokan masukan terhadap satu pola. Algoritma yang tergolong *single pattern matching* yaitu algoritma Knuth-Morris-Pratt dan Boyer-Moore. Sementara itu, *multi-pattern matching* mencocokkan masukan terhadap beberapa pola sekaligus. Algoritma yang termasuk dalam *multi-pattern matching* yaitu algoritma Aho-Corasick dan Wu-Manber, yang merupakan modifikasi dari algoritma Boyer-Moore.

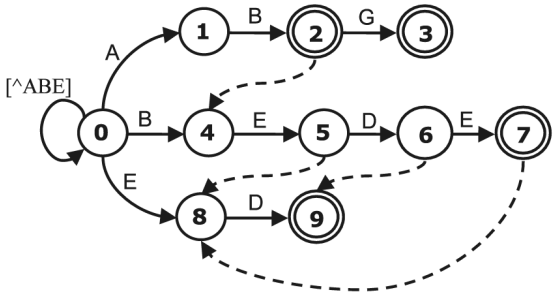
Masalah yang akan diselesaikan dalam pencocokan pola pada IDS merupakan *multipattern string matching*. Algoritma Wu-Manber tidak digunakan dalam implementasi Snort versi 3.0. Sehingga fokus eksperimen ini yaitu algoritma Aho-Corasick.

II.3.1 Algoritma Aho-Corasick

Algoritma Aho-Corasick adalah algoritma *multi-pattern matching* yang dapat mencari kemunculan string dalam sebuah kamus (Aho and Corasick, 1975). Algoritma ini dapat mencari kemunculan setiap string dalam kamus dalam sekali pencocokan. Komponen utama dalam algoritma ini yaitu kamus yang berupa *finite automata* seperti pada Gambar II.3 di bawah.

Kamus berisi kumpulan string *rule* yang akan dicocokkan. Penyusunan kamus hanya perlu dilakukan sekali sebelum pencocokan string dilakukan. Penyusunan kamus dilakukan dalam $O(kp)$, dengan k adalah jumlah string pada kamus dan p adalah panjang maksimum string pada kamus.

Dalam kamus ada yang disebut *failure transition*. *Failure transition* adalah tabel yang berisi posisi terakhir prefiks yang cocok dengan sufiks string yang telah dilewati. Sehingga ketika pencarian tidak cocok pada suatu string, akan dilanjutkan pada string berikutnya



Gambar II.3: *Finite automata* kamus algoritma Aho-Corasick dengan *failure transition* (Lin et al., 2013)

yang prefiksnya cocok dengan sufiks string sebelumnya. Tujuan adanya *failure transition* yaitu untuk menghindari pengulangan ketika adanya *mismatch*. Kompleksitas pencarian algoritma ini yaitu $O(x)$, dengan x adalah panjang string pola.

II.3.2 Pembentukan Kamus

Kamus dalam Aho-Corasick dibentuk dari kumpulan *string* masukan. Pembentukan kamus dimulai dari *state* awal yang berupa *string* kosong. Kemudian tiap *string* akan ditelusuri per huruf. Tiap huruf akan menjadi transisi ke *state* baru. Ketika penelusuran telah mencapai akhir *string*, maka *state* ditandai sebagai *state* akhir.

Langkah selanjutnya yaitu membentuk *failure function*. *Failure function* dari tiap *state* adalah prefiks terpanjang pada kamus yang sama dengan sufiks pola. Transisi *failure function* dicari dengan melakukan iterasi pada *previous state*. Tiap *previous state* akan dicek apakah ada *failure function* yang memiliki transisi yang sama dengan *state* yang ingin dibentuk *failure function*-nya. Ketika ada prefiks yang lebih panjang, maka *failure function* akan diperbarui.

II.3.3 *Boundary Detection Problem*

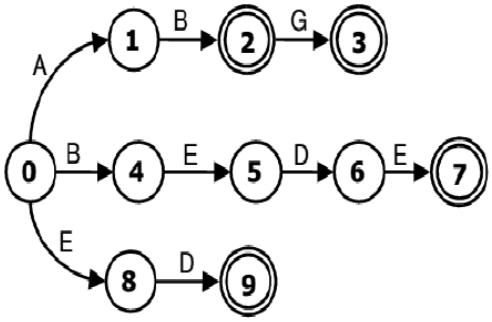
Untuk dapat meningkatkan kinerja algoritma Aho-Corasick, string dapat dipartisi menjadi beberapa segmen. Sehingga tiap partisi dapat dicocokkan menggunakan satu *thread*. Namun terdapat ada masalah pada pencocokan partisi ketika pola yang dicari ternyata berada pada lebih dari satu partisi. Bagian pola akan terpotong seperti pada Gambar II.4 di bawah dan tidak dapat dideteksi. Masalah ini dikenal sebagai *boundary detection problem*.

Gambar II.4: *Boundary detection problem* terjadi pada pola ”AB” tidak terbaca oleh *thread* 3 dan 4 (Lin et al., 2013)

Solusi dari masalah itu yaitu dengan menambah jangkauan pencocokan sepanjang $m-1$, yaitu panjang pola terpanjang dikurangi satu. Total kompleksitas yang diperoleh tiap thread menjadi $O(n/s+m)$ dengan n adalah panjang masukan dan s adalah banyak segmen yang dibentuk. Sehingga total kompleksitas untuk *lookup* memori menjadi $O((n/s+m)*s) = O(n+ms)$. Dibandingkan dengan *lookup* pada Aho-Corasick biasa yang hanya $O(n)$, operasi ini dapat menjadi bottleneck pada pencarian pola.

II.3.4 *Parallel Failureless Aho-Corasick*

Algoritma *Parallel Failureless Aho-Corasick* (PFAC) merupakan pengembangan dari algoritma Aho-Corasick untuk melakukan pencocokan dengan *multithreading*. PFAC menggunakan *state machine* yang tidak mengandung *failure transition* dan masing-masing *thread* memulai pencocokan dari tiap satu karakter masukan (Lin et al., 2013). Bentuk *state machine* PFAC terlihat seperti pada Gambar II.5.

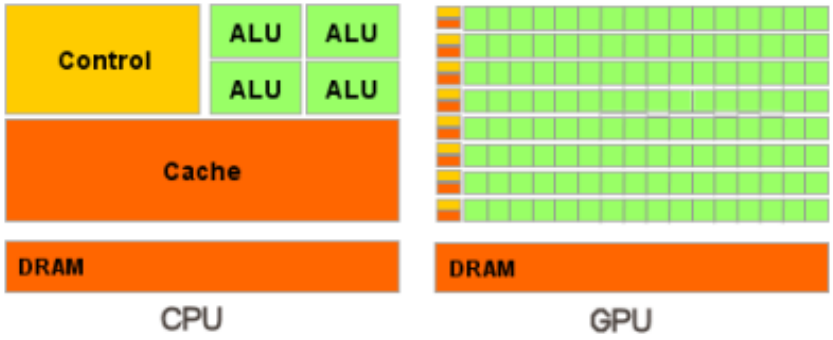


Gambar II.5: *Finite automata* Aho-Corasick tanpa *failure transition* (Lin et al., 2013)

II.4 *General-Purpose computing on Graphics Processing Units*

General-purpose computing on graphics processing units (GPGPU, kadang disebut juga GPGP atau GP^{2}U) adalah teknik untuk menggunakan pemroses grafis / GPU untuk melakukan komputasi umum yang biasa dilakukan pada pemroses CPU. Teknik ini

memanfaatkan desain GPU berbeda dari CPU yang khusus digunakan untuk memroses grafik dengan unit piksel yang sangat banyak. Sehingga GPU memiliki banyak *core* yang berjalan paralel dan berbeda dari CPU yang biasa digunakan untuk melakukan operasi sekuensial (Lindholm et al., 2001). Dalam GPU terdapat beberapa *stream multiprocessor* (SM) yang masing-masing memiliki beberapa *core thread* atau *stream processor* (SP). Ilustrasi perbedaan CPU dan GPU terlihat seperti pada Gambar II.6 berikut.



