SELINUX MALICIOUS LOG DETECTION AND POLICY ADJUSTMENT IN ANDROID

ABSTRACT

Security-Enhanced Linux (SELinux) in android provides mandatory access control in Android to confine apps and services and mitigate exploits . However, SELinux generates voluminous audit logs (AVC logs) for every denied operation, which are difficult to analyze manually. This thesis presents a deep learning framework to automatically distinguish benign from malicious SELinux log sequences and assist in dynamic policy adjustment. We collect Android SELinux denial logs using ADB and the Monkey tool to simulate benign and malicious app behaviors. The logs are preprocessed by parsing key fields (security contexts, process info, etc.), encoding categorical features, and applying Principal Component Analysis (PCA) for dimensionality reduction. We then segment the logs into fixed-length sequences and train sequence models – Bidirectional LSTM, Bidirectional GRU, and an attention-enhanced BiLSTM – to classify each sequence as benign or malicious. The attention mechanism highlights critical log events that contribute to a malicious classification, providing interpretability. Experimental results show high classification accuracy (around 89% validation accuracy) for all three models, with the BiLSTM+Attention model performing slightly best. More importantly, the attention weights help identify policy weaknesses or misconfigurations by pinpointing which denied actions are associated with attacks. Using these insights, administrators can adjust SELinux policies dynamically . This approach thus reduces manual analysis effort and improves Android's security posture. We conclude with key takeaways, including the viability of deep learning for security log analysis, and outline future work on scaling the dataset, exploring transformer-based models, real-time log monitoring, and automated SELinux policy refinement.

INTRODUCTION

Android relies on SELinux to enforce mandatory access control (MAC) over all processes, including those with root privileges. SELinux operates under a **default-deny policy**, meaning any action not explicitly allowed by policy is denied and logged. In Android's enforcing mode, disallowed actions are prevented and every attempt is recorded in the system log (via dmesg/logcat) as an "avc: denied" message. These SELinux logs are crucial for debugging and security auditing, as they reveal operations blocked by the policy. SELinux significantly strengthens Android's security by confining apps to sandboxes and limiting their access to system resources . The integration of SELinux into Android (often termed **SEAndroid**) has been shown to mitigate numerous exploits and vulnerabilities in the platform. Policies in SELinux define which process domains can access which resource types, thereby preventing even privileged processes from actions outside their allowed scope.

While SELinux improves security, it introduces new challenges in management and log analysis. Modern Android devices produce a high volume of SELinux **Access Vector Cache (AVC)** log entries – each denied access attempt generates an audit log. Administrators and developers must sift through these logs to differentiate benign policy denials (which may just indicate non-harmful attempts or missing allow rules for legitimate operations) from truly malicious attempts. This is a difficult task: SELinux policies are complex, with thousands of rules and interactions, and the corresponding logs can be overwhelming to interpret. Prior work notes that SELinux rulesets are "massive and semantically complex," making it challenging to analyze or modify policies manually. In fact, a typical Android SELinux policy can contain many rules, and log outputs can grow quickly during system runtime.

Misconfigurations or blind spots in SELinux policy can have serious security consequences. Improperly configured rules might inadvertently allow dangerous actions or, conversely, overly restrictive policies might disrupt normal app functionality. The difficulty of manual log analysis means malicious behavior could go undetected in the noise of benign denials, or

administrators might use tools to blindly permit operations that were denied, not realizing some denials were triggered by attacks. There is a clear need for intelligent automation to assist in interpreting SELinux logs and maintaining the policy. Wang *et al*. (2024) highlight this need in their work on SELinux Policy Recovery: they propose an architecture (SPRT) that automatically adjusts SELinux policies to mitigate vulnerabilities, by classifying vulnerability descriptions and using audit logs to guide policy changes. Their results (achieving about 92.8% accuracy in classifying known vulnerabilities) show the promise of machine learning in aiding SEAndroid policy management.

