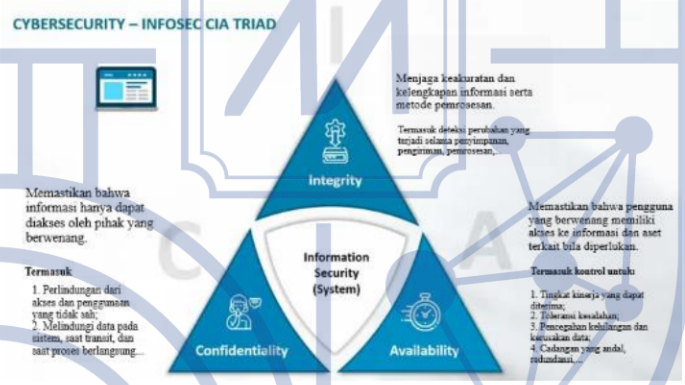
BAB II
KAJIAN LITERATUR

2.1 Keamanan Jaringan Komputer

Keamanan jaringan komputer merupakan aspek penting dalam pengelolaan sistem informasi modern karena jaringan menjadi media utama dalam pertukaran data dan komunikasi antar sistem. Seiring dengan meningkatnya ketergantungan terhadap jaringan komputer, berbagai ancaman terhadap keamanan jaringan juga semakin berkembang dan berpotensi menyebabkan gangguan layanan, pencurian data, maupun kerusakan sistem [9]. Secara umum, keamanan jaringan komputer didefinisikan sebagai serangkaian kebijakan, prosedur, dan mekanisme yang dirancang untuk melindungi integritas, kerahasiaan, dan ketersediaan data yang ditransmisikan melalui jaringan dari berbagai ancaman dan akses yang tidak sah [9], [10]. Konsep ini mengacu pada tiga prinsip dasar yang dikenal sebagai CIA Triad, yaitu *confidentiality*, *integrity*, dan *availability*.



Gambar 2. 1 CIA Triad dalam Keamanan Informasi [11]

Berdasarkan Gambar 2. 1, *Confidentiality* berkaitan dengan upaya menjaga agar informasi hanya dapat diakses oleh pihak yang berwenang. *Integrity* bertujuan untuk memastikan bahwa data tidak mengalami perubahan atau manipulasi selama proses transmisi. Sementara itu, *availability* menekankan pada ketersediaan sistem dan layanan jaringan agar tetap dapat diakses oleh pengguna yang sah ketika dibutuhkan [10].

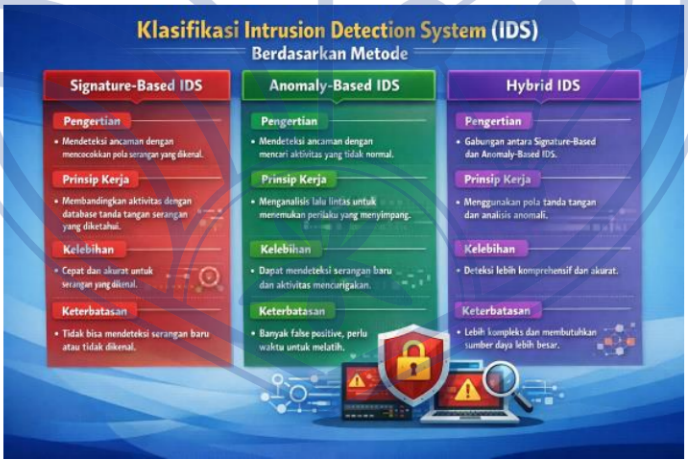
Dalam praktiknya, jaringan komputer menghadapi berbagai jenis ancaman yang dapat mengganggu operasional sistem, baik yang disebabkan oleh serangan siber maupun kesalahan konfigurasi. Salah satu ancaman yang umum terjadi adalah serangan terhadap ketersediaan layanan, seperti DDoS, yang bertujuan untuk membanjiri sumber daya jaringan sehingga layanan tidak dapat diakses oleh pengguna yang sah [12].

Untuk menghadapi ancaman tersebut, berbagai mekanisme keamanan jaringan telah dikembangkan, seperti penggunaan *firewall*, sistem deteksi intrusi (*Intrusion Detection System*), serta teknik analisis trafik jaringan. Dalam konteks ini, kemampuan untuk melakukan deteksi aktivitas jaringan secara cepat dan berkelanjutan menjadi penting, terutama dalam menghadapi serangan yang bersifat dinamis seperti DDoS [12].

Dalam penelitian ini, konsep *Confidentiality* dan *Integrity* tidak diimplementasikan karena dapat menyebabkan lonjakan *latency* yang tinggi akibat proses enkripsi-dekripsi serta *overhead* komputasi tambahan. Namun, konsep *Availability* tetap digunakan karena untuk memastikan ketersediaan proses deteksi serangan DDoS walaupun terjadi lonjakan lalu lintas atau tekanan tinggi pada *pipeline* data.

2.2 Intrusion Detection System

Intrusion Detection System (IDS) merupakan sistem keamanan yang dirancang untuk memantau aktivitas pada jaringan komputer atau sistem informasi dengan tujuan mendeteksi adanya aktivitas mencurigakan, serangan, maupun pelanggaran terhadap kebijakan keamanan. IDS bekerja dengan menganalisis lalu lintas jaringan atau aktivitas sistem untuk mengidentifikasi pola yang menunjukkan adanya potensi intrusi atau ancaman terhadap sistem [12]. Secara umum, IDS dapat diklasifikasikan berdasarkan pendekatan deteksinya menjadi dua kategori utama, yaitu *signature-based detection* dan *anomaly-based detection*.



Gambar 2. 2 Klasifikasi sistem deteksi intrusi menurut metodenya

Berdasarkan penjelasan gambar 2. 2, pendekatan *signature-based detection* bekerja dengan membandingkan pola trafik jaringan dengan data pola serangan yang telah dietahui sebelumnya, sehingga efektif untuk mendeteksi serangan yang sudah dikenali. Namun, pendekatan ini memiliki keterbatasan dalam mendeteksi serangan baru yang belum memiliki pola dalam basis data. Sebaliknya, pendekatan *anomaly-based detection* bekerja dengan

mempelajari pola perilaku normal dari suatu sistem atau jaringan, kemudian mendeteksi setiap penyimpangan sebagai indikasi adanya aktivitas yang tidak wajar. Pendekatan ini banyak digunakan dalam penelitian keamanan jaringan modern karena mampu mengidentifikasi serangan baru yang bersifat dinamis, termasuk serangan DDoS [12].

Dalam perkembangannya, pendekatan *anomaly-based detection detection* sering dikombinasikan dengan teknik *machine learning* untuk meningkatkan kemampuan sistem dalam menganalisis pola trafik yang kompleks. Model *machine learning* dapat dilatih menggunakan dataset trafik jaringan untuk mengklasifikasikan trafik normal dan trafik serangan secara otomatis [13]. Dalam penelitian ini, konsep IDS digunakan sebagai landasan dalam proses deteksi serangan, khususnya melalui pendekatan *anomaly-based detection* berbasis *machine learning*. Namun, penelitian ini tidak berfokus pada pengembangan sistem IDS secara keseluruhan, melainkan pada implementasi *pipeline* pemrosesan *data streaming* untuk mendukung proses deteksi serangan DDos secara *near real-time*.

2.3 Distributed Denial of Service

Serangan *Distributed Denial of Service* (DDoS) merupakan salah satu ancaman keamanan jaringan yang bertujuan untuk mengganggu ketersediaan layanan suatu sistem dengan cara membanjiri sumber daya jaringan menggunakan lalu lintas data dalam jumlah besar. Serangan ini dilakukan secara terdistribusi melalui banyak perangkat yang telah terinfeksi sehingga sistem target tidak mampu menangani permintaan yang masuk dan mengalami penurunan performa atau bahkan tidak dapat diakses oleh pengguna yang sah. Serangan DDoS umumnya melibatkan sejumlah besar *host* yang tersebar di berbagai lokasi jaringan yang secara bersamaan mengirimkan lalu lintas ke sistem target untuk menghabiskan sumber daya jaringan atau komputasi. Perangkat-perangkat tersebut biasanya tergabung dalam suatu jaringan *botnet* yang dikendalikan oleh penyerang [4].

Dalam praktiknya, serangan DDoS terdiri dari tiga komponen utama, yaitu penyerang (*attacker*), jaringan perangkat yang telah terkompromi (*botnet*), dan target serangan (*victim*). Penyerang terlebih dahulu menginfeksi sejumlah perangkat untuk dijadikan *bot*, kemudian mengirimkan perintah secara terpusat agar seluruh *bot* secara bersamaan mengirimkan trafik menuju sistem target sehingga menyebabkan kelebihan beban pada sumber daya jaringan seperti *bandwidth*, CPU, dan memori [4].

Serangan DDoS dapat diklasifikasikan ke dalam beberapa kategori berdasarkan metode yang digunakan, antara lain *volume-based attack*, *protocol attack*, dan *application layer attack*. *Volume-based attack* bertujuan untuk menghabiskan *bandwidth* jaringan,

seperti pada serangan *UDP flood*. Protocol *attack* memanfaatkan kelemahan pada protokol jaringan, seperti *SYN flood*, untuk menghabiskan sumber daya server. Sementara itu, *application layer attack* menargetkan layanan aplikasi tertentu, seperti HTTP atau LDAP, untuk mengganggu operasional sistem yang berjalan pada server. Dalam penelitian ini, fokus deteksi diarahkan pada beberapa jenis serangan DDoS yang umum ditemukan pada trafik jaringan, yaitu *UDP flood*, *SYN flood*, serta serangan berbasis layanan aplikasi seperti LDAP [5].

