Analisis Performa Paralelisasi Operasi Kriptografi terhadap Javascript sebagai Single-threaded Language

Arung Agamani Budi Putera - 13518005[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13518005@std.stei.itb.ac.id

***Abstraksi*—Operasi yang melibatkan kriptografi seringkali membutuhkan waktu yang banyak akibat operasi matematika yang berat. Salah satu cara untuk melakukan optimasi adalah dengan menerapkan *parallel processing*. Namun, tidak semua bahasa pemrograman memiliki dukungan terhadap *parallel processing*, seperti Javascript. Kendati demikian, Javascript memiliki metode agar bisa menerapkan *parallel processing*. Makalah ini akan menganalisis performa yang diberikan oleh Javascript dalam melakukan pemrosesan operasi kriptografi baik secara sekuensial maupun parallel.**

***Keywords*—Kriptografi, Javascript, Single-threaded, Multithreading, Worker.**

I. PENDAHULUAN

Operasi yang menggunakan metode kriptografi seringkali menggunakan sumber daya yang besar dengan banyaknya operasi matematika yang berat. Operasi matematika ini tidak hanya dilakukan terhadap angka-angka dengan jumlah digit yang tidak sedikit (lebih dari 10), namun juga menggunakan operasi matematika yang pada dasarnya sudah "mahal", yakni operasi perkalian, perpangkatan, pembagian, modulo, maupun invers modulo. Hal inilah yang menjadi salah satu tolak ukur untuk tingkat keamanan yang dimiliki oleh suatu algoritma kriptografi.

Adanya tuntutan terhadap sumber daya yang besar membuat para pemrogram harus bisa melakukan optimasi terhadap penggunaan sumber daya yang ada. Mulai dari sisi logikal di mana programmer harus bisa melakukan optimasi dari sisi algoritma, sampai dari sisi teknis di mana programmer harus bisa melakukan optimasi terhadap pemrosesan dalam ranah *low-level*. *Low-level* yang dimaksud disini adalah segala operasi komputasional yang dilakukan oleh komputer pada tingkat yang lebih dekat dengan perangkat keras yang dimiliki. Hal ini mencakup optimasi penggunaan memori, pemanggilan instruksi yang tepat, dan penggunaan inti atau *thread* yang dimiliki atau disediakan oleh CPU.

Melihat ke domain permasalah ini, maka programmer cenderung untuk melakukan komputasi pada level yang lebih dekat dengan perangkat keras itu sendiri. Hal ini dikarenakan terdapat keuntungan dari sisi eksekusi karena komputasi pada level yang lebih tinggi cenderung harus diterjemahkan terlebih dahulu menuju bentuk komputasi yang bisa dimengerti oleh perangkat keras. Nyaris seluruh perangkat keras yang ada sekarang hanya bisa menerima bentuk data dalam bentuk biner, sehingga segala bentuk instruksi pada akhirnya harus diterjemahkan ke dalam bentuk biner ini.

Salah satu bahasa yang populer dan sering disangkutpautkan sebagai bahasa "tingkat menengah" adalah bahasa C. Bahasa ini sudah lama ada semenjak tahun 1972, dan semenjak itu mendominasi sebagai bahasa yang paling fleksibel untuk digunakan sebagai perantara dari pemrograman tingkat tinggi ke pemrograman tingkat rendah. Bahasa C akan melakukan proses penerjemahan menuju bahasa Assembly terlebih dahulu, yakni bahasa yang semakin mendekati level perangkat keras. Proses kompilasi akan menerjemahkan program dalam bahasa Assembly ini menjadi *machine code* yang hanya terdiri atas kode-kode biner, yang dapat dimengerti dan digunakan langsung oleh perangkat keras.

Adanya proses kompilasi terhadap kode pemrograman membuat bahasa C digolongkan sebagai bahasa yang *compiled*. Bahasa yang dikompilasi akan mengambil suatu kumpulan kode untuk suatu program, lalu melakukan proses penerjemahan secara keseluruhan menjadi bentuk *machine code* yang dimengerti oleh perangkat keras. Hal ini menyebabkan segala perubahan yang ingin dilakukan terhadap kode program harus dilakukan kompilasi ulang lagi agar efek yang diberikan dapat berjalan. Proses ini tentunya akan cukup lambat dan menggunakan sumber daya yang tidak sedikit, terlebih apabila program yang dikompilasi tidak sesederhana program Hello World, namun sebuah program besar seperti sistem operasi. Windows adalah salah satu program besar yang kode sumbernya ditulis dalam bahasa turunan C, yakni C++, dan tentunya akan perlu dilakukan kompilasi ulang terhadap seluruh kode sumber apabila ada suatu perubahan.

