
EchoPulse Performance Heatmap: Timing and Mutation Drift Visualizer in Python

1. Purpose

This document specifies a Python script designed to visualize the performance characteristics of the EchoPulse Key Encapsulation Mechanism (KEM) across multiple sessions. The script will render various timing data (encapsulation, decapsulation, mutation cost) using heatmaps and bar charts, facilitating the identification of performance bottlenecks, subtle timing drifts, and potential non-constant-time behaviors indicative of side-channel vulnerabilities. It aims to provide a clear, visual representation of EchoPulse's efficiency and consistency over time and across different internal cryptographic primitives.

2. Performance Data Visualization (Performance Data Visualizer)

The script will generate several types of plots to provide a comprehensive view of EchoPulse performance.

2.1. Plot Types:

- **Overall Timing Heatmap (Encapsulation/Decapsulation/Total):**
  - **Type:** Heatmap or stacked bar chart.
  - **X-axis:** session_id (representing the t value or chronological session index).
  - **Y-axis:** Operation type (Encapsulation, Decapsulation, Mutation, Total KEM).
  - **Color Gradient:** Represents time in microseconds (µs). Darker/hotter colors indicate longer durations.
  - **Purpose:** Quickly identify which part of the KEM is most expensive and how costs change over time. Anomalous spikes will be immediately visible.
- **Mutation Cost Heatmap:**
  - **Type:** Heatmap.
  - **X-axis:** session_id.
  - **Y-axis:** A single row representing mutation_time.
  - **Color Gradient:** mutation_time in µs.
  - **Purpose:** Specifically analyze the cost of the µ(G,t) function. This is critical for assessing how graph complexity or specific mutation operations affect performance. Spikes here could indicate data-dependent mutation behavior.

- **Hash Backend Contribution (Stacked Bar/Heatmap):**
  - **Type:** Stacked Bar Chart or specialized heatmap.
  - **X-axis:** session_id.
  - **Y-axis:** Total KEM time.
  - **Segments/Colors:** Different colors within each bar (or different rows in a heatmap) represent the time spent in different internal hash functions (e.g., SHA3, BLAKE2s) or other internal cryptographic components.
  - **Purpose:** Understand the relative performance impact of cryptographic primitives chosen for EchoPulse's internal operations. Useful for comparing hash_mode choices.
- **Drift/Anomaly Highlighting:**
  - The plots will optionally highlight sessions where timing metrics deviate significantly (e.g., beyond 2 standard deviations) from the moving average, using distinct markers or border colors. This helps draw attention to potential non-constant-time issues or performance regressions.

3. Python Plot Development (Python Plot Developer)

The script will utilize matplotlib and/or seaborn for rendering plots due to their flexibility and industry standard adoption.

3.1. Libraries:

- pandas: For data loading and manipulation (CSV/JSON).
- matplotlib.pyplot: For core plotting functionalities.
- seaborn: For enhanced heatmaps and statistical plotting.

3.2. Input Format:

The script will accept either a CSV or JSON file as input, defined by the --input argument.

- **CSV Format:**
  Code-Snippet
  session_id,enc_time_us,dec_time_us,mutation_time_us,hash_type
  1,150,160,80,BLAKE2s
  2,152,161,81,BLAKE2s
  3,155,165,85,SHA3
  4,153,163,82,BLAKE2s
  ...

- **JSON Format:**

```JSON
[
  {"session_id": 1, "enc_time_us": 150, "dec_time_us": 160, "mutation_time_us": 80, "hash_type": "BLAKE2s"},
  {"session_id": 2, "enc_time_us": 152, "dec_time_us": 161, "mutation_time_us": 81, "hash_type": "BLAKE2s"},
  {"session_id": 3, "enc_time_us": 155, "dec_time_us": 165, "mutation_time_us": 85, "hash_type": "SHA3"},
  ...
]
```