Dampak dari serangan DDoS dapat sangat signifikan, mulai dari terganggunya ketersediaan layanan, kerugian finansial, hingga penurunan reputasi organisasi. Oleh karena itu, diperlukan mekanisme deteksi yang mampu mengidentifikasi aktivitas lalu lintas jaringan yang mencurigakan secara cepat dan berkelanjutan untuk mendukung keamanan dan keandalan sistem jaringan [1], [2].

2.4 CIC-DDoS 2019

CIC-DDoS 2019 merupakan dataset yang digunakan untuk penelitian deteksi serangan DDoS dan dirancang untuk merepresentasikan trafik jaringan modern yang terdiri dari data normal dan berbagai jenis serangan,. Dataset ini memiliki sejumlah fitur berbasis aliran jaringan (*flow-based features*) seperti jumlah paket, jumlah *byte*, dan durasi koneksi yang digunakan untuk menganalisis pola trafik. Selain itu, variasi jenis serangan seperti UDP dan SYN memungkinkan model *machine learning* mempelajari karakteristik serangan yang berbeda, sehingga meningkatkan kemampuan klasifikasi. Dataset ini banyak digunakan dalam penelitian *intrusion detection* karena ukurannya yang besar dan strukturnya yang mendukung proses analisis dan pengujian model, [13].

Dataset CIC-DDoS 2019 yang digunakan dalam penelitian ini memiliki 88 atribut kolom yang merepresentasikan karakteristik trafik jaringan. Atribut-atribut tersebut mencakup informasi identitas komunikasi, protokol, waktu, statistik paket, volume data, serta label klasifikasi. Adapun Rincian atribut yang terdapat pada dataset ini dapat dilihat pada Tabel 2.1.

Tabel 2. 1 Identifikasi Sebagian Atribut Dataset [13]

No	Kelompok	Kolom	Tipe Data	Keterangan
1	Identifier & Addressing	Flow ID	String	ID unik flow (gabungan IP, port, dll)
		Source IP	String	Alamat IP sumber
		Destination IP	String	Alamat IP tujuan
2		Source Port	Integer	Port sumber (0–65535)

	Port & Protocol	Destination Port	Integer	Port tujuan
		Protocol	Integer	kode protokol (TCP=6, UDP=17, dll)
3	Waktu & Durasi	Timestamp	Timestamp	Waktu kejadian
		Flow Duration	Integer	Durasi flow
4	Statistik Paket	Total Fwd Packets	Integer	Atribut yang menggambarkan karakteristik paket data yang ditransmisikan selama komunikasi jaringan berlangsung, seperti jumlah paket serta ukuran paket yang dikirim.
		Total Backward Packets	Integer	
		Packet Length Mean	Double	
		Packet Length Std	Double	
		Max Packet Length	Double	
5	Statistik Volume data	Total Length of Fwd Packets	Double	Representasikan jumlah dan laju transfer data yang terjadi dalam suatu aliran komunikasi jaringan
		Total Length of Bwd Packets	Double	
6	Statistik Laju Trafik	Flow Bytes/s	Double	Atribut yang menunjukkan tingkat intensitas trafik jaringan berdasarkan jumlah paket yang dikirim dalam satuan waktu tertentu.
		Flow Packets/s	Double	
		Fwd Packets/s	Double	
		Bwd Packets/s	Double	
7	Statistik Interval Waktu (IAT)	Flow IAT Mean	Double	Atribut yang menggambarkan interval waktu antar paket dalam suatu aliran komunikasi jaringan.
		Flow IAT Std	Double	
		Fwd IAT Mean	Double	
		Bwd IAT Mean	Double	
8	TCP Flag Features	SYN Flag Count	Integer	Rerepresentasikan jumlah kemunculan flag pada protokol TCP yang dapat menunjukkan karakteristik komunikasi jaringan tertentu.
		ACK Flag Count	Integer	
		RST Flag Count	Integer	
		FIN Flag Count	Integer	
		PSH Flag Count	Integer	
9	Statistik Aktivitas Flow	Active Mean	Double	Atribut yang merepresentasikan jumlah kemunculan flag pada protokol TCP yang dapat menunjukkan karakteristik komunikasi jaringan tertentu.
		Active Std	Double	
		Idle Mean	Double	
		Idle Std	Double	
10	Label Data	Label	String	Kategori trafik jaringan yang digunakan sebagai target dalam proses klasifikasi.

2.5 Algoritma Random Forest

Peningkatan volume lalu lintas jaringan menyebabkan metode deteksi berbasis aturan statis menjadi kurang efektif dalam mengidentifikasi pola serangan yang kompleks. Oleh karena itu, pendekatan berbasis *machine learning* digunakan karena mampu

mempelajari pola dari data historis dan melakukan klasifikasi aktivitas jaringan secara otomatis, sehingga dapat membedakan antara trafik normal dan aktivitas yang berpotensi sebagai serangan siber [13], [14].

Salah satu algoritma yang banyak digunakan dalam deteksi intrusi adalah *Random Forest*. Algoritma ini memiliki kemampuan klasifikasi yang baik, mampu menangani jumlah fitur yang besar, serta lebih tahan terhadap *overfitting* dibandingkan *decision tree* tunggal. *Random Forest* merupakan metode *ensemble learning* yang membangun banyak *decision tree* selama proses pelatihan. Setiap pohon dibangun menggunakan subset data dan subset fitur yang dipilih secara acak (*bagging*), sehingga meningkatkan keberagaman model dan menghasilkan prediksi yang lebih stabil dan akurat [15].

Dalam proses pembentukan *decision tree*, digunakan kriteria pemisahan untuk menentukan *split* terbaik pada setiap node. Salah satu metode yang umum digunakan adalah *Gini impurity*, yang mengukur tingkat ketidakmurnian suatu node berdasarkan distribusi kelas data di dalamnya.

Secara Matematis, Gini impurity dirumuskan sebagai berikut:

$$\mathrm{Gini}(S) = 1 - \sum_i^n P_i^{\,2} \quad \text{.....(2. 1)}$$

di mana:

1. S adalah himpunan data pada suatu node,
2. P_i adalah proporsi data yang termasuk ke dalam kelas ke-i
3. n adalah jumlah kelas.

Nilai *Gini impurity* yang lebih kecil menunjukkan data dalam suatu node semakin homogen, sehingga pemisahan tersebut dianggap lebih baik. Dalam proses pembentukan pohon, algoritma akan mengevaluasi berbagai kemungkinan pemisahan berdasarkan fitur tertentu. Untuk setiap kandidat *split*, nilai *Gini impurity* dari masing-masing subset hasil pemisahan dihitung dan digabungkan dalam bentuk *weighted Gini* sebagai berikut [16]:

$$\mathrm{Gini}_{\mathrm{split}} = \left|\frac{S_1}{S}\right| . \mathrm{Gini}(S_1) + \left|\frac{S_2}{S}\right| . \mathrm{Gini}(S_2) \quad \text{.....(2. 2)}$$

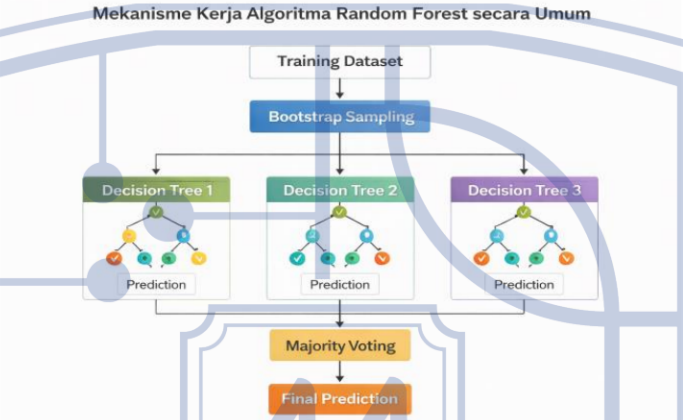
Di mana,

1. S_1 dan S_2 adalah subset hasil pemisahan,
2. $|S|$ adalah jumlah total data,
3. $|S_1|$ dan $|S_2|$ adalah jumlah data pada masing-masing subset.

Pemilihan split terbaik dilakukan dengan memilih nilai $Gini_{split}$ yang paling kecil, karena menunjukkan bahwa hasil pemisahan menghasilkan *node* yang lebih homogen. Dengan pendekatan ini, setiap pohon dalam *Random Forest* memiliki struktur yang berbeda

karena variasi data dan fitur yang digunakan. Perbedaan ini menjadi kunci utama dalam meningkatkan performa model secara keseluruhan melalui mekanisme *ensemble* [15].

Dalam proses klasifikasi, setiap pohon akan menghasilkan prediksi terhadap data yang diuji. Hasil prediksi akhir kemudian ditentukan menggunakan mekanisme *majority voting*, yaitu kelas yang paling banyak diprediksi oleh seluruh pohon keputusan akan menjadi hasil klasifikasi akhir [15]. Pendekatan ini memungkinkan *Random Forest* untuk mengurangi variansi model serta meningkatkan kemampuan generalisasi terhadap data baru.