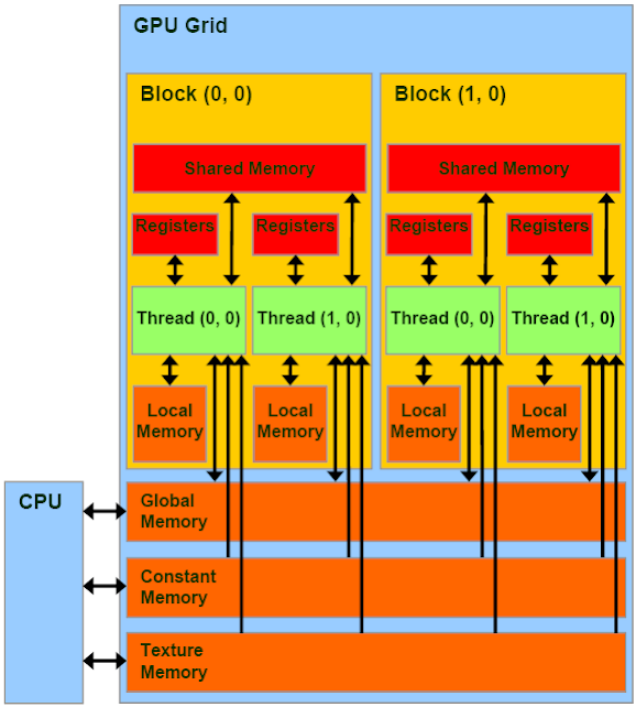
Gambar II.6: Perbedaan arsitektur CPU dan GPU (NVIDIA, 2017)

II.4.1 Struktur Memori

Tiap *core thread* memiliki memori lokal yang saling independen. Dalam tiap blok, *thread* yang ada dalam blok dapat mengakses *shared memory*. Kemudian ada memori global yang dapat diakses semua *thread*. Hirarki memori GPU kurang lebih seperti pada Gambar II.7.

Ada perbedaan kecepatan ketika melakukan operasi pada tiap tingkatan memori. Makin tinggi tingkatan memori, makin besar latensi akses memori. Urutan besar latensi akses memori yaitu memori lokal, memori bersama, kemudian memori global. Sehingga diutamakan menggunakan memori yang lebih rendah dahulu. Jika membutuhkan sinkronisasi antar *thread*, memori yang lebih tinggi dapat digunakan.

Selain ketiga memori tersebut, ada juga memori yang hanya dapat ditulis oleh *host* yaitu memori konstanta dan tekstur. Kedua memori ini hanya dapat ditulis sekali sebelum *kernel* dipanggil. Keuntungan menggunakan kedua memori ini yaitu latensi akses yang lebih kecil dari memori global. Hal ini terjadi karena kedua memori ini memanfaatkan *locality* dalam akses memori. Memori konstanta menggunakan *cache* sehingga akses untuk memori yang sama berkali-kali akan sangat cepat. Sedangkan memory tekstur



Gambar II.7: Tingkatan memori GPU (NVIDIA, 2017)

menggunakan *spatial locality* untuk *thread* yang berdekatan.

II.4.2 *Compute Unified Device Architecture*

Compute Unified Device Architecture (CUDA) adalah platform dan API untuk GPGPU oleh NVIDIA. CUDA menyediakan antarmuka untuk dapat menggunakan GPU NVIDIA untuk melakukan GPGPU (NVIDIA, 2017). CUDA API dapat diimplementasi menggunakan bahasa C, C++, dan Fortran. Selain itu, ada juga *binding* yang dibuat dalam pustaka beberapa bahasa oleh pihak ketiga, seperti Python, Perl, Java, Ruby, Haskell, Lua, MATLAB, dan Mathematica.

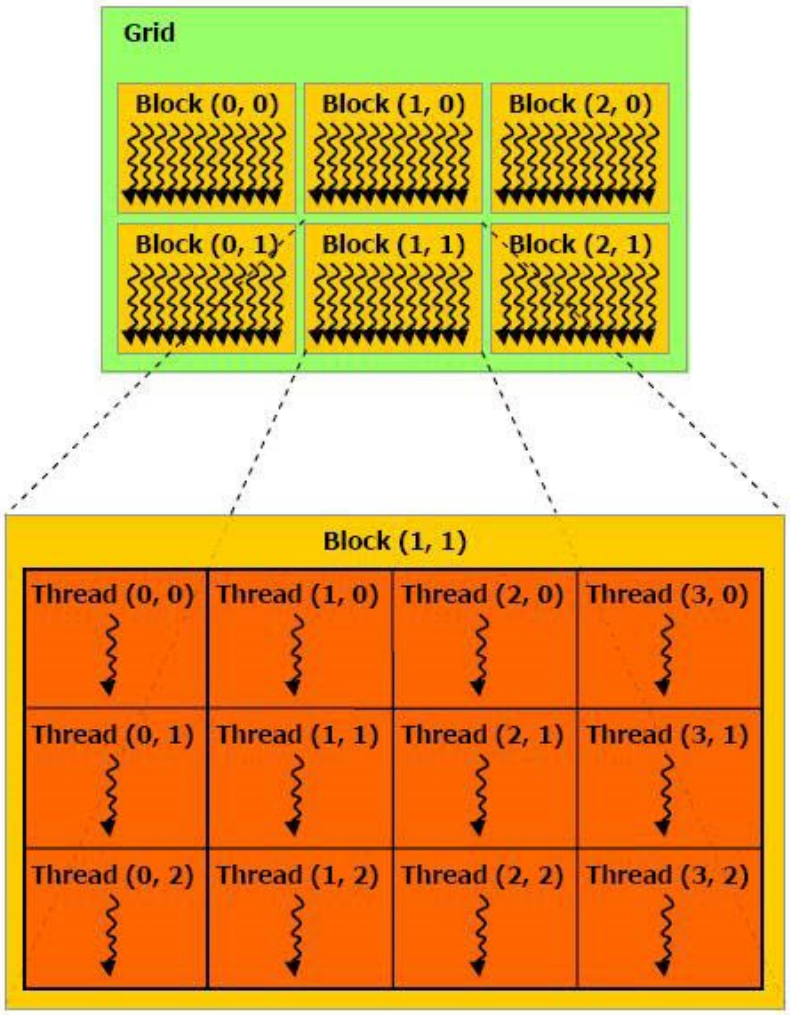
II.4.2.1 Struktur Kode CUDA

Terdapat dua perangkat yang berbeda untuk *runtime* CUDA, yaitu *host* (CPU), dan *device* (GPU). Masing-masing platform akan menjalankan bagian kode tersendiri. Ada dua bagian dalam kode program CUDA, yaitu *host code* dan *kernel code*. Ketika program berjalan, maka *host code* akan berjalan terlebih dahulu. Lalu, *kernel code* akan disalin dan dijalankan di memori GPU. Karena memori yang dapat diakses oleh masing-masing kode juga terpisah,

maka perlu dilakukan alokasi pada memori *device* dan transfer antar memori *device* dan memori *host*.

II.4.2.2 CUDA *Thread*

CUDA *thread* dibagi menjadi beberapa kelompok blok yang disebut *grid*. Masing-masing *grid* berisi beberapa buah blok. Di dalam blok terdapat beberapa *thread* yang akan berjalan terpisah. Struktur *thread* CUDA kurang lebih seperti pada Gambar II.8.