3.3. Rendering Logic (Pseudocode/Function Signatures):

```Python
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import argparse
import os

def load_data(input_path: str) -> pd.DataFrame:
    """Loads performance data from CSV or JSON."""
    if input_path.endswith('.csv'):
        df = pd.read_csv(input_path)
    elif input_path.endswith('.json'):
        df = pd.read_json(input_path)
    else:
        raise ValueError("Unsupported input file format. Use .csv or .json")
    return df

def plot_overall_timing_heatmap(df: pd.DataFrame, output_path: str, title: str):
    """
    Generates a heatmap for encapsulation, decapsulation, and mutation times per session.
    """
```

```
    df['total_kem_time_us'] = df['enc_time_us'] + df['dec_time_us'] # + df['mutation_time_us'] if mutation is part of KEM call
    plot_df = df[['session_id', 'enc_time_us', 'dec_time_us', 'mutation_time_us']].set_index('session_id')
    # Or create a melted DataFrame for a different plot type, e.g., stacked bar
    # plot_df = df.melt(id_vars=['session_id'], value_vars=['enc_time_us', 'dec_time_us', 'mutation_time_us'],
    #                   var_name='Operation', value_name='Time_us')

    plt.figure(figsize=(12, 6))
    sns.heatmap(plot_df.T, cmap="viridis", annot=False, fmt=".0f", linewidths=.5, cbar_kws={'label': 'Time (µs)'})
    plt.title(title)
    plt.xlabel("Session ID")
    plt.ylabel("Operation")
    plt.tight_layout()
    plt.savefig(output_path)
    plt.close()

def plot_mutation_cost_heatmap(df: pd.DataFrame, output_path: str, title: str):
    """Generates a heatmap specifically for mutation cost per session."""
    plt.figure(figsize=(12, 3))
    sns.heatmap(df[['mutation_time_us']].T, cmap="magma", annot=False, fmt=".0f", linewidths=.5, cbar_kws={'label': 'Mutation Time (µs)'})
    plt.title(title)
    plt.xlabel("Session ID")
    plt.ylabel("Mutation")
    plt.tight_layout()
    plt.savefig(output_path)
    plt.close()

def plot_hash_backend_contribution(df: pd.DataFrame, output_path: str, title: str):
    """
    Generates a bar chart or heatmap showing timing contribution per hash backend.
    This would require more granular data in the input, e.g., `hash_time_us` field.
    For simplicity here, assuming `hash_type` only categorizes total operation.
    A true breakdown needs per-hash timing.
    """
    if 'hash_time_us' not in df.columns:
```

```
        print("Warning: 'hash_time_us' not found for hash backend contribution plot.")
        return

    plt.figure(figsize=(12, 6))
    # Example using grouped bar plot for hash types over sessions
    sns.barplot(data=df, x='session_id', y='hash_time_us', hue='hash_type', palette='Paired')
    plt.title(title)
    plt.xlabel("Session ID")
    plt.ylabel("Hash Time (µs)")
    plt.legend(title="Hash Backend")
    plt.tight_layout()
    plt.savefig(output_path)
    plt.close()

def highlight_anomalies(df: pd.DataFrame, column: str, threshold_std: float = 2.0) -> pd.DataFrame:
    """
    Adds a boolean column to the DataFrame indicating anomalies based on std deviation.
    """
    mean = df[column].mean()
    std = df[column].std()
    df[f'is_anomaly_{column}'] = (df[column] > (mean + threshold_std * std)) | \
                                 (df[column] < (mean - threshold_std * std))
    return df

# Main script execution
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="EchoPulse Performance Heatmap
Visualizer.")
    parser.add_argument("--input", type=str, required=True,
                        help="Path to the input CSV or JSON performance data file.")
    parser.add_argument("--output-dir", type=str, default=".",
                        help="Directory to save output plots.")
    parser.add_argument("--plot-type", type=str, default="all",
                        choices=["all", "overall", "mutation", "hash_contrib"],
                        help="Type of plot to generate.")
    parser.add_argument("--hash-filter", type=str,
                        help="Filter data by hash_type (e.g., BLAKE2s, SHA3).")
    parser.add_argument("--anomaly-threshold", type=float, default=2.0,
```

```
                    help="Standard deviation threshold for anomaly detection.")

    args = parser.parse_args()

    os.makedirs(args.output_dir, exist_ok=True)

    try:
        data = load_data(args.input)

        if args.hash_filter:
            data = data[data['hash_type'] == args.hash_filter]
            if data.empty:
                print(f"No data found for hash_type: {args.hash_filter}. Exiting.")
                exit()

        data_anomalies = highlight_anomalies(data.copy(), 'total_kem_time_us',
args.anomaly_threshold)
        data_anomalies = highlight_anomalies(data_anomalies.copy(), 'mutation_time_us',
args.anomaly_threshold)

        if args.plot_type == "all" or args.plot_type == "overall":
            plot_overall_timing_heatmap(data_anomalies, os.path.join(args.output_dir,
"overall_timing_heatmap.png"),
                                        "EchoPulse KEM Overall Timing per Session")
            print(f"Generated: {os.path.join(args.output_dir, 'overall_timing_heatmap.png')}")

        if args.plot_type == "all" or args.plot_type == "mutation":
            plot_mutation_cost_heatmap(data_anomalies, os.path.join(args.output_dir,
"mutation_cost_heatmap.png"),
                                       "EchoPulse Mutation Cost per Session")
            print(f"Generated: {os.path.join(args.output_dir, 'mutation_cost_heatmap.png')}")

        if args.plot_type == "all" or args.plot_type == "hash_contrib":
            # This plot requires 'hash_time_us' column, which might not be in default input
            if 'hash_time_us' in data.columns:
                plot_hash_backend_contribution(data_anomalies,
os.path.join(args.output_dir, "hash_contribution_plot.png"),
```

```
                                    "EchoPulse Hash Backend Contribution per Session")
            print(f"Generated: {os.path.join(args.output_dir, 'hash_contribution_plot.png')}")
        else:
            print("Skipping hash contribution plot: 'hash_time_us' column not found in input data.")

    except Exception as e:
        print(f"Error: {e}")
```