Seiring perkembangan metode pemrograman, tidak seluruh program harus dilakukan kompilasi lagi terhadapnya. Mulai dikembangkan pendekatan dengan menggunakan modul, yakni sebuah bagian terpisah yang dapat diintegrasikan ataupun dipisahkan sesuai dengan kebutuhan. Pendekatan ini membuat pemrogram bisa lebih berfokus terhadap fungsionalitas yang ditawarkan oleh satu modul terlebih dahulu, lalu mencoba untuk berinteraksi dengan program utama dengan menggunakan antarmuka yang telah disediakan oleh program utama. Kendati demikian, masih tetap diperlukan proses kompilasi terhadap seluruh bentuk perubahan yang dilakukan terhadap modul ini. Oleh karena itu, dikembangkan lagi suatu bentuk bahasa yang memiliki golongan sebagai bahasa yang *interpreted* atau

diinterpretasikan.

Interpreted language, atau bahasa yang diinterpretasikan tidak akan melakukan kompilasi terhadap seluruh kode program yang ingin dieksekusi. Cara bagi programmer untuk bisa berinteraksi dengan perangkat keras adalah dengan menggunakan *interpreter*, yakni sebuah perangkat lunak / program yang melakukan proses penerjemahan. Proses ini memiliki karakteristik yakni setiap baris kode akan diterjemahkan satu per satu lalu akan dimuat ke dalam memori. Hal ini memberi kita ruang untuk bisa melakukan pengubahan pada kode program tanpa harus membuat kode program hasil kompilasi terlebih dahulu.

Kekurangan dari bahasa yang diinterpretasikan adalah terdapat harga dari sisi performa agar dapat memberi fleksibilitas yang ditawarkan. Adanya *interpreter* dari *interpreted language* membuat terdapat satu langkah yang harus dilewati oleh suatu kode program agar dapat dieksekusi, yakni penerjemahan oleh *interpreter*. Keluaran dari *interpreter* pun bukanlah *machine code* yang dapat langsung dimengerti oleh perangkat keras, melainkan bentuk instruksi yang dapat dimengerti oleh *interpreter* sebagai suatu perintah untuk memanggil fungsionalitas yang disediakan oleh sistem operasi. Apabila pada *compiled language*, eksekusi dilakukan oleh program itu sendiri, maka pada *interpreted language,* eksekusi dilakukan oleh *interpreter*.

Bahasa yang diinterpretasi tentunya memiliki kelemahan yang sangat besar, yakni lebih boros dalam menggunakan sumber daya. Namun, fleksibilitas yang ditawarkan membuat bahasa golongan ini cenderung menjadi pilihan dalam mengembangkan suatu bentuk layanan yang dapat dipindahkan atau dimodifikasi sesuai kebutuhan. Salah satu bentuk layanan ini adalah komputasi awan.

Makalah ini akan mengkaji aplikasi dari kriptografi dalam salah satu bahasa yang sudah lazim digunakan dalam komputasi awan, yakni Javascript. Javascript sudah menjadi standar *de facto* untuk bahasa yang digunakan untuk menggerakkan web pada sisi klien. Javascript sudah menjadi elemen utama dalam membuat antarmuka dan interaksi pada suatu halaman web. Dengan meningkatnya kebutuhan atas layanan berbasis internet, maka tentunya Javascript akan memiliki peran penting dalam menjalankan instruksi yang diberikan

II. TEORI DASAR

A. *Interpreted Language*

Interpreted language memiliki karakteristik untuk melakukan interpretasi terhadap kode program baris per baris, memuat hasil interpretasi ke memori, lalu melakukan eksekusi terhadap program atas nama program *interpreter* yang digunakan. Adanya satu langkah tambahan membuat *interpreter* cenderung lebih lambat daripada program yang terkompilasi, sehingga sebagian besar aplikasi yang menerapkan operasi kriptografi diimplementasikan dalam bahasa yang terkompilasi.