Gambar II.8: Organisasi *thread* CUDA (NVIDIA, 2017)

CUDA *thread* memiliki indeks, yang berdimensi 3. Indeks *thread* akan menunjukkan nomor blok dan nomor *thread* tersebut dalam blok. Menggunakan kedua indeks tersebut,

dapat dibentuk nomor yang unik tiap *thread*. Nomor unik tersebut dapat digunakan untuk melakukan pemrosesan paralel pada data besar. Hingga CUDA versi 8.0, maksimum dimensi blok *thread*, dimensi grid, dan jumlah *thread* tiap blok berturut-turut adalah 1024 x 1024 x 64, 65535 x 65535 x 65535, dan 1024.

II.4.2.3 *Warp* dan Percabangan

Thread akan dieksekusi bersama-sama dalam sebuah kumpulan yang disebut *warp*. *Thread* yang berjalan dalam satu *warp* akan mengeksekusi instruksi yang sama dalam satu waktu. Ketika ada percabangan dalam instruksi, maka *warp* tersebut akan berpisah. Ukuran maksimal *warp* saat ini yaitu 32 *thread*.

Karena percabangan mempengaruhi eksekusi *warp*, maka penempatan kondisi harus dilakukan secara hati-hati. Jika percabangan tidak dilakukan dengan benar dan menyebabkan banyak *thread* terpisah, maka akan banyak *warp* yang akan terbuang karena hanya mengeksekusi sedikit *thread*.

Pada GPU arsitektur Fermi, terdapat fitur yang disebut *memory coalescing*. *Memory coalescing* terjadi ketika semua *thread* pada *warp* yang sama mengakses memori yang kontigu. Fitur ini memungkinkan akses memori yang kontigu tersebut dilakukan dalam sekali *request*.

II.5 Penelitian Terkait

II.5.1 Rancangan *Multithreading* pada Pencocokan NIDS Berbasis *Signature*

Penelitian terkait optimasi berbasis *multithreading* pada NIDS telah dilakukan oleh Haagdorens et al. (2005). Tujuan optimasi yaitu untuk meningkatkan kapasitas pencocokan *signature*. Optimasi dilakukan berdasarkan pada beberapa strategi berikut:

1. *Thread asynchronous* untuk modul keluaran
2. Pencocokan *signature* secara *multithread*
3. Normalisasi konteks dan pencocokan *signature* secara *multithread*

4. Pencegahan *stateful*, normalisasi konteks dan pencocokan *signature* secara *multithread*

5. Pencegahan *stateful* secara *multithread* dengan normalisasi konteks dan pencocokan *signature* secara *multithread* terpisah

Hasil yang diperoleh dengan pengukuran pada prosesor Intel Xeon dengan *hyper-threading* menunjukkan bahwa desain 2 dan 3 memiliki kinerja yang rata-rata lebih baik dari pada desain lainnya. Pada desain 2 diperoleh penurunan *runtime* menjadi 94,1% dan 86,1%, untuk menggunakan *hyper-threading enabled* dan tidak. Sedangkan pada desain 3, diperoleh *runtime* 89% dan 84%. Sementara itu pada desain 4 dan 5 malah terjadi peningkatan *runtime* sebesar 120% hingga 150%. Hal ini disebabkan *latency* dari penggunaan struktur data yang rumit dan memerlukan banyak sinkronisasi.

II.5.2 Rancangan NIDS berbasis GPGPU

Selain rancangan *multithreading* pada CPU, telah dilakukan penelitian tentang rancangan arsitektur untuk NIDS pada GPGPU oleh Vasiliadis et al. (2008). Rancangan bertujuan mengakomodasi implementasi algoritma Aho-Corasick pada GPU. Implementasi berdasarkan pada NIDS Snort. Beberapa komponen yang dimodifikasi yaitu implementasi *state machine* dan skema transfer antara *host* dan *device*.

Implementasi menggunakan struktur memori untuk menyimpan pola *signature* yang berbentuk *list of string*. *Layout* ini kemudian dimodfikasi menjadi *array* 2D sebelum diimplementasi menggunakan CUDA. Kelebihan *layout* memori ini adalah mudah untuk ditransfer ke memori tekstur. Selain itu algoritma Aho-Corasick memanfaatkan prinsip *locality* dengan baik. Sehingga implementasi dengan memori tesktur dapat meningkatkan kinerja pencocokan NIPS hingga 19%.

Transfer data dari *host* ke *device* maupun *device* ke *host* menggunakan skema *double buffering*. Skema ini menggunakan sebuah *buffer* untuk menampung paket yang ingin dicek menjadi satu batch besar. Batch besar kemudian ditransfer ke memori *device*. Ketika GPU melakukan pemrosesan, maka paket yang baru datang akan ditampung dalam *buffer* berbeda. Sehingga kedua *buffer* dapat selalu bertukar posisi. Hal ini mampu meningkatkan utilitas pengiriman rata-rata paket.

Selain itu, memori *host* dapat dikonfigurasi menggunakan *pinned memory*. Penggunaan *pinned memory* membuat penempatan memori menjadi statik sehingga mengurangi tingkat *memory page swapping* dan juga meningkatkan *throughput* keseluruhan sistem.

Ada 2 desain pencocokan paket yang diteliti:

1. Pencocokan paralel paket tunggal

   Paket akan dibagi menjadi fragmen-fragmen dan disebar merata pada tiap *thread*. Kelebihan dari skema ini yaitu masing-masing *thread* mendapat bagian yang setara dan waktu yang diperlukan untuk pencocokan relatif sama sehingga tidak perlu saling menunggu. Sedangkan kekurangan dari skema ini adalah pencocokan akan mengalami *overlap* satu sama lain dan perlu penanganan khusus.

2. Pencocokan paralel multipaket

   Masing-masing paket akan didelegasikan pada satu *thread* pada *stream processor*. Kelebihan dari skema ini adalah pencocokan tidak akan mengalami *overlap*. Sedangkan kekurangan dari skema ini adalah masing-masing *thread* sering menunggu satu sama lain karena ukuran paket yang berbeda-beda.

Berdasarkan pengujian yang menggunakan 1000 pola acak, pencocokan dengan GPU mendapat *throughput* hingga 2,3 Gbps untuk paket rata-rata sebesar 1500 *byte*. Peningkatan yang berhasil dicapai sebesar 3,2 kali lipat dari *multithread* CPU. Spesifikasi lingkungan pengujian yaitu GPU NVIDIA GeForce 8600GT 1,2 GHz dengan 32 *stream processor* dalam 4 *stream multiprocessor* dengan memori 512 MB, dan CPU Intel Pentium 4 3,4 Ghz dengan memori 2 GB.

II.5.3 Rancangan Algoritma Pencocokan *Multiple-pattern* pada NIDS Berbasis GPU

Huang et al. (2008) mengajukan implementasi algoritma turunan Wu-Manber agar dapat memanfaatkan paralelitas pada GPU. Implementasi dilakukan dengan *shading language* yaitu Cg. Pencocokan akan memanfaatkan *fragment processor*. *Fragment processor* adalah pemroses yang bertugas melakukan operasi pemetaan tekstur terhadap fragmen masukan dan menuliskan hasilnya ke *rendering* target yang berupa *framebuffer* atau memori tekstur.

Tabel pola dan paket masukan disimpan pada memori tekstur. Penyimpanan dapat

berupa tabel dua atau tiga dimensi. Selain pola dan paket, informasi tentang *prefilter* dan paket kandidat juga disimpan di memori tekstur dalam bentuk *hash*. *Hash* berfungsi menggantikan *shift-byte table* pada algoritma Wu-Manber untuk lompat ke pola terdekat dan melewati karakter yang tidak perlu.

Hasil yang diperoleh yaitu peningkatan *throughput* hingga 3 kali dibandingkan implementasi Wu-Manber pada Snort. Struktur tekstur yang digunakan sangat berpengaruh terhadap latensi. Implementasi dengan representasi larik satu dimensi lebih cepat 1,5 kali lipat dibandingkan representasi tabel. Hal ini disebabkan struktur 2D memiliki banyak *branching overhead* karena *hash collision*.