4. Side-Channel Pattern Analysis (Side-Channel Pattern Analyst)

While direct detection of side-channel attacks is beyond a visualization script, the generated plots can highlight patterns that warrant further investigation.

- **Non-Constant Time Behavior:**
  - **Visualization:** Highly variable times for enc_time_us, dec_time_us, or mutation_time_us across sessions, especially if these variations correlate with specific session_id values, input data entropy (if available in logs), or specific graph parameters.
  - **Detection Method:** Look for wide bands in heatmaps or large standard deviations in timing for a given operation. Anomaly highlighting will flag sessions with unusually high/low performance.
  - **Interpretation:** Significant performance variations across sessions could indicate data-dependent branching or memory access patterns, a common source of timing side channels.
- **Unusual Mutation Cost Spikes:**
  - **Visualization:** Isolated "hot" cells or distinct spikes in the mutation_cost_heatmap.
  - **Detection Method:** Observe if these spikes align with specific graph_id, mutation_index, or graph characteristics (e.g., number of nodes/edges).
  - **Interpretation:** Such spikes might indicate that the mutation function's complexity is not constant, potentially revealing information about the graph's structure or the mutation seed if an adversary can correlate timing with known inputs.
- **Symbol-Based Timing Divergence (If Data Present):**
  - *Requires extended input data:* If the mutation_trace.json or a separate log file provides per-symbol traversal times, the script could be extended to visualize this.
  - **Visualization (conceptual):** A heatmap where X-axis is symbol_index_in_path, Y-axis is session_id, and color is time_per_symbol.

- **Detection Method:** Look for specific symbols consistently taking longer to traverse or mutate.
- **Interpretation:** This could indicate data-dependent costs associated with processing specific symbols or traversing certain graph structures, potentially exploitable if an adversary can influence or observe symbol usage.

5. Script Output Design (Script Output Designer)

The script prioritizes ease of use and reproducibility for developers.

5.1. Command-Line Interface (CLI):

As defined in Section 3, the script is executable directly from the command line with clear arguments.

5.2. Output Formats:

- **PNG:** All generated plots will be saved as high-quality PNG images by default.
- **PDF (Optional):** The script could be extended to support --output-format pdf for vector graphics, ideal for reports and publications. This would involve changing plt.savefig() to use a .pdf extension.

5.3. Configuration and Reproducibility:

- **CLI Arguments:** All configurable options (input file, output directory, plot type, filters) are exposed as CLI arguments.
- **Docstrings and Comments:** The Python code will be thoroughly documented with docstrings for functions and inline comments explaining complex logic, enhancing readability and maintainability.
- **Reproducibility:** The script will produce identical plots for the same input data and arguments, ensuring consistency in analysis.

5.4. Example Usage:

Bash

```
# Generate all plots from a CSV file
python render_heatmap.py --input performance_trace.csv --output-dir plots/

# Generate only the mutation cost heatmap, filtering for BLAKE2s hash_type
```

```
python render_heatmap.py --input performance_trace.json \
                         --output-dir plots/ \
                         --plot-type mutation \
                         --hash-filter BLAKE2s

# Generate overall timing heatmap with a stricter anomaly threshold
python render_heatmap.py --input performance_trace.csv \
                         --output-dir plots/ \
                         --plot-type overall \
                         --anomaly-threshold 3.0
```

This specification provides a robust framework for building a valuable visualization tool for EchoPulse performance analysis, crucial for both optimization and side-channel vulnerability assessment in practical deployments.