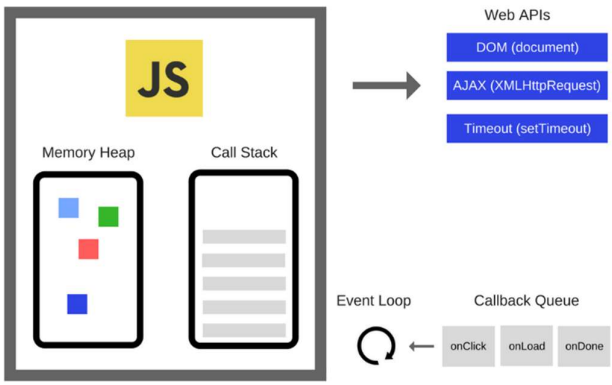
Kendati demikian, *interpreted language* sangat fleksibel dalam penggunaannya, karena sebagian besar penyesuaian terhadap instruksi, penggunaan *system call* dari sistem operasi, bahkan sampai manajemen penggunaan memori telah ditangani oleh *interpreter*. Adanya sistem *garbage collection* adalah salah satu keunggulan dari *interpreted language*, sehingga adanya kemungkinan eror akibat penggunaan memori akan menjadi tanggung jawab sepenuhnya kepada programmer.

Sudah banyak *interpreted language* yang dikenal secara luas, seperti Python, Javascript, Lisp, Ruby, dan masih banyak lagi. Pada konteks ini, akan dibicarakan terkait Javascript yang menjadi penggerak utama dalam teknologi web. Javascript pada awalnya hanya tersedia dalam *web browser*, namun pada tahun 2010 telah muncul distribusi dari Javascript yang bisa berjalan di luar *web browser* dan dapat berinteraksi dengan sistem operasi, yang dikenal sebagai Node.js

B. JavaScript

Javascript secara implementasi adalah bahasa yang hanya mendukung adanya satu *thread* pemrosesan, sehingga dikenal sebagai *single-threaded*. Seluruh pemrosesan dilakukan melalui satu *thread* utama, dikenal dengan sebutan *main thread*. Karakteristik ini adalah sama untuk Javascript pada *web browser* atau pada Node.js.

Seiring perkembangannya jaman, mulai dilakukan modifikasi terhadap model eksekusi dari Javascript. Mulai diperkenalkan suatu sistem bernama *event loop*, yang membuat Javascript bisa mengeksekusi suatu *method* atau fungsi dalam mode asinkron. Model ini tidak membuat Javascript berubah menjadi suatu bahasa yang mendukung *multithreading*, yakni eksekusi program secara paralel dengan menggunakan lebih dari satu *thread*. Ilustrasi terhadap model ekskusi Javascript dapat dilihat pada gambar berikut.



Gambar 1. Model Eksekusi Javascript

Saat suatu program diekskusi, maka *interpreter* akan melakukan scanning dari baris pertama sampai baris terakhir, melakukan deklarasi variabel dan fungsi, melakukan *hoisting* terhadap beberapa bentuk variabel dan fungsi, dan menambahkan instruksi untuk menjalankan suatu fungsi pada *call stack* untuk setiap pemanggilan/invokasi fungsi. Mode beralih menjadi mode eksekusi saat seluruh kode telah dibaca, dan mulai melakukan *pop* pada *call stack*. Instruksi yang telah di-*pop* lalu akan dieksekusi. Apabila terdapat fungsi yang dilabeli sebagai fungsi asinkron, maka fungsi tersebut akan ditambahkan ke dalam *callback queue*. Proses ini akan dilakukan sampai *call stack* kosong.

Setelah *call stack* kosong, maka *event loop* akan melihat kondisi dari *callback queue*. Apabila terdapat item pada antrian, maka item pertama akan di-*deque*, dan di-*push* ke dalam *call*

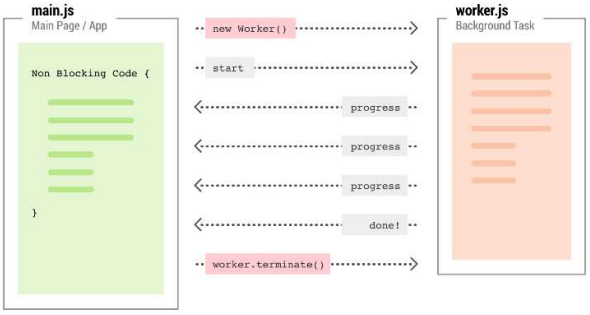
stack. Item itu lalu akan dieksekusi. Apabila dalam proses eksekusi ternyata terdapat pemanggilan terhadap fungsi yang sinkron, maka invokasi fungsi tersebut akan ditambahkan dalam dalam *call stack*, lalu diproses terlebih dahulu sebelum melanjutkan kembali ke fungsi sebelumnya. Apabila terdapat fungsi yang bersifat asinkron, maka invokasi fungsi tersebut akan masuk ke dalam *callback queue*. Proses lalu akan jalan seperti biasa.