Gambar 2. 3 Mekanisme kerja algoritma Random Forest [17]

Berdasarkan gambar 2.3, proses pembentukan model *Random Forest* dilakukan melalui tahapan sebagai berikut [17]:

1. Bootstrap Sampling

   Dataset pelatihan diambil secara acak menggunakan metode *sampling with replacement* untuk membentuk beberapa subset data. Setiap subset digunakan untuk membangun satu pohon keputusan.

2. Pembangunan Pohon Keputusan

   Untuk setiap subset data, dilakukan proses pembentukan pohon keputusan dari *node* akar (*root*) hingga node daun (*leaf*) melalui tahapan berikut:

   a. Pemilihan Fitur Secara Acak (Random Feature Selection)

      Pada setiap *node*, sejumlah fitur dipilih secara acak dari seluruh fitur yang tersedia. Jumlah fitur yang dipilih umumnya ditentukan menggunakan rumus.

      $$\lfloor \sqrt{p} \rfloor \quad (2.\ 3)$$

      di mana p adalah jumlah total fitur.

   b. Penentuan Kandidat Threshold

Untuk setiap fitur terpilih, data diurutkan terlebih dahulu, kemudian ditentukan nilai-nilai kandidat threshold dari titik tengah antara dua nilai unik yang berurutan. Untuk menghitung threshold tersebut digunakan rumus.

$$t_i = \frac{(x_i + x_{i+1})}{2} \quad \text{.....(2. 4)}$$

c. Perhitungan Gini Impurity

Untuk setiap kandidat threshold, data dibagi menjadi dua subset (kiri dan kanan). Nilai Gini impurity masing-masing subset dihitung untuk mengukur tingkat ketidakmurnian data.

d. Perhitungan Gini Split

Nilai *Gini impurity* dari kedua subset digabungkan dalam bentuk *weighted average* untuk memperoleh nilai *Gini split*.

e. Pemilihan Split Terbaik

Split terbaik ditentukan berdasarkan nilai *Gini split* terkecil, karena menghasilkan pembagian data yang paling homogen.

f. Pembentukan Node Anak

Data dibagi berdasarkan *split* terbaik menjadi node kiri dan node kanan.

g. Proses Rekursif

Langkah b–f diulang pada setiap *node* anak dengan hanya menggunakan data yang masuk ke *node* tersebut.

h. Kondisi Penghentian (Stopping Criteria)

Proses pembentukan pohon dihentikan jika:

i. Seluruh data dalam node berasal dari satu kelas (*pure*);
ii. Tidak terdapat kandidat split yang memungkinkan; atau
iii. Memenuhi batas tertentu seperti minimum sampel atau kedalaman maksimum.

i. Penentuan Label Leaf Node

3. Pembentukan Ensemble (Forest)

Proses pembangunan pohon dilakukan berulang kali hingga terbentuk sejumlah pohon keputusan yang berbeda akibat variasi data dan fitur.

4. Proses Klasifikasi

Pada tahap prediksi, setiap pohon memberikan hasil klasifikasi terhadap data uji. Hasil akhir ditentukan berdasarkan kelas yang paling banyak diprediksi oleh seluruh pohon (*majority voting*).

Berdasarkan mekanisme pembentukan pohon keputusan yang dijelaskan, bentuk setiap pohon dalam *Random Forest* tidak seragam. Perbedaan bentuk pohon muncul karena setiap pohon dibangun dari sampel data dan subset fitur yang berbeda akibat proses *bootstrap sampling* dan *random feature selection*. Ada yang bercabang dan ada yang tidak karena proses rekursif berhenti ketika suatu node sudah mencapai kondisi *pure* (semua sampel memiliki kelas yang sama) atau tidak ada kandidat *split* yang dapat menurunkan *impurity*, sehingga node tersebut menjadi daun tanpa cabang berikutnya. Tingkat kedalaman yang berbeda disebabkan oleh kompleksitas data pada setiap subset: jika setelah suatu *split node* anak sudah homogen, pohon berhenti lebih awal. Sebaliknya, jika *node* anak masih heterogen, proses rekursif berlanjut hingga batas yang ditentukan. Dengan demikian, variasi bentuk, percabangan, dan kedalaman pohon merupakan konsekuensi alami dari algoritma *Random Forest* dan justru menjadi sumber diversitas yang meningkatkan akurasi *ensemble* secara keseluruhan [16].

Selain itu, Evaluasi model dilakukan untuk mengukur kinerja algoritma *Random Forest* dalam mengklasifikasikan data trafik jaringan ke dalam beberapa kategori. Pengujian dilakukan menggunakan data uji (test data) yang terpisah dari data latih, dengan menggunakan metrik evaluasi berbasis klasifikasi seperti *confusion matrix, accuracy, precision, recall, dan F1-score* [16].

1. Confusion Matrix

   Confusion matrix digunakan untuk menggambarkan performa model dalam memprediksi setiap kelas dengan membandingkan label aktual dan hasil prediksi. Matriks ini terdiri dari baris sebagai label aktual dan kolom sebagai hasil prediksi. Nilai pada diagonal utama menunjukkan jumlah prediksi yang benar (*true classification*), sedangkan nilai di luar diagonal menunjukkan kesalahan klasifikasi (*misclassification*) [16].

   Selain *multiclass confusion matrix*, digunakan juga *binary confusion matrix* dengan pendekatan *one-vs-rest* untuk setiap kelas. Matriks ini terdiri dari komponen *True Positive* (TP), *False Positive* (FP), *False Negative* (FN), dan *True Negative* (TN), yang menjadi dasar dalam perhitungan metrik evaluasi lainnya [18].

2. Accuracy

   Accuracy merupakan metrik yang digunakan untuk mengukur tingkat ketepatan model secara keseluruhan dalam melakukan klasifikasi. *Accuracy* dihitung sebagai perbandingan antara jumlah prediksi yang benar dengan total seluruh data [16].

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (2.5)$$

3. Precision, Recall, dan F1-Score

Untuk memberikan evaluasi yang lebih komprehensif, digunakan metrik *precision*, *recall*, dan *F1-score* yang dihitung berdasarkan nilai TP, FP, dan FN dari *binary confusion matrix* [18].

$$Precision = \frac{TP}{TP+FP} \quad (2.6)$$

$$Recall = \frac{TP}{TP+FN} \quad (2.7)$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (2.8)$$

Dalam konteks keamanan jaringan, *Random Forest* banyak digunakan untuk melakukan klasifikasi lalu lintas jaringan berdasarkan karakteristik fitur tertentu, seperti jumlah paket yang dikirimkan, ukuran paket, durasi koneksi, serta pola komunikasi antar host. Dengan mempelajari pola dari data lalu lintas jaringan yang telah diberi label, model *Random Forest* dapat digunakan untuk mengidentifikasi aktivitas yang mencurigakan, termasuk serangan DDoS yang ditandai dengan peningkatan lalu lintas jaringan secara tidak normal.

Pada penelitian ini, algoritma *Random Forest* digunakan sebagai model klasifikasi untuk mengidentifikasi aktivitas lalu lintas jaringan yang berpotensi merupakan serangan DDoS. Model yang telah dilatih kemudian diintegrasikan ke bagian *pipeline* yang memprosesan data yaitu Spark Structured Streaming. Dengan pendekatan tersebut, sistem diharapkan mampu melakukan analisis data lalu lintas jaringan secara *near real-time* serta tepat mendeteksi anomali yang mengindikasikan serangan terhadap sistem jaringan.

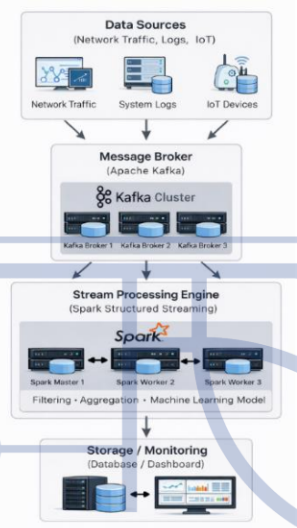
2.6 Pipeline Data Streaming

Perkembangan teknologi menghasilkan aliran data yang besar dan kontinu dari berbagai sumber seperti aplikasi, sistem jaringan, dan perangkat *Internet of Things* (IoT). Kondisi ini menuntut pendekatan pemrosesan data yang mampu bekerja secara *real-time*, yang dikenal sebagai *stream processing* [19]. Berbeda dengan *batch processing*, pendekatan ini memungkinkan data diproses segera setelah diterima dengan *latency* yang rendah.

Dalam implementasinya, *stream processing* diwujudkan dalam bentuk *data streaming pipeline*, yaitu alur pemrosesan data yang mengintegrasikan beberapa komponen utama secara berkelanjutan. *Pipeline* ini dibangun di atas konsep *distributed stream*

processing, di mana pemrosesan dilakukan secara paralel pada beberapa *node* untuk meningkatkan skalabilitas dan performa sistem [20].

Secara umum, arsitektur *pipeline* ditunjukkan pada gambar berikut.



Gambar 2. 4 Arsitektur data pipeline secara umum [20, 21, 22]

Berdasarkan Gambar 2.4, proses dimulai dari data source yang berfungsi sebagai sumber aliran data. Data tersebut kemudian dikirimkan ke *message broker*, seperti Apache Kafka, yang berperan dalam mendistribusikan data secara *real-time* ke berbagai komponen pemrosesan. [20].

