# LightSys Automated Intrusion Detection System Documentation

**Authors:** Sean Liu, Kenneth Burrill
**Date:** June 2025 - August 2025

**Google Doc Version:** Here

## Table of Contents

1. **Project Overview**
2. **File Architecture and Structure**
   - Project Scripts
   - Project Directories
   - JSON Files
   - Additional Files
3. **Model Details**
4. **Data Flow**
   - Preface
   - Parsing
   - Embeddings
   - Model
   - Results
5. **Model Refinement and Future Work**

## 1. Project Overview

### Context:

Mission organizations often have to provide their staff and field team members quite a number of online services, for example for communication, donor information, and more. However, mission organizations often do not have the staff to closely monitor activity on their servers, leaving those systems open to rather deep and lengthy compromise should a security compromise happen. With millions of security entries that are logged each week, it is simply impossible to manually go through all of them and pick them out by hand. Because of this, there is a need to intelligently and automatically detect potential log anomalies on the LightSys system.

### Summary:

In this project, we aim to provide LightSys system administrators greater ease with detecting potentially anomalous security logs through the utilization of an LSTM Autoencoder model. Our project will allow admins to feed log entries into a model, have the model train on the somewhat "normal" log entries, and then return potential anomalous logs when tested on future log entries.

### Dependencies:

This project requires at least `Python 3.6.0` and the following external Python packages:

- torch (PyTorch)
- numpy

You can install them with: `pip install torch numpy`

## 2. File Architecture and Structure

| Project Scripts | Project Directories | JSON Files | Additional Files |
|---|---|---|---|
| `autodict.py` | `dump` | `action_encoding.json` | `autoencoder.py` |
| `data_processing.py` | `model` | `directory_encoding.json` | `section_log.py` |
| `loss_plateau.py` | `output` | `group_encoding.json` | `temp_output.txt` |
| `model.py` | `storage` | `process_encoding.json` | |
| `parser.py` | | `user_encoding.json` | |
| `tester.py` | | | |
| `trainer.py` | | | |

---

### PROJECT SCRIPTS

**`autodict.py`** This class functions like a python dictionary with extra features. It assigns a unique integer to each new key accessed via indexing. Once a key is assigned a value, it cannot be overwritten. Useful for generating consistent, immutable mappings without manual value management. It can optionally track how often missing keys are accessed. A JSON version of an autodict can be found in the generated JSON files.
**Constructor**

- Parameters:
  - `initial_data` (dict, optional): A dictionary of initial key-value pairs. All values ust be integers.
- Raises:
  - `TypeError`: If `initial_data` is not a dictionary.
  - `ValueError`: If any value in `initial_data` is not an integer.

**Methods**

- Returns value if the key exists
- If missing, assigns it as the next integer and stores it

- Disabled. Raises `TypeError`

- Returns `True` if key exists.

- Iterates over keys.

- Returns key-value pairs.

- Returns a shallow copy of its internal dictionary.

- Increments access count for key.
- If the key hasn't been counted before, it assigns the starting value before incrementing it

- Returns a dictionary containing the number of times certain values have been accessed.

`data_processing.py` This contains several constants and functions used by both the trainer and the tester. It mainly deals with data after it has been parsed by the log parser, but before it is passed into the neural net. These functions implement word embeddings and massage the data into a readable format by the model.
**Constants**

- `CHUNK_SIZE`: Number of log entries per chunk (default: 128)
- `DEVICE`: Computation device (cuda if available, else cpu)
- `PATHS`: Maps field categories to their JSON encoding files
- `FIELDS_TO_DICTS`: Maps log fields to their encoding category **Core Functions**
- Embeds all relevant field*s in a log entry using learned vocabularies.
- Parameters:
  - `entry`: Parsed log entry (dict)
  - `vocab_sizes`: Dict mapping field categories to vocab sizes
- Returns
  - Embedded entry (dict with tensors)

- Returns embedding vector(s) for a field and token ID(s). Creates embedding layer on first use.
- Parameters:
  - `value`: Integer or list of integers
  - `field`: Field name
  - `vocab_size`: Size of vocabulary
- Returns:
  - Tensor of shape `[embedding_dim]`