Model ini membuat pengalaman di mana Javascript dapat melakukan eksekusi secara paralel, dengan menghilangkan proses menunggu selesainya suatu fungsi yang dilabeli asinkron. Hal ini menghilangkan karakteristik *event blocking* atau *I/O blocking* yang khas dari *single-threaded application*. Namun, proses eksekusi masih tetap berjalan dengan satu *thread*, karena proses berjalan secara sekuensial dengan urutan yang sama, dan dengan suatu bentuk perulangan. Seluruh invokasi fungsi dengan label asinkron pada akhirnya akan dieksekusi satu demi satu, membuat eksekusi tidak benar-benar berjalan layaknya *multithreaded application*.

C. Pemrosesan Paralel pada JavaScript

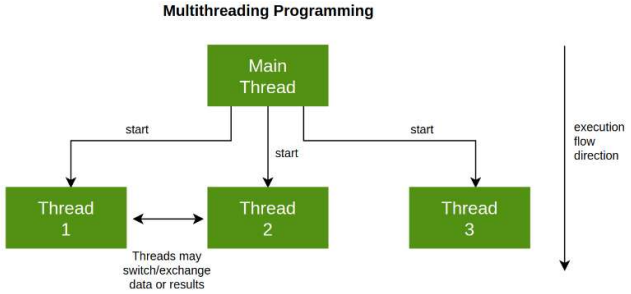
Karakteristik dari Javascript pada akhirnya akan tetap menjadi bahasa pemrograman dengan golongan *single-threaded*, walaupun dengan adanya model eksekusi *event loop* yang telah dibahas pada bagian sebelumnya. Karakteristik ini tetap ada hingga dirilisnya fitur *web worker* pada HTML5.

Web Worker adalah suatu metode untuk menjalankan suatu program di latar belakang, atau dikenal sebagai *background task*. Metode ini dimulai dengan pembuatan sebuah *web worker* pada *main thread*, lalu melakukan *message passing* dari *main thread* ke *web worker*. *Web worker* lalu melakukan eksekusi berdasarkan pesan yang diberikan dari *main thread*, dan apabila pemrosesan telah selesai atau hanya sekadar ingin memberikan status, maka *web worker* akan mengirim pesan ke *main thread*, sebagai respon dari pekerjaan yang telah dilakukan. Ilustrasi dari eksekusi suatu program Javascript dengan *web worker* adalah sebagai berikut.



Gambar 2. Diagram kerja web worker

Dengan adanya proses pembuatan *web worker*, maka kita secara langsung membuat suatu program terpisah yang akan menjalankan suatu fungsi khusus, lalu menerima efek atau hasil dari program terpisah tersebut pada fungsi utama. Hal ini membuat kita bisa melakukan eksekusi terhadap beberapa pekerjaan secara bersamaan, di mana setiap task yang terpisah ditangani oleh *web worker* yang berbeda. Model ini tidaklah berbeda secara prinsip dengan model *multithreading* pada umumnya, sehingga dengan menggunakan *web worker*, maka Javascript sudah bisa menjadi bahasa yang *multithreaded*.



Gambar 3. Alur Eksekusi *Multithreading Programming*

Setiap *web worker* juga dapat melakukan komunikasi terhadap suatu data dengan *web worker* lainnya, sehingga proses *message passing* tidak secara eksklusif hanya bisa dilakukan antara *main-worker* ataupun *worker-main*.

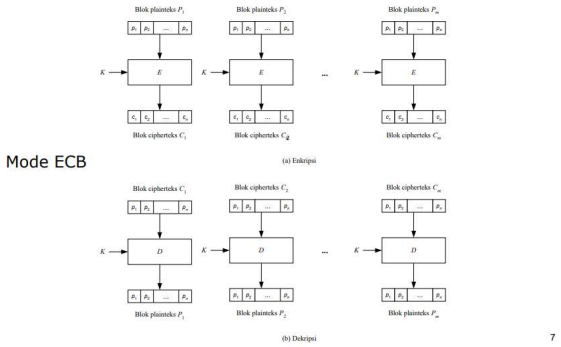
Bentuk komunikasi yang terjadi antara setiap *thread* mengikuti paradigma *event-driven programming*, yakni paradigma pemrograman yang bekerja berdasarkan *event* atau kejadian yang terjadi dalam program secara utuh. Umumnya suatu program yang mengikuti paradigma ini akan memiliki dua komponen utama, yakni *listener* dan *event emitter*. *Listener* akan menerima dan "mendengar" suatu bentuk *event*, lalu melakukan eksekusi terhadap *event* tersebut, sedangkan *event emitter* akan melakukan pengiriman suatu *event* kepada suatu target tujuan. Pada *web worker*, *event emitter* diimplementasikan sebagai suatu *method* bernama "*postMessage"*, sedangkan *listener* dapat diimplementasikan melalui listener *onmessage*, atau penambahan *listener* melalui *method "addEventListener"*. Kedua komponen ini menjadi dasar dari jalannya model *multithreading* pada Javascript.