Selanjutnya, data diproses oleh *stream processing engine*, seperti Apache Spark Structured Streaming, yang melakukan berbagai operasi analisis seperti *filtering*, agregasi, transformasi data, hingga penerapan model *machine learning*. Tahapan ini merupakan inti dari *distributed stream processing*, di mana pemrosesan dilakukan secara paralel pada beberapa *node* dalam suatu klaster untuk mencapai performa yang optimal [20, 21, 22].

Beberapa teknologi telah dikembangkan untuk mendukung implementasi *data streaming pipeline*, salah satu di antaranya Apache Spark Structured Streaming. Teknologi ini dirancang untuk mampu memproses data secara *real-time* dengan dukungan pemrosesan terdistribusi [8]. Dalam konteks keamanan jaringan, *data streaming pipeline* memiliki peran penting dalam melakukan monitoring lalu lintas jaringan secara berkelanjutan. Dengan memanfaatkan pendekatan *distributed stream processing*, sistem mampu mendeteksi pola serangan seperti DDoS secara lebih cepat dibandingkan pendekatan berbasis *batch processing*.

Pada penelitian ini, konsep *distributed stream processing* digunakan sebagai dasar dalam pengembangan *pipeline* jaringan yang mampu memproses aliran data lalu lintas jaringan secara kontinu. Implementasi sistem dilakukan dengan memanfaatkan Apache

Kafka sebagai *message broker* untuk mendistribusikan *data streaming* serta Apache Spark Structured Streaming sebagai stream *processing engine* untuk melakukan pemrosesan dan analisis data secara *near real-time*.

2.7 Apache Kafka

Apache Kafka merupakan *distributed event streaming plarform* yang menggunakan pola arsitektur *publish-subscribe*. Kafka dikembangkan oleh LinkedIn dan kemudian menjadi proyek *open-source* di bawah Apache Software Foundation. Kafka dirancang untuk menangani aliran data dalam jumlah besar dengan tingkat *throughput* yang tinggi serta latency yang rendah. Dengan arsitektur yang terdistribusi, Kafka mampu menangani jutaan pesan per detik serta menyediakan mekanisme penyimpanan log yang tahan terhadap kegagalan sistem [23], [21].

Secara konseptual, Apache Kafka bekerja sebagai *message broker*, yaitu sistem perantara yang bertugas menerima, menyimpan, dan mendistribusikan pesan antara *producer* dan *consumer* [20]. Dalam implementasinya, istilah *broker* pada Kafka merujuk pada server Kafka itu sendiri, yaitu *node* dalam klaster yang menyimpan data dalam bentuk *topic* dan melayani permintaan baca/tulis dari *client*. Dengan demikian, *message broker* merupakan konsep umum, sedangkan Kafka *broker* adalah implementasi konkretnya dalam sistem Kafka. Data yang dikirimkan oleh *producer* akan disimpan dalam struktur *log* yang disebut *topic*, dan kemudian dapat dikonsumsi oleh *consumer* sesuai kebutuhan aplikasi [20], [21]. Selain itu, Kafka menyediakan *fault tolerance* melalui replikasi data antar *broker* sehingga data tetap tersedia meskipun terjadi kegagalan pada salah satu *node* [21].

Pada perkembangan terbaru, Kafka juga memperkenalkan arsitektur KRaft (*Kafka Raft Metadata mode*) yang menggantikan peran Apache ZooKeeper dalam mengelola *metadata* klaster. KRaft menggunakan protokol konsensus Raft untuk mengelola konfigurasi, *metadata* topik, serta koordinasi antar *broker* secara internal, sehingga menyederhanakan arsitektur sistem dan meningkatkan efisiensi operasional [21]. Dengan pendekatan ini, Kafka menjadi lebih mandiri tanpa ketergantungan pada sistem eksternal.

Selain konsep dasar, Apache Kafka juga menyediakan berbagai parameter konfigurasi pada sisi *producer* yang digunakan untuk mengatur mekanisme pengiriman data. Parameter tersebut meliputi pengaturan batching (*batch.size*), waktu tunggu pengiriman (*linger.ms*), kompresi data (*compression.type*), alokasi memori (*buffer.memory*), serta strategi distribusi data ke partisi (*partitioner*). Pengaturan ini berperan penting dalam

menentukan performa sistem, khususnya dalam aspek *throughput* dan *latency* pada proses pengiriman data [24].

Dalam konteks pemrosesan *data streaming*, Kafka berperan sebagai penghubung antara sumber data dan *processing engine*. Data yang dikirim oleh *producer* akan disimpan sementara dalam *topic* sebelum dikonsumsi oleh sistem pemrosesan seperti Apache Spark Structured Streaming [8]. Pada implementasi penelitian ini, Kafka digunakan sebagai *message broker* untuk mendistribusikan data trafik jaringan. *Producer* dikonfigurasi untuk mengirim data ke *topic* yang dibuat dengan pengaturan *bootstrap servers* sebagai alamat klaster Kafka serta nama *topic* tujuan. Selain itu, digunakan mekanisme pengaturan laju data (*input rate*) pada sisi producer untuk mengontrol jumlah data yang dikirim per detik.

2.8 Apache Avro

Apache Avro merupakan *framework* serialisasi data yang dikembangkan dalam ekosistem Apache Hadoop dan banyak digunakan pada sistem *distributed data processing*, termasuk sistem *streaming* berbasis Apache Kafka. Apache Avro digunakan untuk mengubah data menjadi format biner yang lebih ringkas sehingga ukuran data yang dikirim menjadi lebih kecil dan proses transmisi maupun penyimpanan dapat dilakukan dengan lebih efisien [25].

Apache Avro menggunakan *schema* berbasis JSON untuk mendefinisikan struktur data yang akan diserialisasi maupun di-deserialisasi. Setiap *field* dalam data, seperti nama atribut, tipe data, dan urutan *field*, didefinisikan terlebih dahulu dalam file *schema*. Dengan adanya *schema* tersebut, proses serialisasi dan deserialisasi dapat dilakukan secara konsisten antara *producer* dan *consumer* [20]. Selain itu, Avro mendukung *schema evolution*, yaitu kemampuan untuk menambah, menghapus, atau mengubah *field* tanpa merusak kompatibilitas data yang sudah ada sebelumnya [25].

Salah satu keunggulan Apache Avro dibandingkan format seperti JSON atau XML adalah ukuran data hasil serialisasi yang lebih kecil karena Avro menyimpan data dalam bentuk biner. Format biner tersebut membuat proses pertukaran data menjadi lebih cepat dan lebih hemat *bandwidth*, terutama pada sistem streaming yang memproses data dalam jumlah besar dan berkecepatan tinggi. Selain itu, Avro juga tidak menyimpan nama *field* berulang kali pada setiap record karena struktur *field* telah ditentukan pada schema, sehingga ukuran *payload* menjadi lebih efisien dibandingkan JSON [20].

Dalam implementasi pada Apache Kafka, data yang dikirim oleh *producer* terlebih dahulu diserialisasi ke dalam format Avro sebelum diteruskan ke *broker*. Pada penelitian ini,

proses serialisasi dilakukan menggunakan *custom* Avro *serializer* tanpa menggunakan *schema registry*. *Schema* Avro disimpan dan dikelola secara langsung di dalam aplikasi sehingga *producer* dan *consumer* menggunakan schema yang sama untuk melakukan serialisasi dan deserialisasi data [25]. Pendekatan ini lebih sederhana untuk lingkungan penelitian karena tidak memerlukan layanan tambahan seperti *schema registry*, namun tetap mampu menjaga konsistensi struktur data selama proses pertukaran data berlangsung.

Dalam praktik pengembangan program, penggunaan tipe data Avro mensyaratkan pembuatan berkas dengan ekstensi .avsc terlebih dahulu. Berkas tersebut harus diisi dengan skema data yang merepresentasikan struktur informasi yang akan diproses. Penulisan skema wajib mengikuti standar yang telah ditetapkan dalam dokumentasi resmi Avro, yang mencakup beberapa komponen utama, yaitu: tipe Avro (*type*), nama kelas Avro (*name*), *namespace*, serta daftar *field* yang terdiri atas sejumlah atribut data. Lebih lanjut, aturan penamaan *field* hanya mengizinkan penggunaan karakter alfanumerik dan garis bawah. Setiap *field* juga harus dilengkapi dengan tipe data yang sesuai berdasarkan ketentuan yang termuat dalam dokumentasi resmi tersebut. Berikut penggalan Avro skema yang digunakan pada penelitian ini.

```
{
  "type": "record",
  "name": "DataRecord",
  "namespace": "kudadiri.kafka.serializer",
  "fields": [

    {"name": "Unnamed_0", "type": ["null", "int"], "default": null},
    {"name": "Flow_ID", "type": ["null", "string"], "default": null},
    {"name": "Source_IP", "type": ["null", "string"], "default": null},
    {"name": "Source_Port", "type": ["null", "int"], "default": null},
    {"name": "Destination_IP", "type": ["null", "string"], "default": null},
    {"name": "Destination_Port", "type": ["null", "int"], "default": null},
    {"name": "Protocol", "type": ["null", "int"], "default": null},

    {"name": "producer_timestamp", "type": ["null", "long"], "default": null},
    {"name": "producerTimestampStr", "type": ["null", "string"], "default": null}

  ]
}
```

Gambar 2. 5 Penggalan struktur skema avro

Pada penelitian ini, Apache Avro digunakan untuk melakukan serialisasi data sebelum data dikirimkan oleh *producer* menuju *broker* Kafka. Data yang semula berbentuk struktur atribut akan diubah menjadi format biner Avro agar ukuran *payload* lebih kecil dan lebih efisien saat ditransmisikan melalui jaringan. Setelah diterima oleh *consumer* (Spark Structured Streaming), data tersebut akan dideserialisasi kembali ke dalam bentuk struktur data awal untuk diproses lebih lanjut.