- Recursively flattens nested structures into a list of 1D tensors.
- Parameters:
  - `obj`: Scalar, list, dict, or tensor
- Returns:
  - List of tensors

- Loads saved embedding weights from disk.
- Parameters:
  - `path`: File path to `.pt` file
  - `vocab_sizes`: Dict of vocab sizes per category

- Saves current embedding layers to disk.
- Parameters:
  - `path`: Output file path

- Computes vocab size per field using max index + 1 from encoding files.
- Returns:
  - Dict of vocab sizes

- Encodes log entries, flattens them, and yields them in batches.
- Parameters:
  - `file_path`: Path to log file (JSON lines)

  - `vocab_sizes`: Dict of vocab sizes
  - `chunk_size`: Number of entries per batch
  - `inference`: If `True`, uses overlapping chunks

- Yields:
  - Tuple: `(Tensor[1, chunk_size, dim], List[int])`

**`loss_plateau.py`** This class monitors training loss over epochs using an exponential moving average (EMA) and determines whether progress has stalled based on a minimum improvement threshold. It supports warmup epochs to delay detection and can be checkpointed for reproducibility in interrupted or staged training workflows. **Constructor**

- Parameters:
  - `min_improvement`: Minimum relative improvement required to continue training. Treated as a percentage (i.e. 0.001 is 0.1 % improvement threshold)
  - `warmup_epochs`: Number of initial epochs to skip plateau detection.
  - `ema_alpha`: EMA smoothing factor (A float between 0 and 1).

**Method**

- Updates internal state with the current loss and checks for plateau.
- Parameters:
  - `current_loss`: Loss value, as averaged across chunks, from the current epoch. **Attributes**
- `epoch`: Current epoch count.
- `should_stop`: Boolean flag indicating whether training should stop.
- `smoothed_loss`: EMA-smoothed loss value.
- `prev_loss`: Previous smoothed loss used for comparison.

**`model.py`** This is the heart of the project. It is an LSTM which has been modified slightly to allow for recurrency across sequences and batches if desired. More details can be found in the corresponding section below. **Constructor**

- Encodes a sequence of vectors into a latent representation.
- Parameters:
  - `input_dim`: Number of features per log entry.
  - `hidden_dim`: Size of LSTM hidden state.
  - `latent_dim`: Size of output latent vector.
- Forward Input:
  - `x`: Tensor of shape `[batch_size, seq_len, input_dim]`
  - `hidden`: Optional initial hidden state
- Forward Output:
  - `latent`: Tensor `[batch_size, latent_dim]`
  - `hidden_out`: Final LSTM hidden state

- Encodes a sequence of vectors into a latent representation.
- Parameters:
  - `output_dim`: Number of features per log entry.

  - `latent_dim`: Size of input latent vector.
  - `hidden_dim`: Size of LSTM hidden state.

- Forward Input:
  - `x`: Tensor of shape `[batch_size, latent_dim]`
  - `seq_len`: Length of output sequence

- Forward Output:
  - `decoded`: Tensor of shape `[batch_size, seq_len, input_dim]`

- Combines encoder and decoder into a full autoencoder.
- Forward Input:
  - `x`: Tensor of shape `[batch_size, seq_len, input_dim]`
  - `hidden`: Optional initial hidden state

- Forward Output:
  - `reconstructed`: Tensor `[batch_size, latent_dim]`
  - `hidden_out`: Final LSTM hidden state

**`parser.py`** This class is a parser built to process chdir logs into numerical data that can be understood by the model. Its output is saved into a text file which can then be processed further to implement embeddings and the like. **Constructor**

- Parameters:
  - `input`: Path to the input log file.
  - `output`: Path to the output file for parsed entries.
  - `year`: Optional override for timestamp parsing. Defaults to current year.
  - `auto_add`: If `True`, new values are added automatically to the autodicts.
  - `default_answer`: Optional default response for the update autodict prompt **Method**

- `parse()`
  - Parses the input log file and writes structured entries to the output file.
  - Returns 0 on success

- `get_num_logs()`
  - Returns the number of log lines successfully parsed.

- `get_has_updated()`
  - Returns `True` if new categorical values were added during parsing.