D. Operasi Kriptografi

Operasi kriptografi adalah operasi-operasi yang melibatkan teknik kriptografi di dalamnya. Operasi ini dapat mencakup pembuatan kunci, enkripsi, dekripsi, validasi, dan penandatanganan. Operasi kriptografi memiliki tujuan untuk mengubah suatu data mentah menjadi suatu data yang sulit ditebak maknanya, dan dapat dikembalikan menjadi bentuk semula dengan menggunakan suatu kunci. Metode ini digunakan untuk mengirim suatu pesan secara rahasia agar data tetap tidak dapat ditebak maknanya apabila disadap oleh pihak lain.

Terdapat banyak bentuk dari operasi kriptografi, namun yang akan dibahas adalah operasi kriptografi modern yang menggunakan *block cipher*. *Block cipher* adalah metode kriptografi yang membagi suatu pesan menjadi blok-blok dengan panjang tertentu sebelum dilakukan proses terhadapnya. Melalui pembagian ini, maka ada 3 bentuk operasi yang dapat terjadi : ECB, CBC, dan Counter. ECB adalah mode yang akan melakukan operasi enkripsi maupun dekripsi terhadap blok-blok data, dan setiap blok tidak memiliki keterkaitan apapun satu-sama lain. Hal ini membuat paralelisasi dapat diaplikasikan pada enkripsi dalam mode ECB. Lain halnya dengan CBC yang menggunakan hasil dari blok sebelumnya untuk melakukan enkripsi atau dekripsi pada blok yang sedang dikerjakan, mode CBC tidak memungkinkan adanya paralelisasi. Counter adalah

mode yang menggunakan angka *counter* yang sifatnya deterministik dan tidak berkaitan dengan blok lainnya, sehingga paralelisasi dapat dilakukan pada mode ini. Makalah ini akan secara spesifik menganalisa perbandingan pada mode ECB.



Gambar 4. Diagram Pemrosesan Block Cipher mode ECB

Terdapat mode lain, yakni CFB dan OFB, namun sama dengan CBC, mode-mode ini tidak memungkinkan terjadinya paralelisasi karena setiap blok memiliki keterkaitan dengan blok lainnya.

III. IMPLEMENTASI

A. Web Crypto API

Pengujian dilakukan dengan menggunakan *Web Crypto API*, yakni seperangkat pustaka untuk melakukan operasi-operasi kriptografi yang tersedia dalam *browser*. *Web Crypto API* telah mendukung berbagai operasi kriptografi umum, seperti enkripsi, dekripsi, verifikasi, penandatanganan, pembangkitan kunci, dan operasi-operasi lainnya.

Pengaksesan terhadap operasi-operasi ini dapat dilakukan melalui objek *window* pada *main thread*, dan objek *self* pada *worker thread*. Setiap objek tetap memiliki fungsionalitas yang sama sehingga dilansir tidak akan memberi perbedaan secara implementasi antara kedua bentuk *thread*.

Dalam makalah ini, akan digunakan algoritma kriptografi kunci publik RSA, dengan standar RSA-OAEP. Penjelasan yang lebih detail dari standar ini telah diatur dalam dokumen RFC 3447. Proses enkripsi dan dekripsi dapat dilakukan dengan fungsi *encrypt* dan *decrypt* yang sama-sama menerima 3 buah parameter. Parameter pertama mendeskripsikan algoritma yang digunakan, parameter kedua mendeskripsikan kunci yang digunakan, dan parameter ketiga adalah data yang akan dilakukan enkripsi/dekripsi. Terkhusus untuk algoritma RSA-OAEP, maka enkripsi akan menggunakan kunci publik, dan dekripsi menggunakan kunci privat. Seluruh bentuk data harus berada dalam bentuk *typed array*, dengan tipe data *UInt8Array* atau *ArrayBuffer*. Teks dapat diubah ke dalam bentuk tersebut menggunakan objek *TextEncoder*, dan dikembalikan menggunakan objek *TextDecoder*.