2.9 Logging

Logging merupakan proses pencatatan aktivitas yang terjadi pada sistem selama proses berjalan [20]. Informasi yang dicatat dapat berupa waktu eksekusi, status proses, pesan kesalahan, hingga detail aliran data yang diproses oleh sistem. Dalam penelitian

berbasis *data streaming*, *logging* penting dilakukan untuk membantu proses monitoring, *debugging*, dan evaluasi performa sistem, seperti mendeteksi keterlambatan proses, kegagalan pengiriman data, maupun *error* pada *pipeline* [21].

Dalam penelitian ini, *Logging* digunakan untuk menampilkan hasil *debug*, informasi ketika program berjalan, serta *error* yang terjadi ketika dalam melakukan eksekusi program ataupun kegagalan dalam pengiriman data ke topik Kafka.

2.10 Spark Structured Streaming

Apache Spark merupakan platform *open-source* yang menyediakan layanan terpadu untuk pemrosesan data skala besar secara terdistribusi dengan pendekatan *in-memory computing*. Salah satu modul utamanya adalah Spark Structured Streaming, yaitu *engine* pemrosesan data streaming yang dibangun di atas API Spark SQL dan *DataFrame* sehingga memungkinkan pengolahan *data streaming* dengan paradigma data terstruktur. Pendekatan yang digunakan adalah *stream as a table*, di mana aliran data diperlakukan sebagai tabel yang terus bertambah dan dapat diproses menggunakan operasi seperti *filtering*, agregasi, *join*, dan transformasi lainnya [22], [26].

Dalam implementasinya, Spark Structured Streaming dijalankan melalui komponen utama yaitu *SparkSession* yang berfungsi sebagai *entry point* untuk menginisialisasi konfigurasi aplikasi, termasuk pengaturan paralelisme, integrasi dengan sumber data seperti Apache Kafka, serta parameter eksekusi lainnya [22], [26]. Secara arsitektural, *engine* ini menggunakan model *micro-batch*, di mana *data streaming* dikumpulkan dalam interval waktu yang sangat singkat sebelum diproses secara paralel. Proses ini memungkinkan integrasi *pipeline* data yang mencakup pembacaan data, transformasi, hingga penerapan model *machine learning* untuk melakukan prediksi (*inference*) terhadap data yang masuk secara *real-time* [18], [14].

Selain itu, hasil pemrosesan *streaming* dapat disalurkan ke berbagai sistem tujuan melalui mekanisme *output stream* (*sink*), seperti file *system* atau sistem penyimpanan terdistribusi, dengan mode tertentu seperti *append* untuk menambahkan data baru tanpa mengubah data sebelumnya [26]. Untuk menjamin keandalan sistem, Spark Structured Streaming menyediakan mekanisme *fault tolerance* dan *exactly-once processing semantics* melalui fitur *checkpointing*, yang menyimpan status pemrosesan dan *offset* secara *persisten* sehingga sistem dapat melanjutkan proses tanpa kehilangan data saat terjadi kegagalan [22], [26].

Sebelum adanya Structured Streaming, Spark menggunakan pendekatan *discretized streams* (Dstreams) pada Spark Streaming yang relatif lebih kompleks karena tidak terintegrasi dengan API DataFrame. Oleh karena itu, Spark Structured Streaming menjadi pendekatan yang lebih modern karena menawarkan kemudahan pengembangan, skalabilitas tinggi, serta kemampuan pemrosesan data secara *near real-time*. Dalam penelitian ini, Spark Structured Streaming dipilih sebagai *engine* pemrosesan data karena kemampuannya dalam memproses data secara *near real-time*, mendukung skalabilitas tinggi, serta menyediakan integrasi yang baik dengan berbagai sistem sumber *data streaming* seperti Apache Kafka.

2.11 Apache Parquet

Format file Apache Parquet merupakan format penyimpanan data berbasis kolom (*columnar storage*) yang dirancang untuk mendukung proses analisis data dalam jumlah besar secara efisien. Berbeda dengan format baris seperti CSV, Parquet menyimpan data berdasarkan kolom sehingga proses pembacaan hanya dilakukan pada atribut yang diperlukan. Hal ini membuat penggunaan memori menjadi lebih hemat dan waktu pemrosesan menjadi lebih cepat, terutama pada sistem *big data* dan *machine learning* [22], [27].

Selain itu, Parquet juga mendukung kompresi data dan penyimpanan *metadata schema*, sehingga ukuran file menjadi lebih kecil dan struktur data dapat dipertahankan dengan lebih baik [27]. Oleh karena itu, format ini banyak digunakan pada *framework* pengolahan data salah satunya adalah Apache Spark.

2.12 Pemilihan Fitur Model

Pemilihan fitur merupakan tahapan penting dalam pembangunan model *Random Forest* karena berpengaruh langsung terhadap akurasi, waktu pelatihan, serta kemampuan model dalam membedakan trafik normal dan anomali. Pada dataset deteksi intrusi, jumlah atribut umumnya sangat banyak dan sebagian memiliki korelasi tinggi atau informasi yang redundan. Oleh karena itu, pemilihan fitur dilakukan agar model hanya menggunakan atribut yang paling relevan sehingga dapat mengurangi kompleksitas komputasi dan meningkatkan generalisasi model [14].

Kelompok fitur pertama adalah statistik aliran (*flow statistics*). Fitur-fitur ini digunakan untuk menggambarkan karakteristik umum lalu lintas jaringan, seperti durasi komunikasi, jumlah paket, serta intensitas transfer data. Pada serangan seperti DDoS atau *flooding*, jumlah paket dan byte per detik biasanya meningkat secara signifikan sehingga

pola anomali dapat dikenali dengan lebih mudah. Selain itu, fitur berbasis waktu dipilih karena mampu menunjukkan interval kedatangan paket yang abnormal, misalnya paket yang datang terlalu cepat atau terlalu seragam [13].

Kelompok kedua adalah fitur ukuran paket. Fitur ini penting karena berbagai jenis serangan memiliki pola ukuran paket yang khas. Sebagai contoh, trafik serangan *flooding* sering menggunakan ukuran paket yang seragam dan kecil, sedangkan trafik normal cenderung lebih bervariasi [13].

Kelompok berikutnya adalah fitur *header dan subflow*. Fitur-fitur ini dipilih karena dapat merepresentasikan struktur komunikasi dua arah antara pengirim dan penerima. Trafik serangan umumnya menunjukkan ketidakseimbangan jumlah paket atau *byte* pada salah satu arah komunikasi [14], [13].

Terakhir, fitur TCP *flag* dan *window size* seperti dipilih karena mampu menangkap perilaku koneksi TCP yang tidak normal. Serangan *SYN Flood*, misalnya, ditandai dengan jumlah SYN yang tinggi tanpa diikuti ACK yang memadai. Nilai *initial window size* juga dapat menunjukkan pola komunikasi abnormal yang berkaitan dengan eksploitasi koneksi atau *scanning* [14], [13].

Berdasarkan referensi yang ada, diputuskan kolom-kolom data pada dataset yang digunakan sebagai fitur model *Random Forest* seperti ditunjukkan pada tabel 2.2 berikut.

Tabel 2. 2 Pemilihan Fitur [13], [14]

No	Jenis	Fitur Kolom	Penjelasan Singkat
1	Statistik Aliran	1. Flow Duration (0) 2. Total Fwd Packets (1) 3. Total Backward Packets (2) 4. Flow Bytes/s (3) 5. Flow Packets/s (4)	Menggambarkan durasi, intensitas data, serta jumlah paket maju dan mundur dalam suatu aliran. Berguna untuk mendeteksi lonjakan trafik seperti DDoS.
2	Ukuran Paket	1. Min Packet Length (5) 2. Max Packet Length (6) 3. Packet Length Mean (7) 4. Packet Length Std (8) 5. Packet Length Variance (9)	Merepresentasikan distribusi ukuran paket. Serangan flooding cenderung menghasilkan ukuran paket yang seragam, sedangkan trafik normal lebih bervariasi.

3	Interval Kedatangan Paket (IAT)	1. Flow IAT Mean (10) 2. Flow IAT Std (11) 3. Fwd IAT Mean (12) 4. Fwd IAT Std (13) 5. Bwd IAT Mean (14) 6. Bwd IAT Std (15)	Menunjukkan pola waktu antar kedatangan paket. Anomali seperti serangan brute-force atau scanning menghasilkan interval yang tidak wajar (terlalu cepat atau seragam).
4	TCP Flag	1. SYN Flag Count (16) 2. RST Flag Count (17) 3. ACK Flag Count (18)	Digunakan untuk mengidentifikasi pola serangan berbasis protokol TCP.
5	Header dan Subflow	1. Fwd Header Length (19) 2. Bwd Header Length (20) 3. Subflow Fwd Packets (21) 4. Subflow Bwd Packets (22) 5. Subflow Fwd Bytes (23) 6. Subflow Bwd Bytes (24)	Merepresentasikan struktur dan distribusi aliran data.
6	Window Size TCP	1. Init Win bytes forward (25) 2. Init Win bytes backward (26)	Menggambarkan kapasitas awal jendela transmisi pada komunikasi TCP.