- `get_has_extras()`
  - Returns `True` if new values were encountered but not added.

- `save_new_data(output)`
  - Writes newly encountered categorical values to a text file.
  - `output`: Path to the output file for saving new values. **Attributes**

- `year`: Year used for timestamp parsing.

**tester.py** This script parses raw log files, encodes them into vector representations, and uses a trained LSTM autoencoder to flag anomalous entries based on reconstruction error. It supports both batch inference and ingesting precomputed anomaly scores from a dump file.

- **Positional Arguments**
  - `<filename>`: Path to the log file to analyze. This file path should be located in the working directory
  - `<model_name>`: Name of the model and its associated embeddings and dump file without the file extension
- **Optional, Mutually Exclusive Filtering Options**
  - (Only one of these optional flags can be used at a time):
  - `-t, --top N`: Select the top N log entries. N must be a positive integer
  - `-p, --pcent P`: Filter log entries by percentile. P must be a valid float between 0 and 1 inclusive. Example, `--pcent 0.05` selects the top 5%
- **Optional, Mutually Exclusive Workflow Modes**
  - Only one of these optional flags can be used at a time:
  - `-d, --dump`: Dumps the intermediary data into a file in the dump directory before it is processed for displaying. This allows the user to reprocess the data with different filtering options using `--ingest` without needing to query the model again with the same data. Requires the `-thresh flag` to remain its default value of 0.0. Will still produce a normal output file with the fully processed data as well
  - `-i, --ingest`: Ingest previously dumped data instead of running the model. Useful for reprocessing or exporting with different filtering options or formats.
- **Optional Flags**
  - `-y, --year YEAR`: Year of the log file to train. Defaults to the current year according to the system if not specified.
  - `-r, --thresh T`: Set an absolute threshold for reconstruction error. Log entries above this value will be flagged and passed into the output. Default: 0.0
  - `-s, --sort`: Enable sorting of flagged entries by reconstruction error in descending order. This effectively shows the most "anomalous" logs first. If this flag is absent, the logs are sorted by line number in ascending order
  - `-c, --csv`: Export the final output to a CSV file for further analysis or the like. If this flag is absent, the data will be saved to a human readable `.txt` file
  - `-u, --update_embeddings BOOL`: If a new value is found, specify whether to update the model's embeddings. Accepts `True` or `False`. If not specified, the script will ask you to decide at the appropriate time if a new value is found.

**trainer.py** This script trains a model on parsed log data to learn normal system behavior. It supports checkpointing, resuming from saved states, and saving trained models and embeddings for downstream anomaly detection. Training can run for a fixed number of epochs or dynamically extend until a loss plateau is detected using an EMA-based detector.

- **Positional Arguments**
  - `filename`: Path to the log file used for training. Example: `error.log` or `latest.txt`
  - `model_name`: Base name (i.e. no file extension) for saving the trained model, embeddings, and checkpoints. Example: `anomaly_model` will produce files like `anomaly_model.pth`, `anomaly_model_embeddings.pth`, and checkpoints like `anomaly_model_5.pt`
- **Optional Flags**

  - `-y, --year YEAR`: Year of the log file to train. Defaults to the current year according to the system if not specified.
  - `-r, --resume EPOCH`: Resume training from a previously saved checkpoint. Example: `--resume 5` will load `checkpoint/anomaly_model_5.pt` and continue training from epoch 5.
  - `-c, --checkpoint INTERVAL`: Save the model every `X` epochs to allow for further training. Frequent checkpointing can significantly impact performance and cost a lot in storage. Example: `--checkpoint 10` saves checkpoints at epochs 10, 20, 30, etc.
  - `-e, --epoch COUNT`: The number of epochs to run during training. If not specified, the program will continue training until the loss values plateau.

---

## PROJECT DIRECTORIES

**`checkpoint`** This directory is optional and used only during model training. It stores intermediate snapshots of the model to support recovery, analysis, or fine-tuning. When training a model, periodic checkpoints can be saved to this directory. These checkpoints allow for resuming training from a specific epoch, inspecting model performance over time, or rolling back to earlier states for experimentation.