Pembangkitan kunci menggunakan fungsi *generateKey* yang menerima 3 parameter. Parameter pertama adalah objek yang mengandung informasi mengenai algoritma yang digunakan dan parameter terkait dengan algoritma tersebut. Parameter kedua menyatakan apakah kunci yang dibangkitkan dapat diekstraksi, yakni diproses menjadi data yang dapat ditransfer dan digunakan ulang. Parameter ketiga adalah *Array* yang menyatakan tujuan penggunaan kunci, di mana dalam makalah ini digunakan *"['encrypt', 'decrypt']"* yang menyatakan bahwa kunci akan digunakan untuk proses enkripsi dan dekripsi saja.

Secara ringkas, proses pembangkitan kunci adalah dengan kode berikut

```
const crypto = window.crypto.subtle;
const keypair = await crypto.generateKey({
    name: 'RSA-OAEP',
    modulusLength: 2048,
    publicExponent: new Uint8Array([1,0,1]),
    hash: 'SHA-256'
}, true, ["encrypt", 'decrypt']);
```

Dari contoh kode yang diberikan, variabel *keypair* akan memiliki kunci publik dan kunci privat yang dapat diakses melalui membernya.

B. Enkripsi Sekuensial

Karakteristik dari *block cipher* adalah data harus terlebih dahulu dipecah menjadi blok-blok dengan suatu panjang maksimal. Blok terakhir umumnya memiliki panjang lebih kecil dari panjang maksimal, dengan kasus khusus untuk menambahkan *padding* yakni data tambahan untuk membuat panjang blok terakhir adalah sama dengan panjang maksimal.

Proses enkripsi akan menggunakan satu agen yang akan melakukan proses enkripsi terhadap blok per blok. Hal yang sama dilakukan untuk proses dekripsi. Karena hanya akan ada satu agen yang terlibat, maka proses enkripsi dan dekripsi akan dijalankan pada *main thread*.

C. Enkripsi Paralel

Proses yang dilakukan untuk melakukan proses enkripsi secara paralel adalah dengan membuat beberapa *worker thread* lalu melakukan pembagian tugas dengan mengirim blok-blok ke setiap thread untuk dilakukan proses enkripsi ataupun dekripsi. Karena *worker thread* tidak mengetahui kondisi apapun yang sedang terjadi di *main thread*, maka data yang dikirim melalui fungsi *postMessage* diberi keterangan mengenai data-data yang diperlukan, seperti nomor blok dan ID dari worker yang dituju. Pengiriman data dari *main thread* ke *worker thread* diimplementasikan dalam potongan kode berikut.

```
const message = {
            message,
            operation: 'ENC',
            targetId,
            key: keypair.publicKey,
            seqNum
        }
const worker = workerObj[workerId];
worker.postMessage(message);
```

Seluruh operasi kriptografi diimplementasikan dalam kode *web worker*, dan *web worker* mampu menangani dua mode yakni mode enkripsi dan dekripsi. Implementasi kode adalah serupa dengan implementasi dalam mode sekuensial, dengan penambahan bahwa hasil akan dikembalikan dalam sebuah objek yang mengandung informasi nomor blok, dan ID *web worker* yang digunakan.

Pencatatan dari waktu mulai adalah saat blok pertama dikirim

ke *web worker*, dan waktu akhir adalah saat seluruh blok sudah diterima dari *web worker*. Hal ini adalah sama untuk mode enkripsi maupun mode dekripsi.

IV. ANALISIS

Hipotesis awal yang dimiliki oleh penulis adalah akan diperoleh hasil di mana proses paralel akan lebih efisien ketimbang proses sekuensial. Hal ini didasarkan dengan anggapan bahwa semakin banyak agen yang dapat melakukan operasi kriptografi, maka semakin cepat waktu yang dibutuhkan untuk mengenkripsi atau mendekripsi seluruh blok. Namun hasil yang didapatkan sangatlah berbeda.

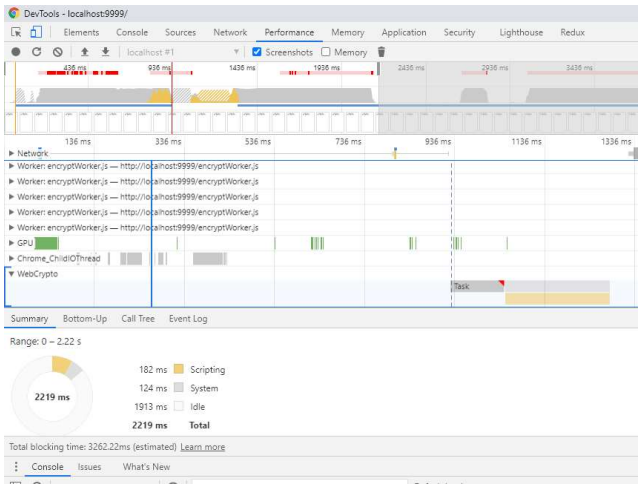
Proses *benchmarking* dilakukan dengan menggunakan teks yakni 5 paragraf pertama dari "Lorem Ipsum". Teks dipecah menjadi 210 blok sehingga akan dibutuhkan 210 *web worker* untuk memproses seluruh blok.

```
Total chunks: 210
Sequential Encrypt: Time Elapsed: 32ms
Sequential Decrypt: Time Elapsed: 224ms
Beginning Multithreaded Encryption
Spawning Workers
Starting message passing
Paralel Encrypt: Time Elapsed: 3053ms
Paralel Decrypt: Time Elapsed: 475ms
```