2.13 Teknik Penanganan data

1. Data Cleaning

   Data cleaning merupakan proses pembersihan data untuk menghilangkan nilai hilang, data duplikat, inkonsistensi format, dan noise agar kualitas data menjadi lebih baik sebelum dianalisis . Dalam sistem deteksi serangan jaringan, *data cleaning* penting dilakukan untuk memastikan bahwa data yang diproses benar-benar valid dan tidak mengandung kesalahan yang dapat memengaruhi hasil prediksi model *machine learning* [28].

2. Data Enrichment

   Data enrichment merupakan proses penambahan atribut atau informasi baru ke dalam dataset untuk meningkatkan nilai informatif data [29]. Pada data trafik jaringan, pengayaan dapat dilakukan dengan menambahkan atribut seperti durasi koneksi, rata-

rata paket per detik, atau jenis protokol sehingga model *machine learning* dapat mengenali pola trafik normal dan serangan dengan lebih baik [30].

3. Data Transformation

Data transformation adalah proses mengubah format atau struktur data agar sesuai dengan kebutuhan sistem dan algoritma yang digunakan. Bentuk transformasi yang umum dilakukan meliputi normalisasi, standarisasi, *encoding* data kategorikal, dan konversi tipe data agar seluruh atribut dapat diproses secara konsisten oleh sistem analitik maupun *machine learning* [31].

4. Pemilihan Data

Pemilihan data merupakan proses menentukan data yang akan digunakan sesuai kebutuhan penelitian. Pada penelitian deteksi serangan DDoS, pemilihan data dilakukan dengan memilih data normal dan beberapa jenis serangan tertentu agar jumlah data lebih terkontrol. Tahap ini juga penting untuk mencegah *data imbalance*, yaitu kondisi ketika jumlah data pada suatu kelas jauh lebih banyak dibandingkan kelas lainnya, karena dapat menyebabkan model lebih cenderung memprediksi kelas dengan jumlah data terbesar [29].

5. Konversi Timestamp ke Long

Konversi *timestamp* ke *long* merupakan salah satu bentuk transformasi data yang dilakukan dengan mengubah format waktu menjadi bilangan milidetik sejak *epoch time*, yaitu 1 Januari 1970 pukul 00:00:00 UTC [20]. Representasi ini lebih efisien untuk proses komputasi karena memungkinkan perhitungan selisih waktu atau *latency* dilakukan secara cepat dan konsisten pada sistem *streaming* [32].

6. Feature Engineering

Feature engineering merupakan proses memilih atau membentuk atribut yang relevan agar model *machine learning* dapat mengenali pola data dengan lebih baik [33]. Pada penelitian deteksi serangan DDoS, tahap ini dilakukan dengan memilih atribut penting seperti jumlah paket, jumlah *byte*, durasi koneksi, dan atribut trafik lainnya yang dianggap berpengaruh terhadap proses klasifikasi [13].

7. Split Data Berdasarkan Delimiter

Split data berdasarkan *delimiter* merupakan proses pemisahan data teks menjadi beberapa atribut dengan menggunakan karakter pemisah tertentu, seperti koma, titik koma, tab, atau karakter lainnya. Proses ini umumnya dilakukan pada data mentah yang masih berbentuk satu baris teks, misalnya data CSV atau log jaringan, sehingga setiap nilai dapat dipisahkan ke dalam kolom yang sesuai. Dengan melakukan split

berdasarkan delimiter, data menjadi lebih terstruktur dan lebih mudah diproses pada tahap analisis maupun pelatihan model *machine learning* [28].

8. Split Dataset Model

Split data untuk model merupakan proses pembagian dataset menjadi dua bagian, yaitu data latih dan data uji [34]. Data latih digunakan untuk membangun model *machine learning*, sedangkan data uji digunakan untuk mengevaluasi kemampuan model dalam melakukan prediksi pada data yang belum pernah dilihat sebelumnya. Pembagian data ini penting untuk menghindari overfitting, yaitu kondisi ketika model terlalu menyesuaikan diri terhadap data latih sehingga performanya menurun pada data baru. Umumnya, pembagian dilakukan dengan proporsi tertentu, seperti 80:20 atau 70:30, tergantung pada jumlah data yang tersedia dan kebutuhan penelitian [31].

9. Serialisasi dan Deserialisasi Data

Serialisasi adalah proses mengubah data atau objek menjadi format *byte stream* agar dapat dikirim atau disimpan, sedangkan deserialisasi adalah proses mengembalikannya ke bentuk semula untuk diproses oleh sistem [20]. Dalam sistem *streaming*, proses ini penting untuk memastikan data dapat berpindah antar komponen seperti *producer*, *broker*, dan *consumer* secara konsisten. Format seperti Avro sering digunakan karena mendukung skema data yang terstruktur, sehingga meminimalkan kesalahan saat pertukaran data [25]. Pada sisi pemrosesan, deserialisasi memungkinkan data yang diterima diubah menjadi struktur yang dapat diolah lebih lanjut, seperti *DataFrame* pada Spark Structured Streaming [22].

Pada Apache Avro, proses serialisasi dilakukan menggunakan teknik *encoding* yang disesuaikan dengan tipe data untuk mencapai efisiensi penyimpanan dan transmisi. Untuk tipe numerik seperti *integer* dan *long*, digunakan kombinasi *ZigZag encoding* dan *variable-length integer* (VarInt), di mana nilai n ditransformasikan menggunakan persamaan

$$Z(n) = (n \ll 1) \oplus (n \gg 31) \quad (2.10)$$

Tipe *boolean* direpresentasikan dalam satu *byte*, sedangkan tipe *null* tidak menghasilkan representasi *byte*. Untuk tipe *string*, Avro menggunakan pendekatan *length-prefix encoding*, yaitu panjang *string* $|S|$ dikodekan sebagai *VarInt*, kemudian diikuti oleh representasi UTF-8, yang dapat dinyatakan sebagai.

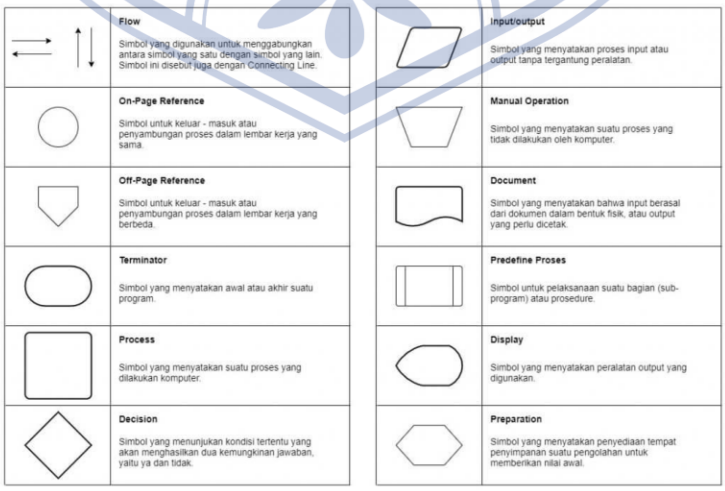
$$encode(S) = VarInt\big(Z(|S|)\big) \,||\, UTF8(S) \quad (2.11)$$

Selain itu, tipe kompleks seperti *array* dan *map* dikodekan dalam bentuk blok (*block encoding*) [20], [25].

Pada sisi deserialisasi, proses dilakukan dengan membaca *byte stream* secara berurutan sesuai skema yang digunakan, kemudian menerapkan *invers* dari setiap teknik *encoding*. Nilai *integer* diperoleh kembali melalui proses *VarInt decoding* yang diikuti dengan *invers ZigZag*, panjang data digunakan untuk menentukan batas pembacaan *byte* pada tipe *string* atau *array*, dan struktur kompleks direkonstruksi berdasarkan definisi skema. Dengan demikian, skema berperan sebagai acuan utama dalam menginterpretasikan data biner menjadi struktur yang dapat diproses lebih lanjut, misalnya dalam bentuk *DataFrame* pada Spark Structured Streaming [22], [25]. Namun, perhitungan *byte array* secara manual umumnya hanya realistis untuk kasus sederhana. Pada skenario yang melibatkan *nested record*, *array*, *union*, *null handling*, maupun *logical type* seperti *timestamp* dan *decimal*, proses *encoding* menjadi lebih kompleks karena melibatkan mekanisme tambahan seperti penanda tipe (*type marker*), *block encoding*, serta *schema resolution*, sebagaimana dijelaskan dalam spesifikasi Apache Avro dan pembahasan sistem data modern [20], [25].

2.14 FlowChart

Flowchart merupakan representasi grafis dari suatu proses atau algoritma yang digunakan untuk menggambarkan alur kerja sistem secara sistematis dan terstruktur. *Flowchart* membantu dalam memahami urutan langkah-langkah yang dilakukan dalam suatu proses, mulai dari *input*, pemrosesan, hingga *output*. Dalam pengembangan sistem, *flowchart* sering digunakan sebagai alat bantu perancangan untuk memvisualisasikan logika program sebelum diimplementasikan ke dalam kode [35].