- `[model_name]_[epoch].pt`: A PyTorch checkpoint file containing the model's state at a specific training epoch.
  - `model_name`: Identifier for the model used during processing. This name is shared across all related artifacts (model, embeddings, dump files, etc.).
  - `epoch`: The epoch number at which the snapshot was taken

**`dump`** This directory supports optional workflows for efficient reprocessing. It is not required for core functionality but is used when the `--dump` or `--ingest` flags are present when running `tester.py` When the `--dump` flag is used, `tester.py` saves intermediary results—specifically reconstruction errors—into this directory. These dumped files allow the user to reprocess logs without re-querying the model, enabling faster iteration with different filtering or output formats via the `--ingest` flag.

- `[log_file]_[model_name].npy`: A binary NumPy file containing reconstruction errors for each line in the log.
  - `log_file`: Name of the log file without its extension (e.g., error from `error.log`)
  - `model_name`: Identifier for the model used during processing. This name is shared across all related artifacts (model, embeddings, dump files, etc.).

**`model`** This directory stores the trained models and their associated embeddings. It is essential for running inference and reusing trained components. These files are loaded by `tester.py` to compute reconstruction errors and detect anomalies.

- `[model_name].pth`: Serialized PyTorch model file containing the trained weights and architecture.
  - `model_name`: Identifier for the model used during processing. This name is shared across all related artifacts (model, embeddings, dump files, etc.).
- `[model_name]_embeddings.pth`: Serialized file containing learned embeddings for log entries.
  - `model_name`: Identifier for the model used during processing. This name is shared across all related artifacts (model, embeddings, dump files, etc.).

**`output`** This folder contains the results generated by `tester.py`, including flagged anomalies and new data discovered during model querying.

- `[model_name]_new_data.txt`: Tracks newly encountered log data like directory or process names and how often they appeared in the testing logs.
  - `model_name`: Identifier for the model used during processing. This name is shared across all related artifacts (model, embeddings, dump files, etc.).
- `[log_file]_[model_name]_flagged.txt`: Human-readable list of flagged entries with their corresponding line number and error score.
  - `log_file`: Name of the log file without its extension (e.g., error from error.log)
  - `model_name`: Identifier for the model used during processing. This name is shared across all related artifacts (model, embeddings, dump files, etc.).
- `[log_file]_[model_name]_flagged.csv`: CSV-formatted flagged entries with their corresponding line number and error score along with appropriate headers for structured analysis.
  - `log_file`: Name of the log file without its extension (e.g., error from error.log)
  - `model_name`: Identifier for the model used during processing. This name is shared across all related artifacts (model, embeddings, dump files, etc.).

**`storage`** This directory contains the generated json files used to save the autodict data structure between training and testing. For a model to function correctly, these files must not change after training. See the next section below for a breakdown of each file.

---

## JSON FILES

**`action_encoding.json`** This file indexes every possible action found in the logs. At present, this project can only process ***chdir*** logs. The following log entry field contributes keys to this file:

- action

**`directory_encoding.json`** This file indexes every unique directory name found in the logs. This includes ***target*** directories, ***path*** directories, etc. The following log entry fields contribute keys to this file

- target directory
- process path
- parent process path

**`group_encoding.json`** This file indexes every unique ***gid*** and ***egid*** found in the logs. The following log entry fields contribute keys to this file:

- process gid
- process egid
- parent process gid
- parent process egid

`process_encoding.json` This file indexes every unique ***process*** name found in the logs. The following log entry fields contributes keys to this file:

- process command
- parent process command

`user_encoding.json` This file indexes every unique ***uid***, ***euid***, and ***ouid*** found in the logs. The following log entry fields contribute keys to this file:

- process uid
- process euid
- process ouid
- parent process uid
- parent process euid
- parent process ouid

---

**ADDITIONAL FILES**

`autoencoder.py` This contains the first version of the model. It is no longer used by the other scripts, but it could be reimplemented with some tweaking. See model selection rationale below

`section_log.py` This script is used to split logs up into multiple sections for easier training and testing. Used for developing the model and not destined for the end product.