Gambar 5. Benchmark Pemrosesan Kriptografi

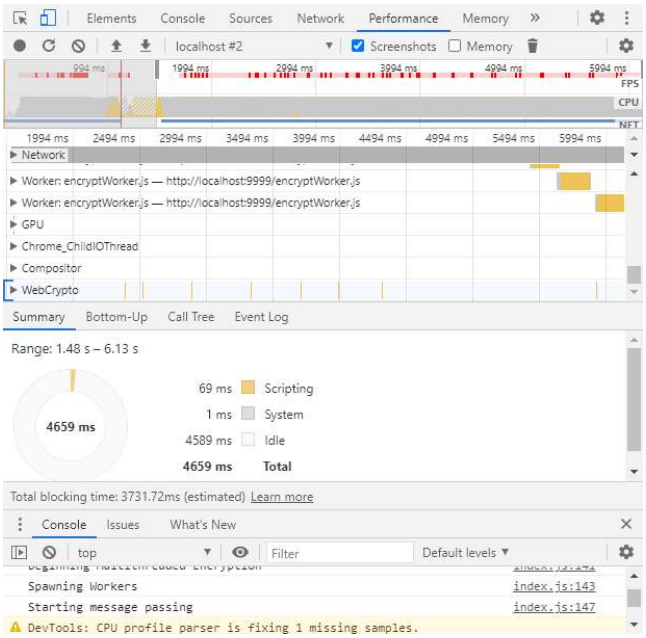
Diperoleh bahwa waktu untuk melakukan enkripsi dan dekripsi dalam mode sekuensial adalah lebih baik daripada waktu untuk melakukan enkripsi dan dekripsi dalam mode paralel, dengan perbedaan yang sangat signifikan pada proses enkripsi. Dapat dilihat performa enkripsi secara sekuensial lebih baik nyaris 10 kali dibandingkan dengan mode paralel, dan untuk mode dekripsi, performa lebih baik 2 kali lipat dibandingkan dengan mode paralel.

Setelah melihat bagian *memory profiling* dan grafik *waterfall* dari aplikasi, maka dapat dilihat bahwa untuk bagian sekuensial menggunakan waktu yang cukup singkat untuk melakukan eksekusi dari operasi kriptografinya. Hal ini dapat dilihat dari bagian "Scripting" pada bagian WebCrypto, yang menyatakan penggunaan Web Crypto API pada aplikasi.



Gambar 6. Tab Performa saat Eksekusi Secara Sekuensial

Kendati demikian, setelah melihat total waktu yang digunakan oleh seluruh *web worker* yang melibatkan pemanggilan Web Crypto API, maka ditemukan bahwa waktu eksekusi yang diberikan ternyata jauh lebih sedikit, dengan total rata-rata 60-80 milisekon untuk melakukan eksekusi operasi kripto. Hal ini menunjukkan bahwa pemrosesan secara paralel tetap memberi hasil yang lebih baik, apabila waktu yang dihitung adalah waktu total untuk melakukan proses komputasi saja. Dari sini dapat dilihat terdapat faktor lain yang membuat proses tetap berjalan seperti dalam mode sekuensial, ditunjukkan dengan grafik *waterfall* (warna kuning) dengan bentuk yang menyerupai tangga pada setiap *web worker*.



Gambar 7. Tab Performa saat Eksekusi secara Paralel

Setelah melihat dokumentasi dan cara kerja dari *web worker*, maka ditemukan bahwa sebagian besar latensi yang muncul disebabkan oleh proses *message passing* itu sendiri. Telah dijelaskan sebelumnya bahwa untuk melakukan komunikasi antar *thread*, digunakan metode *message passing* dengan fungsi *postMessage*. Proses pengiriman pesan ini melakukan proses

duplikasi terhadap seluruh pesan terlebih dahulu di belakang layar, lalu mulai mengirim pesan tersebut ke *thread* tujuan. Adanya proses duplikasi ini tentu saja memerlukan beberapa langkah tambahan, seperti proses duplikasi itu sendiri, proses pembersihan memori, lalu masuk ke proses pengiriman data itu sendiri.