Gambar 2. 6 Simbol-simbol *flowchart* [36]

Secara umum, *flowchart* terdiri dari beberapa simbol standar, seperti *terminator* (menunjukkan awal dan akhir proses), *process* (menunjukkan langkah pemrosesan), *decision* (menunjukkan percabangan kondisi), serta *flowline* (menunjukkan arah aliran proses). Penggunaan simbol-simbol ini memungkinkan representasi proses menjadi lebih jelas dan mudah dipahami oleh pengembang maupun pemangku kepentingan lainnya [37]. Berikut selengkapnya simbol-simbol dalam *flowchart*. Dalam konteks penelitian ini, *flowchart* digunakan untuk menggambarkan alur kerja *pipeline data streaming*, mulai dari proses pengambilan data melalui Kafka, pemrosesan data menggunakan Spark, hingga tahap klasifikasi menggunakan model *Random Forest*.

2.15 Chevron FlowChart

Chevron flowchart merupakan jenis *flowchart* yang digunakan untuk menggambarkan tahapan proses secara berurutan dalam bentuk panah atau *chevron* yang saling terhubung. Setiap *chevron* mewakili satu langkah atau aktivitas tertentu, sehingga alur proses dapat ditampilkan secara ringkas, jelas, dan mudah dipahami. *Chevron flowchart* umumnya digunakan untuk menunjukkan proses yang bersifat linear, seperti tahapan pengolahan data, alur kerja sistem, maupun langkah-langkah implementasi penelitian. *Chevron flowchart* memiliki tampilan yang lebih sederhana dan lebih cocok digunakan untuk memvisualisasikan proses tingkat tinggi tanpa terlalu banyak detail teknis [38].

2.16 Evaluasi Performa Sistem

Evaluasi performa pada *Data Streaming Pipeline* dilakukan untuk menilai kemampuan *pipeline* dalam memproses aliran data secara efisien dan stabil, terutama saat terjadi peningkatan beban data. Evaluasi ini penting untuk memastikan bahwa sistem mampu melakukan analisis data secara cepat sehingga deteksi serangan dapat dilakukan secara tepat waktu [3], [8].

Secara umum, evaluasi dilakukan menggunakan pendekatan *performance benchmarking*, yaitu pengujian sistem pada berbagai tingkat beban untuk mengamati karakteristik performanya. Dalam konteks *pipeline data streaming*, metrik utama yang digunakan adalah *latency* dan *throughput*, yang merepresentasikan kecepatan pemrosesan data serta kemampuan sistem dalam menangani volume data secara berkelanjutan [8].

1. Latency

   Latency merupakan metrik yang digunakan untuk mengukur waktu yang dibutuhkan sistem dalam memproses suatu data sejak data tersebut dihasilkan hingga

menghasilkan *output* akhir. Dalam sistem, *latency* sering digunakan sebagai indikator utama responsivitas sistem terhadap data yang masuk [3]. Secara Matematis, *latency* dapat dirumuskan sebagai:

$$Latency = T_{output} - T_{input} \quad \text{.....(2. 12)}$$

di mana,

a. T_{input} adalah waktu saat data dihasilkan,

b. T_{output} adalah waktu saat hasil pemrosesan dari data selesai.

Dalam penelitian ini, *latency* diukur terhadap 3 bagian yaitu, Kafka, Spark, dan *end-to-end pipeline*. Untuk mendapatkan nilai *latency* setiap data, diperlukan pencatatan beberapa titik waktu (*timestamp*) dalam proses *pipeline*, dimana beberapa diantaranya meliputi:

a. *Producer Timestamp* (*source*), waktu data dikirim ke *broker* atau server Kafka;

b. *Broker Timestamp*, waktu data telah diterima oleh *broker* Kafka;

c. *Processing End Time* (*sink*), waktu hasil prediksi selesai disimpan.

Sehingga, setiap bagian dirumuskan sebagai berikut:

$$\text{Latency}_{\text{producer}} = T_{\text{broker timestamp}} - T_{\text{source}} \quad \text{.....(2. 13)}$$

$$\text{Latency}_{\text{E2E}} = T_{\text{sink}} - T_{\text{source}} \quad \text{.....(2. 14)}$$

Sedangkan untuk *Latency* Spark terhitung dari awal data diambil dari *topic* sampai selesai diproses.

Pendekatan ini banyak digunakan dalam evaluasi sistem *streaming* karena mampu merepresentasikan total waktu pemrosesan data dari sumber hingga *output* akhir secara menyeluruh [3], [8]. Nilai *latency* yang rendah menunjukkan bahwa sistem mampu merespons perubahan data secara cepat, yang sangat penting dalam aplikasi monitoring jaringan secara *real-time* [8].

Namun dalam praktik evaluasi latency, perlu dilakukan proses agregasi statistik untuk menentukan nilai *latency* akhir dari setiap data yang diperoleh. Pengukuran *latency* tidak cukup hanya mengandalkan nilai rata-rata (*average latency*). Pada sistem *distributed streaming*, variasi beban, antrean data (*queueing*), serta proses komputasi pada *engine* seperti Apache Kafka dan Spark Structured Streaming, termasuk proses inferensi model *machine learning*, dapat menyebabkan sebagian kecil data mengalami keterlambatan yang signifikan. Kondisi ini sering kali tidak tercermin dalam nilai rata-rata, sehingga berpotensi memberikan gambaran performa sistem yang kurang representatif atau bahkan menyesatkan [39].

Oleh karena itu, nilai rata-rata *latency* tidak digunakan sebagai dasar perhitungan. Sebaliknya, digunakan pendekatan statistik berbasis distribusi seperti *percentile latency*, misalnya p50, p95, dan p99. Nilai p95 dan p99 merepresentasikan *tail latency*, yaitu kondisi di mana sebagian kecil data mengalami waktu pemrosesan paling lambat. Pendekatan ini memberikan gambaran yang lebih akurat terhadap performa sistem secara keseluruhan, terutama dalam kondisi beban tinggi dan pada sistem yang melibatkan banyak komponen seperti Kafka, Spark, dan model *machine learning* [39].

Dalam konteks pengukuran *latency* berbasis distribusi, nilai *percentile latency* dihitung berdasarkan urutan data *latency* yang telah diurutkan secara menaik. Misalkan terdapat data *latency* sebanyak n yang telah diurutkan secara menaik $\{l_1, l_2, l_3, ..., l_n\}$. Maka nilai *percentile* ke-p dapat dihitung dengan menentukan posisi indeks sebagai berikut:

$$K = \left\lceil \left(\frac{p}{100}\right) \times n \right\rceil \quad (2.15)$$

di mana,

a. K adalah indeks data pada urutan ke-k,
b. p adalah nilai *percentile* (misalnya 95 atau 99), dan
c. n adalah jumlah total data *latency*.

Nilai p95 menunjukkan bahwa 95% data memiliki *latency* lebih kecil atau sama dengan nilai tersebut, sedangkan p99 menunjukkan batas *latency* untuk 99% data. Oleh karena itu, p99 cenderung lebih sensitif terhadap nilai ekstrem (*outlier*) dibandingkan p95. Dalam penelitian ini, analisis *latency* dilakukan dengan menghitung nilai 95th *percentile latency* dari seluruh data yang telah diproses. Diputuskannya nilai p95 *latency* sebagai dasar perhitungan latency, karena metrik ini dianggap mampu menggambarkan kondisi mayoritas sistem secara stabil tanpa terlalu dipengaruhi oleh nilai ekstrem yang jarang terjadi [39].

Walaupun, nilai p99 tidak digunakan sebagai keputusan akhir nilai latency, tetap dicobakan untuk mengidentifikasi kondisi terburuk (*worst-case scenario*) dalam *pipeline*, yang dapat terjadi akibat lonjakan beban, antrean data, atau keterlambatan pada proses komputasi di Spark maupun inferensi model *machine learning*. Dengan demikian, kombinasi penggunaan p95 dan p99 memungkinkan evaluasi performa sistem yang lebih komprehensif, baik dari sisi kondisi normal maupun kondisi ekstrem.

2. Throughput

Throughput merupakan metrik yang digunakan untuk mengukur jumlah data yang dapat diproses oleh sistem dalam satuan waktu tertentu. Metrik ini mencerminkan kapasitas sistem dalam menangani volume data yang besar secara kontinu [3], [8]. Secara matematis, *throughput* dapat dirumuskan sebagai:

$$\text{Througput} = \frac{N}{T} \quad \text{.....(2. 16)}$$

dimana,

a. N adalah jumlah pesan atau data yang diproses,
b. T adalah waktu yang dibutuhkan dalam memproses data.

Dalam konteks sistem *streaming*, *throughput* biasanya dinyatakan dalam satuan *records per second*. Nilai *throughput* yang tinggi menunjukkan bahwa sistem memiliki kemampuan skalabilitas yang baik dalam menangani peningkatan beban data [8].

Pada penelitian ini, *throughput* dianalisis dalam tiga perspektif:

a. *Throughput* Kafka, jumlah pesan yang berhasil diterima oleh broker per detik
b. *Throughput* Spark, jumlah data yang berhasil diproses dan diprediksi per detik (berdasarkan skenario *micro-batch*)
c. *Throughput End-to-End*, jumlah data yang berhasil diproses dari producer hingga menghasilkan *output* akhir per detik

Analisis *throughput* digunakan untuk mengevaluasi skalabilitas sistem, yaitu kemampuan sistem dalam mempertahankan performa ketika terjadi peningkatan beban *input* data [8].