II.5.4 Rancangan NIDS Berbasis *Signature* yang *Scalable* pada GPU

Dalam penelitiannya, dijelaskan oleh Jamshed et al. (2012) bahwa pada NIDPS, *bottleneck* utama terdapat pada 3 komponen, *packet acquisition*, *multi-string pattern matching*, dan pencocokan opsi *rule*. Pencocokan opsi *rule* hanya dilakukan saat paket terindikasi mengandung serangan oleh tahap *multistring pattern matching*.

Opsi dapat mengandung PCRE (*Perl Compatible Regular Expression*). Sebelum PCRE dicocokkan, PCRE akan dipreproses menjadi DFA terlebih dahulu menggunakan algoritma Thompson. Kemudian pencocokan opsi dapat dilakukan menggunakan metode yang serupa dengan pencocokan *multistring*.

Sehingga dalam penelitian ini, diajukan beberapa optimasi untuk pencocokan *rule option*. Salah satu optimasi yang diajukan untuk tahap pencocokan yaitu penggunaan *pipelining*. *Pipelining* dilakukan dengan memisahkan *thread* yang digunakan untuk operasi *I/O* dengan operasi analisis. Kemudian pada tiap operasi *I/O*, akan dilakukan pengumpulan paket dalam *batch* sebelum dilakukan transfer antar komponen. Implementasi dilakukan dengan menggunakan basis IDS Snort.

Pengujian dilakukan dengan konfigurasi CPU ganda Intel X5680 dengan 12 core dan dua GPU NVIDIA GTX 580. Hasil yang didapatkan menggunakan desain ini mampu mencapai 1,5 sampai 4 kali lipat kinerja Snort. *Latency* dapat menurun hingga 13 mikrodetik pada *offloading* paket awal untuk batch sebesar 1,5 kB. Pada pengujian dengan *traffic* jaringan, rancangan ini mampu mencapai *throughput* 25,2 Gbps pada *input* sebesar 40 Gbps.

II.5.5 Optimasi Algoritma Pencocokan Pola pada GPU

Salah-satu algoritma yang sering digunakan dalam deteksi pola adalah Aho-Corasick yang bersifat *multi-pattern string matching*. Dalam penelitian yang dilakukan Lin et al. (2013), dilakukan optimasi pada implementasi algoritma Aho-Corasick pada GPU. Implementasi memanfaatkan fitur-fitur pada GPU seperti *memory coalescing*, penghematan transaksi pada *global memory*, dan akses *shared memory* yang bebas *bank-conflict*.

Desain akan menggunakan pendekatan algoritma AC tanpa *failure function* yaitu PFAC (*Parallel Failureless Aho-Corasick*). Pendekatan ini dapat memaksimalkan penggunaan *thread* pada GPU tanpa menyebabkan *overlapping*. Transaksi ke *global memory* dapat dikurangi dengan memuat transisi *state* kosong ke *shared memory* dan penggunaan *texture memory* yang memanfaatkan *cache*.

Hasil pengujian dengan CPU Intel Core i7-950 dan GPU NVIDIA GTX 580 menunjukkan peningkatan yang signifikan. Optimasi dengan PFAC pada *multithread* CPU mencapai 7 kali lipat dari AC. Sedangkan PFAC pada GPU dapat mencapai 10 kali lipat daripada AC pada CPU. Masukan didapatkan dari dataset *traffic* kompetisi DEFCON.

II.5.6 Penyimpanan *Signature* dengan Struktur *Trie* Terkompresi

Penyimpanan *signature* untuk NIDS dapat dilakukan dengan struktur *trie* terkompresi. Desain utama *trie* menggunakan representasi penelusuran *breadth first search* (BFS). Lokasi *pointer* simpul anak akan dihitung menggunakan *sparse matrix*. Selain itu dilakukan kompresi terhadap bentuk *trie* sehingga memori yang dibutuhkan lebih padat (Bellekens et al., 2014).

Sparse matrix adalah matriks yang berisi *offset* dari lokasi simpul anak terkecil dan *bitmap* yang berisi flag dari semua simpul anak dalam 256 karakter ASCII. Ketika ingin melakukan penelusuran, simpul akan melakukan penghitungan karakter berdasarkan offset pertama simpul ditambah dengan kumulatif dari tiap bit. Hasil yang diperoleh adalah indeks dari simpul anak berikutnya.

Optimasi berikutnya yaitu kompresi pada sufiks kalimat. Kompresi dilakukan dalam dua tahap: penggabungan pada huruf terakhir dan tiga huruf terakhir yang sama. Huruf

terakhir dapat digabungkan karena simpul tersebut tidak memiliki anak lagi. Dan tiap simpul tersebut pasti akan menuju ke penanda akhir string. Lalu untuk tahap kedua, akan dihitung sufiks dari pola. Tiap sufiks pada tiga karakter terakhir yang sama akan digabungkan.

BAB III

ANALISIS PERMASALAHAN DAN SOLUSI

III.1 Analisis Permasalahan

Dampak dari pengecekan paket adalah adanya *overhead* waktu respons. Lama *overhead* tergantung dari waktu pencocokan pola. Pada sistem pencegah intrusi jaringan (NIPS), hal ini dapat membuat hasil analisis menjadi tidak akurat. Maka perlu adanya metode untuk mempercepat proses analisis. Sebagai perbandingan, *bandwidth* jaringan US Naval Postgraduate School sudah mencapai 20 Gbps dengan traffic rata-rata sebesar 200 Mbps per hari. Sedangkan maksimum paket yang dapat dianalisis secara serial oleh Snort tidak lebih besar dari 300 Mbps (Albin and Rowe, 2012).

Beberapa penelitian tentang teknik mempercepat NIDS dan NIPS telah dilakukan. Desain paling awal yaitu menggunakan desain konkuren dengan *multithreading* pada CPU (Haagdorens et al., 2005). Desain ini mampu meningkatkan penggunaan utilitas *thread* CPU secara drastis. Kemudian desain berbeda yang menggunakan GPU mulai diajukan oleh Vasiliadis et al. (2008). Hasil yang didapat mampu mempercepat sistem hingga 5x dibandingkan CPU dengan harga yang sepadan (Smith et al., 2009).

Berdasarkan pengukuran yang dilakukan oleh Jamshed et al. (2012), didapatkan bahwa sebagian besar beban analisa paket berada pada tahap pencarian string pada *payload* paket. Pada tahapan ini, sebagian besar paket yang tidak terindikasi sebagai serangan akan diloloskan. Sehingga beban pencocokan pada tahap berikutnya, yaitu pencocokan *option rule* akan berkurang drastis. Maka, fokus dari solusi yang akan diajukan yaitu implementasi desain yang akan mempercepat kinerja pencocokan *string* paket pada NIDS Snort menggunakan GPU.

Sampai saat ini ada beberapa penelitian terkait pencocokan *string* menggunakan GPU. Vasiliadis et al. (2008) merancang sebuah arsitektur untuk mengakomodasi algoritma Aho-Corasick pada GPU. Implementasi dilakukan pada NIDS Snort menggunakan *state machine*

yang berbentuk tabel 2D untuk memaksimalkan *locality* dan skema transfer antara *host* dan *device* menggunakan *double buffering*. Sedangkan *buffer* juga dialokasi dengan *pinned memory*. Desain ini meningkatkan *throughput* rata-rata sistem sebesar 3 kali lipat.

Kemudian, Jamshed et al. (2012) mengajukan solusi untuk mengurangi ketergantungan antar komponen dengan menggunakan *pipelining*. *Thread* untuk operasi analisis dipisahkan dengan *thread* untuk operasi *I/O*. Menggunakan solusi ini *throughput* rata-rata sistem meningkat hingga 4 kali lipat.

