Threat Intelligence Correlation Engine (TICE)

---

1. Executive Summary

The **Threat Intelligence Correlation Engine (TICE)** is an automated software solution designed to address a critical inefficiency in modern cybersecurity operations. Currently, security analysts must manually investigate suspicious IP addresses by cross-referencing them on numerous, disparate security websites (like VirusTotal, AbuseIPDB, etc.), a process that is slow, repetitive, and error-prone.

TICE solves this problem by functioning as a central, high-speed aggregator. Upon receiving a single IP address, the TICE backend engine concurrently queries multiple threat intelligence APIs , gathers all the "messy" JSON responses , and "normalizes" them into a single, unified profile.

It then applies a weighted scoring algorithm to generate a simple, actionable **"Danger Score"** (e.g., 85/100) and a qualitative **verdict** (e.g., "Malicious," "Safe"). This final, consolidated report is delivered to the analyst via a clean web dashboard and a secure REST API. The primary business value of TICE is its ability to provide analysts with a rapid, high-confidence assessment, **significantly reducing investigation time**[10].

---

2. Project Overview & Objectives

2.1 Problem Statement

A Security Operations Center (SOC) analyst or Incident Responder who finds a suspicious IP address must manually check it on 10 or more different websites to build a complete profile. This manual process is a key bottleneck in threat investigation[12].

2.2 Proposed Solution

TICE is a standalone system composed of two primary parts:

1. **Backend "Engine":** A REST API service that receives an IP, does all the work of calling other APIs, and calculates the score.

2. **Frontend "Dashboard":** A simple, web-based user interface where an analyst can enter the IP and see the final, visualized report.

2.3 Key Objectives

- **Aggregation:** Concurrently query 2-3+ external security APIs (e.g., VirusTotal, AbuseIPDB) for a given IP address.

- **Normalization:** "Translate" the different JSON responses from each source into one standardized, simple format.

- **Scoring:** Apply a simple, configurable logic to generate a final quantitative score (e.g., 0-100) and a qualitative verdict (e.g., "Malicious").

- **Presentation:** Expose this final report via both a REST API endpoint and a clean, user-friendly dashboard.

---

3. Technical Architecture

The system operates on a standard client-server model. The frontend (client) communicates with the TICE backend, which in turn acts as a client to multiple third-party external APIs.

3.1 System Components

- **Frontend (Dashboard):** A single-page application (SPA) that provides the user interface. It contains an input field for the IP address and a "Submit" button. It is responsible for calling the TICE backend API and visualizing the returned JSON report , showing key data like the final score (as a gauge) and verdict (as a color-coded banner).

- **Backend (Engine):** The core of the project, built as a REST API. This engine receives the IP from the frontend , calls all external APIs asynchronously , runs the normalization and scoring logic , and returns the final consolidated report.

- **External APIs:** Third-party threat intelligence services (e.g., VirusTotal, AbuseIPDB) that TICE queries for data. TICE must authenticate to these services using secret API keys.

3.2 Technology Stack

As per the project constraints, the technology stack is strictly defined: ●

Backend Language: Python

- **Backend Framework: FastAPI** (chosen specifically for its native asynchronous capabilities)
- **Web Server: Uvicorn** (to run the FastAPI application)

- **HTTP Client: httpx** (required for making asynchronous/concurrent API calls)

- **Configuration: python-dotenv** (for securely managing API keys)

3.3 Core Data Flow

1. An analyst enters an IP into the frontend dashboard and clicks "Submit".

2. The frontend's JavaScript calls the TICE backend endpoint: GET /analyze/{ip_address}.

3. The analyze_ip function in the backend is triggered.

4. The backend securely loads API keys from a .env file.

5. Using httpx, the backend launches **parallel (concurrent) API calls** to VirusTotal, AbuseIPDB, and other configured sources. This "all at once" approach is a key performance requirement.

6. As the JSON responses return, they are fed into the **Data Normalization Engine** , which "translates" them into a standard internal format.

7. This list of normalized data is passed to the **Threat Scoring Engine**, which applies a weighted logic to calculate a final_score and verdict.

8. The backend consolidates this into a final JSON "Threat Attribution Report" and returns it.
9. The frontend receives the JSON and displays the results in a human-readable format.

---

4. Key Requirements Highlights

4.1 Functional

- **Main Endpoint:** The system *must* provide a GET /analyze/{ip_address} endpoint.
- **Asynchronous Processing:** All external API calls *must* be performed asynchronously (concurrently).
- **Report Structure:** The final JSON report *must* contain top-level fields including ip_address, final_score, verdict, and source_reports.
- **Dashboard UI:** The UI *must* feature a single IP input and must visualize the final_score (e.g., as a gauge) and verdict (e.g., as a banner).

4.2 Non-Functional

- **Security (Critical):** API keys **must not** be hard-coded. They must be loaded from secure environment variables , as demonstrated by the python-dotenv and os.getenv implementation. Keys must **never** be exposed in API responses or frontend code.
- **Performance:** The total API response time shall be determined by the *slowest single external API query*, not the sum of all queries. The target p95 response time is under 5 seconds.

- **Reliability:** The system *must* be resilient to individual API failures. A timeout or error from one source (e.g., AbuseIPDB) must not terminate the entire analysis; the system should gracefully handle it and return a partial report from the successful sources.

---

5. Implementation & Deployment Plan

5.1 Initial Setup

1. **Install Prerequisites:** Install **Python**, a code editor like **VS Code**, and the required Python libraries via pip:

   - pip install fastapi
   - pip install "uvicorn[standard]"
   - pip install httpx
   - pip install python-dotenv

2. **Acquire API Keys:** Sign up for free accounts at **AbuseIPDB** and **VirusTotal** and copy the generated API keys .

3. **Secure Keys:** Create a file named exactly .env in the project root. Add the keys to this file:
   `ABUSEIPDB_KEY="your-key-from-abuseipdb-goes-here"`
   `VIRUSTOTAL_KEY="your-key-from-virustotal-goes-here"`

4. **Load Keys:** In main.py, use the following code to load keys securely :

```
Python import
os
from dotenv import load_dotenv
load_dotenv() # Loads variables from .env

abuse_key = os.getenv("ABUSEIPDB_KEY")
vt_key = os.getenv("VIRUSTOTAL_KEY")
```

5.2 Development & Deployment

- **Local Development:** The server is run locally from the terminal using the command `uvicorn main:app --reload`. The API will then be accessible at `http://127.0.0.1:8000` (localhost).

- **Public Deployment (for Hackathon):** To allow teammates and judges to access the backend, it must be "deployed" to a public URL. For a hackathon, the fastest and simplest recommended method is **Replit**, which instantly provides a public URL upon running the code. Other free options include Deta Space and GitHub Codespaces.

---

6. Conclusion

The TICE project is a high-value, well-scoped solution that directly addresses a common pain point for cybersecurity analysts. By leveraging a modern, asynchronous Python backend (FastAPI, httpx) and adhering to security best practices for API key management, the project is not only feasible within a hackathon timeframe but also provides a robust foundation for a powerful security tool. The implementation plan is clear, with a direct path from local development to public deployment.