3. Pendekatan dan Kriteria Analisis Performa Sistem

Evaluasi performa pada sistem *distributed stream processing* umumnya tidak memiliki batas baku yang bersifat universal dalam mengkategorikan sistem sebagai baik atau buruk. Namun demikian, praktik industri serta literatur akademik memberikan acuan mengenai karakteristik performa sistem yang dapat digunakan sebagai dasar interpretasi hasil evaluasi [19, 40]. Dalam penelitian ini, analisis performa sistem dilakukan dengan mengacu pada kombinasi *benchmark* industri dan pendekatan ilmiah untuk mengevaluasi perilaku sistem dalam memproses data streaming secara *near real-time*.

a. Aspek Analisis Performa Sistem

Analisis performa dilakukan berdasarkan tiga aspek utama:

i. Kestabilan Sistem

Kestabilan sistem dianalisis berdasarkan konsistensi nilai *latency* dan *throughput* pada berbagai skenario *input rate*. Sistem dikatakan stabil apabila mampu mempertahankan performa tanpa fluktuasi yang signifikan ketika beban meningkat [19].

ii. Identifikasi Bottleneck

Analisis *bottleneck* dilakukan untuk mengidentifikasi komponen dalam *pipeline* yang menjadi titik keterbatasan performa, seperti pada sisi *producer*, *broker* Apache Kafka, maupun *processing engine* Apache Spark Structured Streaming. Indikasi *bottleneck* dapat diamati dari peningkatan *latency* yang tidak proporsional atau penurunan *throughput* [40].

iii. Trade-off Latency dan Throughput

Dalam sistem *streaming*, terdapat hubungan *trade-off* antara *latency* dan *throughput*, di mana peningkatan *throughput* umumnya diikuti oleh peningkatan *latency* akibat antrean dan proses komputasi [19]. Oleh karena itu, sistem yang baik diharapkan mampu menjaga keseimbangan antara kedua metrik tersebut.

b. Kategorisasi Performa Berdasarkan Latency

Terdapat 2 ketentuan kategorisasi yang dapat digunakan pada penelitian ini untuk memberi penilaian atas hasil evaluasi performa dari pipeline data streaming yang dikembangkan.

Tabel 2. 3 Kategorisasi *Latency* komponen Kafka [40]

Rentang Waktu	Kategori	Interpretasi
< 100 ms	Sangat baik (Real-time)	Pipeline sangat responsif dalam menangani data
100 ms – 500 ms	Baik (Near Real-time)	Pipeline masih responsif dan cepat menangani data
500 ms – 2 second	Cukup	Masih dalam batas wajar waktu proses data
2 – 5 second	Kurang	Terjadi delay yang mulai terasa
> 5 second	Buruk	Mengindikasikan Bottleneck pada pipeline

Latency pada tingkat komponen menggambarkan performa dari masing-masing bagian sistem secara individual. Dalam sistem *streaming* modern, latency

di bawah 100 ms umumnya dikategorikan sebagai *real-time* karena sistem mampu merespons data secara hampir instan [40]. Namun, pada sistem berbasis *micro-batch* seperti Apache Spark Structured Streaming, *latency* cenderung berada pada rentang ratusan milidetik hingga beberapa detik karena adanya proses pengelompokan data dalam *batch* kecil sebelum diproses [6].

Oleh karena itu, latency hingga 2 detik masih dapat dianggap sebagai kondisi normal dalam konteks pemrosesan *streaming* berbasis *micro-batch*. Ketika *latency* melebihi 5 detik, hal ini mengindikasikan adanya bottleneck dalam sistem, yang dapat disebabkan oleh keterbatasan sumber daya, beban data yang tinggi, atau konfigurasi sistem yang belum optimal [19], [40], [41].

Tabel 2. 4 Kategorisasi *latency Spark dan pipeline E2E near real-time* [40], [41].

Rentang Waktu	Kategori	Interpretasi
< 1 second	Sangat baik	Mendekati Real-time
1 – 5 second	Baik	Pipeline dalam kondisi normal
6 – 10 second	Kurang	Mulai terjadi delay
> 10 second	Buruk	Mengindikasikan bottleneck/ pipeline overload

End-to-End latency merupakan metrik yang mengukur total waktu pemrosesan data sejak data dihasilkan oleh producer hingga menghasilkan output akhir. Berbeda dengan latency pada tingkat komponen, *E2E latency* mencakup seluruh tahapan dalam pipeline, termasuk *ingestion delay*, antrian data pada server, *scheduling micro-batch*, serta proses komputasi pada sistem pemrosesan. Dalam sistem berbasis Apache Spark Structured Streaming, mekanisme *micro-batch* menyebabkan adanya waktu tunggu sebelum data diproses, sehingga latency secara alami berada pada rentang detik [6].

Selain itu, berdasarkan model pemrosesan data modern, terdapat *trade-off* antara *latency* dan *throughput*, di mana peningkatan beban data dapat menyebabkan peningkatan *latency* secara signifikan [19]. Dengan mempertimbangkan karakteristik tersebut, *latency* hingga 3 detik masih dapat dikategorikan sebagai kondisi normal dalam sistem *streaming*. Sementara itu, *latency* di atas 6 detik menunjukkan adanya indikasi *bottleneck*, seperti keterlambatan dalam proses *ingestion*, keterbatasan *resource* komputasi, atau *backlog data* dalam sistem.

c. Kategorisasi Performa Berdasarkan Throughput

Dalam memberikan interpretasi yang lebih representatif, digunakan rasio *throughput* terhadap *input rate* untuk mengkategorisasi performa. Sehingga dapat di rumusan sebagai berikut.

$$R = \frac{T_{actual}}{T_{input}} \quad \text{.....(2. 17)}$$

Tabel 2. 5 Kategorisasi *Throughput* [3], [8], [19]

Rasio (R)	Kategori	Interpretasi
$R \geq 0.9$	Tinggi	Sistem mampu merespon hampir seluruh data
$0.6 \leq R < 0.9$	Sedang	Sistem mulai mendekati batas kapasitas
$R < 0.6$	Rendah	Sistem mengalami *bottleneck*

Throughput merupakan indikator kapasitas sistem dalam memproses data secara kontinu [19]. Sistem seperti Apache Kafka dirancang untuk mempertahankan *throughput* tinggi melalui arsitektur terdistribusi. Ketika *throughput* aktual lebih kecil dibandingkan *input rate*, maka akan terjadi antrean data yang menyebabkan peningkatan *latency*, yang mengindikasikan adanya *bottleneck* dalam sistem [19].

2.17 Penelitian Terdahulu

Penelitian mengenai deteksi serangan DDoS terus berkembang seiring meningkatnya kebutuhan terhadap sistem keamanan jaringan yang mampu bekerja secara cepat dan skalabel. Berbagai penelitian memanfaatkan pendekatan *machine learning* dan *distributed stream processing* untuk mendukung proses deteksi serangan secara *near real-time*. Selain itu, integrasi *pipeline data streaming* dengan model klasifikasi juga mulai banyak diterapkan untuk mengatasi keterbatasan pendekatan berbasis batch processing dalam menangani trafik jaringan yang besar dan dinamis. Beberapa penelitian terdahulu yang berkaitan dengan deteksi serangan DDoS, *machine learning*, dan *distributed stream processing* ditunjukkan pada Tabel 2.1 berikut.

Tabel 2. 6 Penelitian Terdahulu

No	Referensi	Penelitian
1	An Adaptive Scalable Data Pipeline for Multiclass Attack Classification in Large-Scale IoT Networks (2024) [2]	Mengembangkan adaptive scalable data pipeline untuk klasifikasi serangan pada jaringan IoT secara real-time.

2	SSK-DDoS: distributed stream processing framework based classification system for DDoS attacks (2022) [4]	Mengembangkan framework deteksi DDoS berbasis Apache Kafka dan Spark Streaming yang mampu mengklasifikasikan trafik jaringan secara real-time menggunakan Spark MLlib pada lingkungan terdistribusi.
3	Benchmarking of Machine Learning for Anomaly Based Intrusion Detection Systems in the CICIDS2017 Dataset (2021) [13]	Melakukan benchmarking algoritma machine learning pada anomaly-based intrusion detection system.
4	Towards real-time ML-based DDoS detection via cost-efficient window-based feature extraction (2024) [14]	Mengimplementasikan Random Forest pada Network Intrusion Detection System (NIDS) untuk deteksi ancaman jaringan.
5	Benchmarking Distributed Stream Processing Frameworks for Real Time Classical Machine Learning Applications (2020) [3]	Melakukan benchmarking Apache Spark Streaming, Apache Flink, dan Apache Storm untuk aplikasi machine learning real-time serta menunjukkan perbedaan performa throughput dan latency pada masing-masing framework distributed stream processing.

Berdasarkan penelitian terdahulu tersebut, diketahui bahwa *machine learning* dan *distributed stream processing* memiliki potensi dalam mendukung deteksi serangan DDoS secara *near real-time*. Namun, implementasi *pipeline data streaming end-to-end* berbasis Apache Kafka dan Spark Structured Streaming yang disertai evaluasi performa metrik *processing latency* dan *throughput* masih terbatas. Oleh karena itu, penelitian ini berfokus pada pengembangan *pipeline data streaming* untuk deteksi serangan DDoS *secara near real-time.*