Sedangkan Lin et al. (2013) menemukan bahwa selain *I/O*, sumber latensi berikutnya yaitu akses memori. Operasi pencocokan *string* merupakan operasi yang *memory-bound*. Rancangan sebelumnya menyimpan keseluruhan kamus pada memori global. Hal ini diatas dengan penggunaan *shared memory* secara efektif dan kombinasi dengan *texture memory*. Selain itu, terdapat modifikasi pada implementasi Aho-Corasick yang memaksimalkan penggunaan *thread* pada GPU. Hasil eksperimen menunjukkan peningkatan *throughput* hingga sebesar 10 kali lipat.

III.2 Analisis Solusi

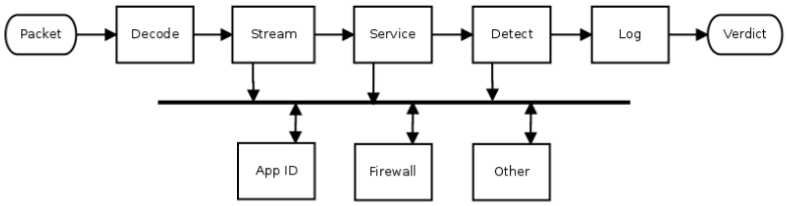
Eksperimen penggunaan GPGPU pada pencocokan string NIDS akan memodifikasi NIDPS Snort. Snort digunakan karena kode bersifat *open-source* dan mudah untuk dikembangkan, sehingga pengembangan dan eksperimen tidak memerlukan banyak biaya dan dapat dieksplorasi secara mandiri.

Pengembangan akan menggunakan platform GPGPU CUDA. Platform CUDA merupakan platform GPGPU yang dibuat pada GPU NVIDIA. Platform CUDA digunakan karena saat ini platform CUDA lebih matang daripada OpenCL. Selain itu, dokumentasi dan contoh CUDA lebih banyak dan API yang lebih sederhana dan spesifik dibanding OpenCL.

III.2.1 Struktur IDS Snort

Komponen utama yang akan dioptimasi yaitu pencocokan string yang dilakukan setelah preproses. Pencocokan string dilakukan dengan mengimplementasi *interface* MPSE (*multipattern search engine*) yang terdapat pada modul *detect*. Modul ini dipanggil ketika *rule* selesai dikelompokkan berdasarkan servis dan protokol seperti terlihat pada

Gambar III.1 di bawah. Tiap *rule group* akan memiliki *instance* MPSE sendiri dan membentuk kamus sesuai *rule* yang ada pada tiap *rule group*.



Gambar III.1: Arsitektur Snort versi 3. Pencocokan dilakukan pada bagian Detect

Setelah pembentukan *instance* dilakukan, maka input akan di-*streaming*, dipreproses, dibagi berdasarkan *rule group*, kemudian disuplai ke *instance* MPSE yang terkait. Hasil pencocokan paket akan dikumpulkan dalam *reducer* untuk kemudian dibandingkan dengan *policy* apakah dianggap paket berbahaya atau tidak.

III.2.2 Algoritma Pencocokan *Signature*

Komponen utama yang akan dioptimasi yaitu pencocokan string. Diantara pencocokan pola yang digunakan pada Snort, versi Aho-Corasick memiliki runtime yang paling cepat menggunakan teknik *multithreading* pada CPU. Idealnya, modul yang ingin dikembangkan akan melampaui kinerja dari Snort yang menggunakan algoritma Aho-Corasick ini.

Berdasarkan pengujian eksperimen yang dilakukan oleh Lin et al. (2013), implementasi langsung dari algoritma ini tidak memiliki keunggulan yang signifikan pada GPU. Salah satu kendala dari algoritma AC (Aho-Corasick) adalah *boundary matching problem*. Masalah ini bisa diatasi dengan menambahkan jangkauan pencarian sebesar pola terpanjang. Namun, efek dari trik ini adalah peningkatan kompleksitas yang cukup drastis dari $O(n)$ menjadi $O(n+ms)$.

Solusi yang diajukan oleh Lin et al. (2013) yaitu dengan menggunakan algoritma PFAC (*Paralel Failureless Aho-Corasick*). PFAC adalah varian dari AC yang memulai pencocokan *thread* pada tiap karakter dan tidak menggunakan *failure transition*. Ada beberapa keuntungan dari algoritma ini:

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.

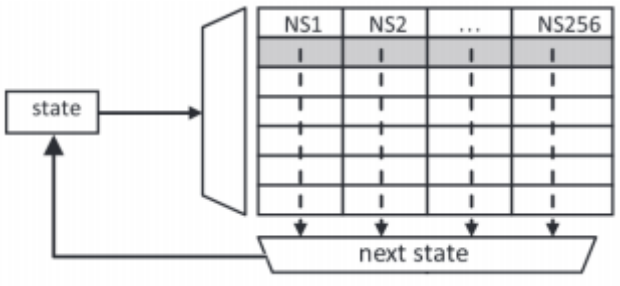
2. Dalam mesin PFAC, tiap 32 *thread* dalam *warp* akan mengakses memori yang berurutan dari memori global. Dengan demikian, akses tabel transisi fitur *memory coalescing* dapat dimanfaatkan.

3. Tidak memerlukan penyimpanan tambahan untuk *failure transition*. Sehingga konsumsi memori akan lebih sedikit dan kemungkinan *cache hit* lebih besar.

III.2.3 Struktur Penyimpanan Kamus

Implementasi dari mesin PFAC yang digunakan ada beberapa macam. Salah satu model yang dapat digunakan yaitu dengan bentuk seperti pohon dengan *list of pointer* ke status anak. Namun, model yang dialokasi secara dinamis tidak cocok digunakan dalam GPU karena alokasi pada GPU memiliki latensi yang tinggi. Selain itu, lokasi memori dapat menjadi tidak kontigu dan tidak mendukung adanya *spatial locality*.

Opsi lainnya yaitu dengan tabel transisi dua dimensi. Untuk tiap status, akan dibuat daftar status transisi terhadap tiap karakter. Model ini lebih sederhana dan juga mendukung adanya *memory coalescing*. Tabel akan dibentuk di *host memory*. Setelah semua pola didaftarkan, tabel akan disalin ke GPU hanya sebesar ukuran tabel yang terisi agar tidak boros memori. Tiap baris menunjukkan transisi dari tiap status yang bukan status akhir dan memiliki 256 kolom. Tabel transisi akan berbentuk seperti Gambar III.2 di bawah.



Gambar III.2: Tabel transisi dari *state machine* kamus

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.

Penghematan memori lebih jauh dapat dilakukan dengan menerapkan skema kompresi tabel menjadi *bitmap*. Namun, keuntungan dari penggunaan memori tekstur akan terhambat. Dibutuhkan komputasi tambahan untuk melakukan *lookup fetch* dari memori tekstur. Sehingga teknik ini tidak digunakan pada rancangan modul.

III.2.4 Alokasi *Thread*

Seperti dijelaskan pada bagian III.2.1, *thread* akan menelusuri *state machine* yang sama dan berhenti ketika tidak ada transisi yang valid. Sehingga, banyak diantara *thread* yang akan berhenti sangat awal. Untuk menunjang *stream processor*, Lin et al. (2013) 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, diharapkan kemungkinan ketimpangan muatan dari masing-masing *thread* akan berkurang. Pengujian yang dilakukan oleh Lin et al. (2013) menunjukkan hasil serupa. Sehingga, skema inilah yang akan digunakan dalam implementasi kode *kernel*.

III.2.5 Optimasi Latensi pada GPU

Pencocokan pola dengan tabel transisi adalah aplikasi yang *memory-bound*. Penyimpanan pola pada memori global adalah salah satu sumber latensi. Salah satu optimasi yang dapat digunakan yaitu menyimpan pola pada memori tekstur. Memori tesktur dapat digunakan untuk mengurangi latensi pada *state* yang berdekatan atau sering diakses. Berdasarkan Lin et al. (2013), penggunaan memori tekstur dapat menurunkan latensi hingga 12%. Lebih lanjut, dapat dilakukan pemuatan baris pertama tabel transisi, yaitu baris milik status awal, ke dalam *shared memory* karena baris ini yang pasti akan digunakan semua *thread*. Peningkatan dari memuat baris pertama ke *shared memory* akan berdampak besar ke

latensi sistem.