`temp_output.txt` This is a temporary output file that is generated by the log parser during training or testing to hold the numerical version of each log before word embeddings are implemented. It will stick around if the program crashes or exits unsuccessfully. Safe to delete if the script fails.

## 3. Model Details

### Model Type

This project utilizes an **LSTM (Long Short-Term Memory) Autoencoder** built with PyTorch. LSTM is a specialized type of **Recurrent Neural Network (RNN)** designed to capture long-range dependencies in sequential data, making it particularly **effective for modeling log sequences** to learn compact representations of normal system behavior.

### Model Selection Rationale

Log data is sequential by nature—its meaning often depends on the surrounding context. Anomalies typically arise from unexpected patterns or timing, rather than isolated events.

We chose an LSTM-based autoencoder, a variant of RNNs, because of its **ability to retain and model temporal dependencies over extended sequences**. The encoder compresses a sequence of log entries into a latent representation, and the decoder reconstructs the original sequence from that representation. This allows the model to learn the structure and dynamics of normal log behavior.

At inference time, the model attempts to reconstruct incoming log sequences. The reconstruction error is then compared to a threshold: **sequences with high error are flagged as anomalous, indicating a deviation from learned patterns**.

**Model Structure**

**Encoder** The encoder is composed of stacked LSTM layers that read each log entry in the sequence one at a time. As each entry is processed, the LSTM updates its internal hidden state—a kind of memory that accumulates information about everything it has seen so far. This hidden state captures temporal dependencies, meaning it learns how earlier events influence later ones. The encoder's job is to summarize the entire sequence into a single vector that reflects its overall behavior.

**Latent Space** This summary vector is known as the latent representation, and it lives in what's called the latent space. The latent space is a lower-dimensional space where the model compresses the input sequence. Instead of remembering every detail, the model learns to retain only the most relevant patterns—like which users tend to access which directories, or which processes are typically spawned together. This compression forces the model to generalize. If the decoder can reconstruct the original sequence from this compressed vector with low error, it means the input was familiar. But if the reconstruction error is high, the input likely deviated from known patterns and may be anomalous.

**Decoder** The decoder is another set of LSTM layers that take the latent vector and attempt to reconstruct the original sequence. To do this, the latent vector is repeated across the sequence length and fed into the decoder LSTM. The decoder generates a predicted sequence of log entries, which is then compared to the actual input using Mean Squared Error (MSE). This difference is called the reconstruction error.

**Thresholding** Once the reconstruction error is computed, the model applies thresholding. A user-defined threshold determines whether a sequence is flagged as anomalous. If the error is below the threshold, the sequence is considered normal. If it exceeds the threshold, it's flagged. Adjusting this threshold allows users to control the model's sensitivity: a lower threshold catches more anomalies (but may include false positives), while a higher threshold is stricter (but may miss subtle anomalies).

**Recurrency** All of this is powered by recurrency—the ability of LSTM layers to maintain and update a hidden state over time. This is crucial for system logs, which often contain long-range dependencies. For example, a configuration change might affect behavior hours later. The LSTM's hidden state allows the model to "remember" these earlier events and factor them into its understanding of the current sequence. This makes the autoencoder well-suited for modeling the complex, temporal nature of system logs.

## 4. Data Flow

I. Preface |

II. Parsing | III. Embeddings | IV. Model | V. Results |

### I. Preface

This section outlines the flow of data through the system, covering preprocessing, training, testing, and analysis. It also provides background on the log data used to train the model. For details on specific files and modules, refer to the File Structure section—this section focuses on how the components interact.

### II. Parsing

The security log files used in this system are generated through GRSecurity, a suite of kernel-level security patches and access control enhancements for Linux systems. GRSecurity extends the default Linux kernel with advanced security auditing, role-based access control (RBAC), and process monitoring features that help detect and prevent unauthorized activity.

At LightSys, GRSecurity is configured to run automatically via cron jobs, which executes scheduled auditing and monitoring tasks at regular intervals (e.g., hourly or daily). These jobs trigger system scans and produce a high-volume stream of security-related log entries—often millions per week—capturing low-level system activity.