Secara kasar dan melihat ke hasil yang dirasakan oleh pengguna, maka mode sekuensial akan memberi hasil yang lebih cepat ketimbang mode paralel. Namun secara bersih, yakni dengan melihat waktu eksekusi yang melibatkan pemanggilan fungsi Web Crypto API, maka mode paralel tetap memberi waktu pemrosesan yang lebih baik. Kendala hanya terdapat dari karakteristik dari Javascript sebagai *interpreted language* yang melakukan beberapa operasi tambahan ditengah-tengah, seperti *garbage collection*.

V. KESIMPULAN

Operasi kriptografi adalah operasi yang terkenal menggunakan sumber daya yang besar sebagai upaya untuk menyulitkan proses dekripsi secara *brute force*. Aplikasi dari kriptografi telah diimplementasikan dalam berbagai *platform*, mulai dari sebuah aplikasi khusus yang dibuat di desktop, menjadi sebuah API seperti Web Crypto API agar dapat dijalankan dalam web, atau bahkan menjadi sebuah layanan yang tersedia secara publik.

Pada umumnya, aplikasi yang dapat menggunakan sumber daya secara optimal akan memberi performa yang lebih bagus. Penggunaan beberapa *thread* secara sekaligus adalah suatu praktik yang efektif dalam meningkarkan performa dari suatu aplikasi. Namun, penggunaan dari lebih dari satu sumber daya mewajibkan kita agar dapat menangani sumber daya yang digunakan.

Pada analisis dalam makalah ini, digunakan Javascript sebagai *interpreted language*, lalu dilakukan operasi kriptografi dalam mode paralel dengan menggunakan *web worker*. Ditemukan hasil bahwa terdapat pemborosan waktu atau *overhead* akibat proses dari *message passing* ataupun proses-proses lainnya, seperti *garbage collection*. Adanya proses ini adalah wajar mengingat kebanyakan *interpreted language* tidak memberi kontrol secara langsung terhadap penggunaan memori dan instruksi-instruksi yang *low-level*, sehingga secara implisit akan selalu dijalankan oleh *interpreter* untuk meminimalisir adanya kesalahan atau eror saat melakukan eksekusi yang berkaitan dengan *low level programming*. Oleh karena itu, waktu eksekusi secara paralel memberi hasil yang lebih buruk dari pengalaman yang dirasakan oleh pengguna, namun secara teknis tetap memberi hasil yang lebih baik apabila hanya memperhitungkan konteks waktu eksekusi operasi kriptografi.

Proses enkripsi dan dekripsi secara paralel akan lebih baik digunakan pada *compiled language*, agar kita bisa memiliki kontrol yang lebih terhadap penanganan sumber daya yang dimiliki.

VII. PENUTUP

Penulis mengucapkan terima kasih sebesar-besarnya kepada Allah SWT, atas berkat dan rahmat-Nya penulis masih diberi kesehatan agar dapat menyelesaikan makalah ini. Penulis juga mengucapkan terima kasih kepada seluruh keluarga penulis, terutama Ibu sebagai sosok yang selalu perhatian dan senantiasa mengingatkan penulis untuk makan dan jaga kesehatan. Terima kasih yang sebesar-besarnya kepada bapak Rinaldi Munir, selaku dosen IF4020 Kriptografi atas bimbingan dan ilmu yang diberikan. Terima kasih kepada seluruh pihak yang memberi dukungan, dan ucapan terima kasih kepada dua sosok Vtuber yakni Nakiri Ayame dan Shiina Nanoha sebagai penyemangat bagi penulis agar bisa rileks saat mengalami buntu dalam kekurangan ide.

REFERENSI

[1] Munir, Rinaldi. "Kripto Modern Bagian 3", http://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi/2020-2021/Kripto-modern-2020-bagian3.pdf (terakhir diakses pada 15.32 WITA, 21 Desember 2020)

[2] Web Crypto API – Web APIs | MDN, https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API (terakhir diakses pada 16.26 WITA, 21 Desember 2020)

[3] "The Difference Between Asynchronous And Multi-Threading", https://www.baeldung.com/cs/async-vs-multi-threading (terakhir diakses pada 14.12 WITA, 21 Desember 2020)

PERNYATAAN

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

Makassar, 21 Desember 2020

Arung Agamani Budi Putera - 13518005