Kemudian sumber latensi kedua adalah mengambil karakter dari *stream* masukan. Ketika pencocokan dilakukan, *thread* yang bersebelahan akan mengakses karakter dalam *stream input* yang sebelumnya diakses oleh *thread* lain. Berdasarkan skenario ini, Lin et al. (2013) mengusulkan untuk menyalin string masukan ke *shared memory* terlebih dahulu. Selain itu untuk mencegah masukan pada memori global terkena *swap*, maka *stream buffer* akan menggunakan *pinned memory*.

III.3 Rancangan Solusi

Pembahasan mengenai rancangan solusi dibagi menjadi 3 bagian. Bagian pertama dibahas mengenai gambaran umum solusi. Bagian kedua dibahas mengenai kebutuhan perangkat lunak. Bagian ketiga dibahas arsitektur perangkat lunak.

III.3.1 Gambaran Umum Solusi

Berdasarkan analisis yang telah dilakukan, NIDS akan dikembangkan dari NIDS Snort dengan beberapa metode. Metode pengembangan akan mengacu pada implementasi Lin et al. (2013) serta sedikit penyesuaian untuk berdasarkan Vasiliadis et al. (2008).

Secara garis besar, terdapat 4 tahapan yang harus dilakukan dalam modul pencocokan pola yang akan dibangun:

1. Pembentukan tabel transisi

   Pembentukan tabel transisi akan dilakukan saat persiapan pencocokan. Struktur tabel dan operasi pembentukan didasarkan pada Lin et al. (2013). Struktur akan menggunakan tabel transisi 2D dengan struktur internal larik 1D. Pembentukan dilakukan pada *host* kemudian baru akan ditransfer ke *device*. Pembentukan tabel transisi dilakukan dengan mengumpulkan pola terlebih dahulu. Setelah pola terkumpul, lalu *state* akan disusun kembali agar *state* akhir berada di bagian paling awal tabel. Sehingga *state* selain *state* akhir dimulai dari N+1 dengan N adalah jumlah pola.

2. Persiapan tabel transisi dalam *device memory*

Setelah tabel transisi dibentuk, tabel akan dikirimkan ke *device memory*. Penyimpanan pola akan diikat pada memori tekstur sesuai Lin et al. (2013). Selain itu juga disiapkan tabel untuk menampung hasil pencocokan. Tabel akan dialokasikan sebesar jumlah *thread* yang digunakan dalam pencocokan.

3. Penampungan *stream* masukan

   Stream input akan ditampung dalam *secondary buffer* hingga sebesar 256 kB. *Buffer* akan dialokasi dengan *pinned memory* untuk memungkinkan DMA (*direct memory access*) dari memori *host* ke *device* (Vasiliadis et al., 2008). Kemudian setelah *buffer* mendekati penuh, *stream* akan ditransfer ke memori *device*.

4. Penelusuran automata

   Pencocokan dilakukan dengan penelusuran mesin PFAC yang sudah dibentuk. *Stream* masukan akan dimuat pada *shared memory*. Tiap *thread* akan memulai pencocokan pada satu karakter pada *stream* masukan. Satu *thread* melakukan pencocokan sebanyak 4 kali untuk mengurangi kemungkinan ketimpangan beban (Lin et al., 2013). Penelusuran dilakukan hingga mencapai state akhir. Jika state akhir bukan merupakan *state dummy*, maka hasil pencocokan akan diset pada tabel hasil yang berada di *global memory*.

Masing-masing akan dilakukan

III.3.2 Kebutuhan Perangkat Lunak

Berdasarkan analisis yang dilakukan sebelumnya, modul yang akan dibangun setidaknya memiliki spesifikasi kebutuhan fungsional seperti yang dijabarkan dalam Tabel III.1.

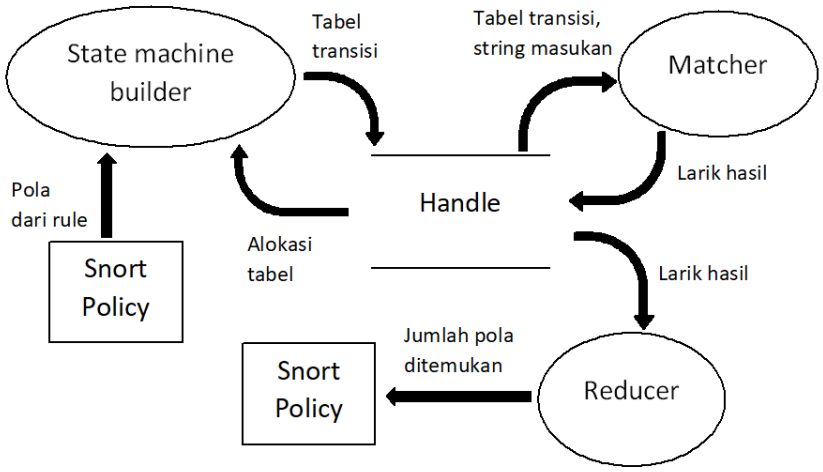
Tabel III.1: Kebutuhan Fungsional

No	Kebutuhan Fungsional
1	Mampu membaca paket dari berkas PCAP
2	Mampu membuat struktur automata untuk pencocokan paket
3	Mampu menjalankan pencocokan pada GPU
4	Mampu menghasilkan *alert* pada paket yang terindikasi berbahaya

III.3.3 Arsitektur Perangkat Lunak

Arsitektur yang digunakan akan menggunakan arsitektur dasar Snort. Dalam mengimplementasikan modul pencocokan, Snort menyediakan API yang dapat diimplementasi dengan mudah. Sehingga dalam penggabungan modul tersebut, hanya perlu dibuat *wrapper* untuk memanggil modul yang telah diimplementasi secara terpisah.

Modul akan terdiri dari beberapa komponen. Komponen-komponen tersebut akan tersusun menjadi sebuah modul yang akan terintegrasi pada Snort seperti pada Gambar III.3. Ada beberapa komponen utama dalam modul pencocokan pola:



Gambar III.3: Arsitektur modul

1. *State machine builder*

   State machine builder melakukan pembangkitan kamus dari *ruleset*. Pola akan ditambahkan dalam bentuk *list of string* untuk kemudian ditampung dalam *temporary table*. Kemudian *temporary table* akan diiterasi untuk membentuk kamus. Kamus yang dibentuk menggunakan struktur tabel 2D dengan baris sebagai *state* dan kolom sebagai status transisi. Setiap baris memiliki kolom sebesar 256 *byte*.

2. *Matcher*

   Matcher akan menyalin *stream* dari *host buffer* ke *device buffer*. Kemudian *stream* dari *buffer* akan disalin ke *shared memory*. Setelah *stream* siap, *matcher* akan melakukan pencocokan dengan menelusuri kamus secara paralel berdasarkan tiap

karakter.

3. *Reducer*

   Reducer akan menggabungkan hasil pencocokan dari tiap *byte* sehingga diperoleh jumlah total rule yang terdeteksi. Jumlah ini akan dibandingkan dengan *policy* oleh Snort untuk menentukan apakah *stream* dianggap berbahaya atau tidak. *Reducing* akan dilakukan secara sekuensial pada CPU.

BAB IV

IMPLEMENTASI DAN PENGUJIAN

Pembahasan pada bab ini akan dibagi menjadi dua bagian. Bagian pertama dipaparkan tentang lingkungan dan detail implementasi. Sedangkan bagian kedua dibahas mengenai teknis dan hasil pengujian terhadap perangkat lunak yang sudah dibangun.