**Example Log:**

```
Jun 18 23:13:38 vs-serv-rl8 kernel: grsec: chdir to / by /usr/lib/systemd/systemd[systemd:1] uid/euid/ou
```

Although raw log formats can vary, this program specifically targets `chdir` entries generated by GRSecurity and parses each entry into the following fields. The corresponding section of the log is surrounded by parenthesis.

- `Time (Jun 18 23:13:38)`: The date and time of the log entry. The year is either user-specified or defaults to the current year. The time is saved in four fields using a cyclical format of `sin` and `cos` for both the time and the day. (23:00 and 0:00 are only 1 hour apart, but numerically they're 23 and 0 — far apart!) This lets the model recognize that midnight follows 11 PM, not that it "resets."
- `Hostname (vs-serv-rl8)`: The hostname of the machine that generated the log as expressed in one-hot encoding. Currently hardcoded.
- `Container (absent)`: Another one-hot encoding field, this time for the eleven different containers in the LightSys system. There are also two extra fields in this encoding—one for no container present, and the other as a fallback in case a container appears that isn't one of the eleven.
- `Action (chdir)`: This field records which action was performed. In the case of our program, it is always `chdir`.
- `Target Directory (/)`: This records the directory to be changed to in the `chdir` command.
- `Process Path (/usr/lib/systemd/systemd)`: Similar to target directory, only instead the path to the process executing the `chdir`.
- `Process Command (systemd)`: The process command, i.e. the actual executable running `chdir`.
- `Process User (uid/euid/ouid:0/0/0)`: Actually stored as six separate fields, it records the `uid`, `euid`, and `ouid` as well as some extra info like if it is root or a system user.
- `Process Group (gid/egid:0/0)`: Similar to the process user, it is stored as four separate fields; it records the `gid` and `egid` as well as some extra info like if it is root or a system group.
- `Parent Process Path (/)`: Same as for the process, except that it contains the info for the parent of the process.
- `Parent Process Command (swapper)`: Same as for the process, except that it contains the info for the parent of the process.
- `Parent Process User (uid/euid/ouid:0/0/0)`: Same as for the process, except that it contains the info for the parent of the process.
- `Parent Process Group (gid/egid:0/0)`: Same as for the process, except that it contains the info for the parent of the process.

There are a few fields that are not processed since they appear in every log:

- `Source (kernel)`: Indicates that the message is from the kernel.
- `System (grsec)`: Log entry is produced by GRSecurity's kernel auditing.

When the logs are finished being processed by the parser, they are saved to a temporary output file with each line being one JSON log entry in the following format:

```json
{
  "line_number": 0,
  "time": {
    "sin_day": 0.7818314824680298,
    "cos_day": 0.6234898018587336,
    "sin_time": -0.4998740364227649,
    "cos_time": 0.8660981166764031
  },
  "hostname": [1, 0],
  "container": [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
  "action": [1],
  "target_directory": [1],
  "process": {
    "path": [1],
    "command": [6],
    "uid": 1,
    "euid": 1,
    "ouid": 1,
    "uid_flags": [1, 1],
    "euid_flags": [1, 1],
    "ouid_flags": [1, 1],
    "gid": 1,
    "egid": 1,
    "gid_flags": [1, 1],
    "egid_flags": [0, 0]
  },
  "parent": {
    "path": [2, 17, 18],
    "command": [5],
    "uid": 1,
    "euid": 1,
    "ouid": 1,
    "uid_flags": [1, 1],
    "euid_flags": [1, 1],
    "ouid_flags": [1, 1],
    "gid": 1,
    "egid": 1,
    "gid_flags": [1, 1],
    "egid_flags": [0, 0]
  }
}
```

At this point, the data is only half processed. The important fields have been picked out, but the bolded sections—namely `action`, `target_directory`, and `path`, `command`, `uid`, `euid`, `ouid`, `gid`, and `guid` for both `process` and `parent`—will be fully implemented with embeddings.

**NOTE**: Why not one-hot or normalization for all fields?
One-hot encodings treat all categories as completely unrelated—there's no notion of similarity between them. Embeddings are trainable: the model learns to place similar log elements closer together in vector space.

### III. Embeddings

As shown above, those sixteen fields are only partially processed. They still need to be entered into embedding fields.