IV.1 Implementasi

Bagian implementasi dibagi menjadi tiga bagian. Bagian pertama dibahas mengenai lingkungan implementasi. Bagian kedua dibahas mengenai batasan implementasi. Bagian terakhir dibahas mengenai spesifikasi komponen.

IV.1.1 Lingkungan Implementasi

Dalam proses implementasi digunakan sebuah komputer dan sejumlah perangkat lunak untuk menyelesaikan perangkat lunak yang dibangun. Detil spesifikasi perangkat keras dan lunak yang digunakan adalah seperti pada Tabel IV.1.

Tabel IV.1: Spesifikasi Lingkungan Implementasi

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

IV.1.2 Batasan Implementasi

Implementasi dilakukan dengan batasan-batasan sebagai berikut.

1. Implementasi hanya mengerjakan komponen MPSE (*multipattern search engine*) pada Snort
2. Pencocokan menggunakan satu *instance* GPU
3. Pencocokan hanya dilakukan pada satu *thread kernel code*
4. Akuisisi *payload* hanya dilakukan dari berkas PCAP

IV.1.3 Spesifikasi Komponen

Berdasarkan analisis pada Bab III, implementasi akan berbasis implementasi Lin et al. (2013) dengan skema *pinned memory buffer* seperti pada Vasiliadis et al. (2008). Implementasi dilakukan dengan membuat beberapa komponen, yaitu:

1. *Handle*

   Handle merupakan struktur data yang membungkus fungsionalitas pencocokan untuk dapat digunakan pada *pipeline* Snort. *Handle* berisi daftar pola, *buffer* masukan, tabel transisi (sisi *host* maupun *device*), jumlah *state* akhir dan *list* hasil pencocokan. *Buffer* masukan hanya dialokasi sekali sebesar 256 MB

   Handle juga berisi fungsi-fungsi yang terkait modul yang dikembangkan seperti menambahkan pola, pembentukan kamus, pencocokan, dan mencetak informasi spesifik *handle*. Fungsi-fungsi pada *handle* akan memanggil modul lain yang terkait, misalnya fungsi pembentukan kamus akan memanggil komponen pembentukan kamus pada kode PFAC yang dibuat.

2. *State machine builder*

   State machine builder merupakan modul yang digunakan membentuk kamus untuk pencocokan. *State machine* dibentuk di memori *host*. Pola dikumpulkan dahulu dalam sebuah *list*. Kemudian tiap pola akan dilakukan penelusuran DFS (*depth-first search*) untuk membangun transisi tiap *state*.

   Dalam penelusuran, simpul baru akan terbentuk ketika simpul hasil transisi karakter tidak terdefinisi (bernilai *null*). Untuk mengurangi *lookup* ke *global memory*, *final state* tidak disebutkan secara eksplisit. Melainkan, dengan mengubah susunan nomor *state* sehingga *state* akhir berada pada nomor awal. Sehingga pengecekan

hanya tinggal membandingkan nomor *state* dengan jumlah *state* akhir.

3. *Matcher wrapper*

   Matcher wrapper adalah *wrapper* dari *kernel* sebelum dijalankan. *Wrapper* akan menyiapkan *payload* masukan sebelum *kernel* dijalankan. Masukan disalin dari *host buffer* ke *device buffer*. Setelah persiapan selesai, *wrapper* akan memanggil kode *kernel* untuk menjalankan pencocokan. Setelah pencocokan selesai dilakukan, *wrapper* akan menyalin *array* hasil dari *device* ke *host*. *Array* hasil tadi akan dikirim ke *reducer* untuk digabungkan.

```
1  __global__ void TraceTable_kernel(char *d_input_string, int
2  *d_match_result, int input_size, int initial_state){
3     int start = blockIdx.x * blockDim.x + threadIdx.x;
4     int pos;
5     int state;
6     int inputChar;
7     int match;
8
9     for ( ; start < input_size; start = start + gridDim.x *
10 blockDim.x){
11        state = initial_state;
12        pos = start;
13        match = 0;
14        while ( pos < input_size ){
15           inputChar = d_input_string[pos];
16           state = tex2D(tex_PFAC_table, inputChar, state);
17           if (state == trap_state ){ break ; }
18           if(state < initial_state){
19              match = state;
20           }
21           pos = pos + 1;
22           if ( pos >= input_size ) { break ; }
23         }
24      d_match_result[start] = match;
25    }
26 }
```

Gambar IV.1: *Pseudocoode kernel*

4. *Kernel code*

   Kernel adalah kode GPU yang akan dijalankan untuk pencocokan. *Kernel* hanya akan melakukan *looping* terhadap nomor *state* berdasarkan tabel transisi yang telah dimuat. *Looping* akan berhenti ketika *state* telah menuju *state* akhir maupun status *dummy*. Kemudian *state* akhir akan ditulis ke *output array*. *Buffer* akan dibagi menjadi 4 bagian sebagai titik mulai pencocokan. Sehingga tiap *thread* akan mencocokkan sebanyak 4 kali. *Pseudocode* dari kode *kernel* akan terlihat seperti pada Gambar IV.1 di atas.

5. *Reducer*

   Reducer akan menggabungkan hasil pencocokan dari tiap *thread* sehingga diperoleh total kumpulan pola yang terdeteksi. *Reducer* dipanggil setelah *matcher* dijalankan. *Reducing* akan dilakukan secara sekuensial pada CPU. Hasil *reducing* akan dikembalikan ke fungsi *search* milik *handle* untuk dikembalikan lagi ke Snort dalam bentuk jumlah.

IV.2 Pengujian

IV.2.1 Tujuan 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. *Live testing* tidak diujikan dalam Tugas Akhir ini mengingat kapasitas *network interface* dapat menjadi *bottleneck* sehingga hasil perbandingan tidak akurat.

IV.2.2 Skenario Pengujian

Pengujian dilakukan untuk mengukur peningkatan kinerja deteksi Snort. Skenario yang akan diujikan meliputi strategi-strategi yang dibahas pada Subbab III.2. Berikut ini adalah tipe-tipe skenario yang diuji dalam pengujian 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.

Kelima skenario akan diuji dengan ukuran *buffer* berbeda, yaitu 128 kB (bawaan), 512 kB, 1024 kB, dan 2018 kB.

IV.2.3 Lingkungan Pengujian

Pengujian dilakukan pada komputer dengan spesifikasi seperti pada Tabel IV.2 berikut.

Tabel IV.2: 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

IV.2.4 Hasil Pengujian

Berikut merupakan hasil pengujian dari seluruh skenario di atas. *Ruleset* yang digunakan yaitu Snort VRT *ruleset* versi 3000. Sedangkan berkas masukan didapatkan dari *log traffic* kompetisi DEFCON 25 yang berlangsung pada bulan Juli tahun 2017. Dataset didapatkan dari jaringan *torrent* milik arsip DEFCON berupa beberapa berkas PCAP sebesar sekitar 2,6 GB. Berikut adalah hasil pengujian pada ukuran *buffer* berbeda.

Tabel IV.3: Hasil pengujian

Skenario	*Buffer* (kB)	Runtime (detik)	Speedup
Baseline	128	94.632	1
	512	92.656	
	1024	90.118	
	2048	96.097	
Skenario 1	128	450.629	0.21
	512	463.281	0.2
	1024	500.656	0.18
	2048	436.805	0.22
Skenario 2	128	29.207	3.4
	512	39.321	3.16
	1024	31.29	2.88
	2048	31.302	3.07
Skenario 3	128	31.44	3.01
	512	38.446	2.41
	1024	35.202	2.56
	2048	31.404	3.06
Skenario 4	128	30.825	3.07
	512	29.137	3.18
	1024	22.417	4.02
	2048	20.446	4.7

IV.2.5 Analisis Hasil Pengujian

Dari hasil pengujian, didapatkan bahwa hasil skenario 1 memilki kapasitas paling rendah. Alasannya yaitu karena operasi pencocokan *string* adalah operasi yang *memory-bound*. Diperlukan banyak akses ke penyimpanan kamus untuk mendapatkan transisi ke *next state*. Karena kamus disimpan di *global memory*, maka *fetch* ke *global memory* menjadi sering dan menimbulkan latensi yang cukup besar.