Embedding fields require a vocabulary size; consequently, we can't embed these fields before knowing the total number of vocabulary words.
Thus, we keep track of all words in five data structures known as *autodicts*. All fields of a similar type are grouped together.

The autodicts are essentially dictionaries with some special properties. They keep a running index for each key added to them.
This is all then saved to a JSON file between runs. These files are stored in a directory named `storage/` and in the following five files:

- `action_encoding.json`
- `directory_encoding.json`
- `group_encoding.json`
- `process_encoding.json`
- `user_encoding.json`

The integers listed in each of the bolded fields above are the unique IDs/index values for a specific vocab word.
Take the parent's path for example: `[2, 17, 18]`.
Looking in the corresponding JSON file that stores directory names, we see the following:

```json
{
  "": 1,
  "usr": 2,
  "lib": 3,
  "systemd": 4,
  "var": 5,
  "www": 6,
  "html": 7,
  "mib": 8,
  "sbin": 9,
  "php-fpm": 10,
  "root": 11,
  "crond": 12,
  "docker": 13,
  "overlay2": 14,
  "Ec2f3eda493344ef... [shortened for clarity] ...3d20bb8b8c6215c": 15,
  "merged": 16,
  "bin": 17,
  "runc": 18,
  "etc": 19,
  ...
}
```

If we match the keys to their index values, we see that the parent's path is **`usr/bin/runc`**! These five autodicts, ACTION, DIR, GROUP, PROC, and USER maintain a mapping of each raw value to its corresponding, unique ID (assigned by the order in which it was found).

Once the embeddings are formed, it all gets flattened, i.e. made into a 1D tensor, and passed into the model.

## IV. Model

Once the embeddings are formed, it all gets flattened, i.e. made into a 1D tensor, and passed into the model. The model processes these tensors in chunks, which are fixed-length sequences of log entries. Each chunk is a 2D tensor of shape `[chunk_size, input_dim]`, where chunk_size (default: 128) defines how many log entries are grouped together to form a single input sequence. These chunks are then stacked into a 3D tensor of shape `[1, chunk_size, input_dim]` — this outer dimension represents the batch size, which is always 1 during training.

It's important to distinguish between chunk, batch, and sequence in this context:

- A **chunk** is a single sequence of log entries used as input to the model.
- A **batch** is a collection of chunks processed together. In this implementation, the batch size is 1, so in this case, each chunk is its own batch.
- A **sequence** refers to the temporal ordering of log entries within a chunk. The LSTM processes this sequence step-by-step, maintaining internal memory across time steps.

The model is an LSTM autoencoder that compresses each input sequence into a latent vector and reconstructs it to detect anomalies. The encoder reads the entire chunk and summarizes it into a hidden state, which is projected into a lower-dimensional latent representation. The decoder then reconstructs the original sequence from this vector by repeating it across the sequence length and generating predicted log entries.

During inference, an optional sliding window mechanism is used. Instead of processing non-overlapping chunks, the loader retains the second half of each chunk and uses it as the beginning of the next one. This overlapping strategy ensures that anomalies occurring near chunk boundaries are not missed and that context is preserved across adjacent sequences. The stride used, in this current implementation, for this sliding window is half the chunk size, meaning each new chunk overlaps 50% with the previous one.

This chunked, sequential processing allows the model to learn nuanced patterns in system behavior and detect deviations based on reconstruction error.

### V. Results

For each log entry, the reconstruction error is computed using Mean Squared Error (MSE) between the original and reconstructed vectors. During training, these errors are averaged across the chunk to produce a single loss value, which guides model updates via the optimizer.

In inference, reconstruction errors are retained per entry for fine-grained anomaly detection. Each error is compared against a user-defined threshold; entries exceeding it are flagged as anomalous. This threshold acts as a filter, passing only high-error entries to the final results.

Flagged entries are sorted (if requested), deduplicated by line number, and written to an output file with their error scores. Output can be formatted as plain text or CSV. Optionally, raw anomaly scores are dumped to disk for reuse, allowing reanalysis without rerunning inference.

## 5. Model Refinement and Future Work

After working on the project up to this point, we have a few ideas for improvements, refinements, and potential implementations.

### Input Validation and Parsing Robustness

Currently, log entries lack input validation and sanitization. Some valid entries fail to parse due to conflicts with reserved keywords or unexpected syntax. For instance, a directory name containing the word "by" (with surrounding spaces) can break parsing if it's in the wrong location (such as at the end of a directory name). These edge cases should be addressed before deployment.

### Word Embedding Expansion

Some fields (e.g. hostname) still use one-hot or other encoding methods. Expanding the use of word embeddings could improve model expressiveness. If you do this, remember to create a new storage JSON file for each new field type. You'll find the relevant logic at the top of `data_processing.py`.

### Validation Step

Our model currently lacks a validation phase during training. Adding one would help monitor generalization and detect overfitting. That said, in this context, overfitting may be preferable—a more sensitive model is better at flagging anomalies.

### Model Swapping

To swap the LSTM model out for the autoencoder or some other type of model:

- Update `tester.py` and `trainer.py` to instantiate and initialize the new model. If appropriate, ensure it outputs values in the range `[-1, 1]` to support cyclical time encoding.
- Adapt the training loop to handle hidden states or recurrence as needed.
- Modify `load_vectors` to return tensors in the correct shape (e.g. autoencoders expect `[chunk_size, input_dim]` vs. the LSTM's `[1, chunk_size, input_dim]`).
- Ensure reconstruction loss is computed correctly for the new model.

### Fine Tuning

The model has only undergone minimal tuning. Most hyperparameters remain at their default values, and we've done little experimentation to assess their impact. The only parameter we adjusted was the embedding dimension in `data_processing.py`. While we briefly attempted to scale it dynamically based on vocabulary size, this led to significantly worse results—so we reverted to a fixed value.

That said, there are many dials worth exploring: embedding dimensions, sliding window stride, chunk size, batch count, sequence length, hidden layer size, optimizer settings, and more. Each of these can influence model performance, and the number of possible combinations is vast. Future work should include systematic tuning and evaluation across these parameters to optimize results.

### Increase Scope

Right now, the project only looks at `chdir` logs from a single host. This has been useful for testing, but there are several ways to expand the system.

**Multi-Host Support**:
The model could be extended to handle logs from multiple machines. To do this:

- Train the model to spot patterns that span across hosts, like coordinated activity or unusual access sequences.
- Depending on how data is stored, you could either combine logs centrally or use a federated setup where each host trains locally.

**Additional Log Types**:
We're only using `chdir` logs for now, but other types could add valuable context:

- File access, process creation, login attempts, and network activity are all good candidates.
- Each log type would need its own parser and embedding logic, but the structure can be reused.
- If you go this route, consider using models that can handle mixed inputs.

## Implementation Suggestions

For deployment, we recommend the following practices:

- **Automated Inference**: Use cron jobs or a lightweight scheduler to run inference at regular intervals (e.g., every minute). This keeps the system responsive to new data and minimizes latency.
- **Periodic Retraining**: The model should be retrained periodically to capture evolving patterns, especially as system behaviors shift over time. Depending on data drift, retraining could be scheduled weekly, monthly, or triggered by performance degradation.
- **Versioning and Rollback**: Maintain versioned checkpoints of trained models to allow rollback in case a new version underperforms.

## General Improvements

There are many other ideas that could improve the model in future versions. Most of these haven't been tested yet, but they're worth exploring.

- **Better Handling of Recurrence**: Right now, the model uses LSTMs to capture patterns over time. But there might be better ways to handle recurring behavior—like using attention mechanisms or memory-based models that can track long-term dependencies more effectively.
- **Use of Frequency Space**: Instead of just looking at logs as sequences, you could analyze them in the frequency domain. This means converting time-based patterns into wave-like signals using tools like Fourier transforms. It might help detect periodic or cyclical behavior that's hard to spot in raw sequences.
- **Markov Models**: These were attempted before with limited results, but it was from that project that we built our own. However, they might be worth revisiting in a different capacity. Markov models are good at modeling systems where the next state depends only on the current one. They're simple but powerful for certain types of log data.

All of these ideas would require changes to the model architecture and training loop, but they could lead to better performance or more interpretable results.

---