Skenario 2 secara umum lebih cepat beberapa kali daripada skenario 1, yaitu sekitar 15 kali lipat lebih cepat. Hal ini karena kecepatan akses *global memory* lebih besar dari *shared memory*. Dalam skenario ini, *input string* seharusnya tidak berpengaruh banyak karena adanya *coalescing access*. Sehingga pemuatan baris transisi dari *state* awal ke *shared memory* memiliki pengaruh yang besar terhadap *throughput* sistem. Ini menunjukkan bahwa mayoritas *thread* akan berhenti pada awal pencocokan.

Skenario 3 tidak terlalu berpengaruh terhadap skenario 2. Keuntungan *pinned memory* tidak terlalu terlihat bahkan cenderung memperlambat. Sejalan dengan eksperimen yang

dilakukan Vasiliadis et al. (2008). Ini karena *pinned memory* memiliki *overhead* saat alokasi dan dealokasi. Kemungkinan besar ini terjadi karena ukuran *buffer* belum terlalu besar. *Overhead* saat alokasi masih belum sebanding dengan perbedaan *transfer* antara DMA dan melalui CPU. Selain itu, juga tidak terjadi *swap* terhadap *page* yang terdapat *buffer* karena ukuran memori masih kecil.

Sedangkan skenario 4 baru terlihat perbedaan signifikan terhadap skenario 2 dan 3 pada *buffer* sebesar 1 MB ke atas. Terlihat bahwa adanya *cache* pada memori tekstur membantu menurunkan akses ke memori global dan meningkatkan kinerja keseluruhan modul.

BAB V

SIMPULAN DAN SARAN

V.1 Simpulan

Beberapa kesimpulan yang didapat dari pengerjaan tugas akhir ini adalah sebagai berikut.

1. Peningkatan kinerja pencocokan string pada IDS Snort menggunakan GPU berhasil dilakukan dengan cara mengubah pendekatan algoritma yang digunakan agar memaksimalkan utilisasi GPU.

2. Implementasi dilakukan dengan berbasis pada rancangan Lin et al. (2013), yaitu dengan Aho-Corasick varian tanpa *failure function* dan delegasi *thread* per karakter. Teknik ini memanfaatkan kemampuan GPU untuk membangkitakan banyak *thread* serta *memory coalescing* ketika mengakses karakter yang bersebelahan dalam *stream*.

3. Operasi pencocokan string dari kamus merupakan operasi yang *memory bound* sehingga dilakukan beberapa teknik untuk mengurangi *latency gap* antara akses I/O, memori dan komputasi GPU. Di antara teknik yang digunakan yaitu penggunaan *pinned memory* pada *buffer input* dan *texture memory* pada kamus. Selain itu, digunakan juga penggunaan *shared memory* untuk menampung *buffer input* dan tabel transisi dari *state* awal untuk menghemat *fetch* pada memori global.

4. Parametrisasi dapat dilakukan pada ukuran *buffer input*. Dengan menambah ukuran *buffer input*, jumlah pengiriman paket dari *host* ke *device* dapat berkurang dan dapat menurunkan latensi secara drastis.

V.2 Saran

Pengembangan yang dapat dilakukan selanjutnya terkait tugas akhir ini adalah sebagai berikut:

1. Pencocokan dilakukan oleh beberapa MPSE *instance* secara simultan dengan memanfaatkan *semaphore*.
2. Melakukan penghematan transfer memori dari *host* ke *device* dengan hanya menggunakan satu struktur data.
3. Penggunaan CUDA *stream* untuk melakukan pengiriman memori dari *host* ke *device* maupun sebaliknya secara *asynchronous* sehingga meningkatkan utilisasi komputasi GPU.
4. Jumlah alokasi yang dilakukan dapat dikurangi dan memori dapat di-*reuse* untuk menghindari *memory leak*.
5. Variasi jumlah *thread* dalam *block* dan ukuran *shared memory*.

Daftar Pustaka

Aho, A. V. and Corasick, M. J. (1975). Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340.

Albin, E. and Rowe, N. C. (2012). A Realistic Experimental Comparison of the Suricata and Snort Intrusion-Detection Systems. In *Proceedings of the 2012 26th International Conference on Advanced Information Networking and Applications Workshops*, WAINA '12, pages 122–127, Washington, DC, USA. IEEE Computer Society.

Azimi, E., Ghaznavi-Ghoushchi, M. B., and Rahmani, A. M. (2009). Implementation of Simple SNORT Processor for Efficient Intrusion Detection Systems. In *2009 IEEE International Conference on Intelligent Computing and Intelligent Systems*, volume 3, pages 533–537.

Bellekens, X., Tachtatzis, C., Atkinson, R., Renfrew, C., and Kirkham, T. (2014). A Highly-Efficient Memory-Compression Scheme for GPU-Accelerated Intrusion Detection Systems. *The 7th International Conference of Security of Information and Networks, SIN 2014, Glasgow, UK, September, 2014*, pages 302–309.

Das, A., Nguyen, D., Zambreno, J., Memik, G., and Choudhary, A. (2008). An FPGA-Based Network Intrusion Detection Architecture. *IEEE Transactions on Information Forensics and Security*, 3(1):118–132.

Evans, D., Bond, P., and Bement, A. (2003). *Standards for Security Categorization of Federal Information and Information Systems*. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

Haagdorens, B., Vermeiren, T., and Goossens, M. (2005). Improving the Performance of Signature-based Network Intrusion Detection Sensors by Multithreading. In *Proceedings of the 5th International Conference on Information Security Applications*, WISA'04, pages 188–203, Berlin, Heidelberg. Springer-Verlag.

Huang, N. F., Hung, H. W., Lai, S. H., Chu, Y. M., and Tsai, W. Y. (2008). A GPU-Based

Multiple-Pattern Matching Algorithm for Network Intrusion Detection Systems. In *22nd International Conference on Advanced Information Networking and Applications - Workshops (Aina Workshops 2008)*, pages 62–67.

Jamshed, M. A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., and Park, K. (2012). Kargus: A Highly-scalable Software-based Intrusion Detection System. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 317–328, New York, NY, USA. ACM.

Kizza, J. M. (2015). *Guide to Computer Network Security*. Springer Publishing Company, Incorporated, 3rd edition.

Lin, C.-H., Liu, C.-H., Chien, L.-S., and Chang, S.-C. (2013). Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *IEEE Trans. Comput.*, 62(10):1906–1916.

Lindholm, E., Kilgard, M. J., and Moreton, H. (2001). A User-programmable Vertex Engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 149–158, New York, NY, USA. ACM.

Mitra, A., Najjar, W., and Bhuyan, L. (2007). Compiling PCRE to FPGA for Accelerating SNORT IDS. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '07, pages 127–136, New York, NY, USA. ACM.

NVIDIA (2017). CUDA Toolkit Documentation. *NVIDIA Developer Zone - CUDA C Programming Guide v8.0*. Section 3.1.5.

OWASP (2017). OWASP Top 10 Application Security Risks - 2017.

Parker, D. B. (1998). *Fighting Computer Crime: a New Framework for Protecting Information*. John Wiley & Sons.

Scarfone, K. A. and Mell, P. M. (2007). SP 800-94. Guide to Intrusion Detection and Prevention Systems (IDPS). Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States.

Smith, R., Goyal, N., Ormont, J., Sankaralingam, K., and Estan, C. (2009). Evaluating GPUs for Network Packet Signature Matching. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 175–184.

Vasiliadis, G., Antonatos, S., Polychronakis, M., Markatos, E. P., and Ioannidis, S. (2008). Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Recent Advances in Intrusion Detection*, pages 116–134. Springer, Berlin, Heidelberg.