Problem Statement: In this context, our work addresses the problem of distinguishing malicious activity from benign behavior using SEAndroid audit logs on Android. The central challenge is that **AVC logs are high-volume and complex**, making manual analysis inefficient and error-prone. Security analysts struggle to determine whether a sequence of denied operations represents a legitimate app hitting a harmless permission denial, or an attacker probing the system. Without automated help, one might either overlook a stealthy attack or misclassify normal behavior as malicious, leading to misconfigurations. This thesis proposes to leverage deep learning to automatically detect malicious log sequences and to use the model's insights to inform SELinux policy adjustments in Android.

Our **proposed solution** is a deep learning framework for malicious log sequence detection and policy enhancement. We employ recurrent neural networks to model sequences of log events, combined with dimensionality reduction and an attention mechanism for interpretability. In summary, the contributions of this work include: (1) a methodology to collect and label real SELinux AVC logs from Android in both benign and malicious scenarios; (2) a data preprocessing pipeline that extracts relevant features from raw logs and applies PCA to handle high-dimensional categorical data; (3) design and implementation of sequence classification models (BiLSTM, GRU, and BiLSTM with an attention layer) to learn patterns of malicious behavior; and (4) a demonstration of how the learned attention weights can pinpoint policy weaknesses or misconfigurations, enabling dynamic SELinux policy adjustment. By integrating log analysis with policy tuning, the

system can both detect attacks and help prevent future ones (e.g., by suggesting policy changes to close attack paths or to reduce false alarms). The remainder of this thesis is organized as follows: **Section II: Background** covers SELinux policy enforcement in Android and related work. **Section III: Problem Statement** reiterates the challenges in AVC log analysis. **Section IV: Methodology** gives an overview of our approach. **Section V: Data and Preprocessing** details data collection, labeling, and feature engineering. **Section VI: Deep Learning Models** describes the BiLSTM, GRU, and attention mechanisms. **Section VII: Experimental Results** presents model performance and analysis. **Section VIII: SELinux Policy Adjustment** discusses how attention results guide policy changes. Finally, **Section IX: Conclusion and Future Work** summarizes key findings and outlines directions for further research.

BACKGROUND: SELINUX AND SEANDROID

SELinux is a Linux kernel security module that implements mandatory access control. Unlike traditional discretionary access control (DAC) where resource owners decide permissions, SELinux's MAC model consults a central policy for every access decision . In SELinux, every process and object is assigned a **security context** (a tuple of *user, role, type, level*). Android leverages the *Type Enforcement* (TE) aspect of SELinux policy : each process (subject) runs in a domain (type) and each resource (file, socket, etc.) has a type label; policy rules then specify which types can access which others. An SELinux policy rule follows the form `allow <source_type> <target_type>:<class> <permissions>;` . For example, a rule may allow the type `untrusted_app` (applications) to read and write objects of type `app_data_file` (their own data files). Anything not explicitly allowed by a rule is denied by default. This restrictive default-deny stance is key to limiting the damage that compromised or malicious apps can cause on Android.

SELinux in Android (SEAndroid): Android introduced SELinux in a permissive mode in version 4.3 and moved to enforcing mode by Android 5.0, applying SELinux across all apps and system processes. This was a major security enhancement for Android, as documented

by Smalley and Craig in their work *Security Enhanced (SE) Android* . By confining even root processes to least privilege domains and isolating apps from each other, SELinux mitigates the impact of malware and vulnerabilities on Android. For instance, even if a malicious app gains native code execution, SELinux can prevent it from reading or modifying other apps' data, accessing system resources, or escalating privileges in unauthorized ways . Over successive Android releases, SELinux policies have been tightened to cover dozens of domains (e.g., separate domains for media server, network daemon, etc.), reducing the attack surface. The Android SELinux policy is built into the system images (with split policies for system and vendor in modern Android) and is carefully curated to balance security and functionality.

SELinux Policy Enforcement and Logging: SELinux enforces MAC at the kernel level via the Linux Security Modules (LSM) hooks. When a process attempts an operation (open a file, access a socket, etc.), the kernel's SELinux module checks the loaded policy: it looks at the subject's domain type, the target object's type and class, and the requested permission. If a matching "allow" rule exists, the action is permitted; if not, the kernel denies the action. In enforcing mode, these denials are real – the kernel returns an error (usually EPERM) to the process and prevents the operation. *Importantly, SELinux logs every denial*. In Android, these AVC (Access Vector Cache) messages are emitted to the device logs (kernel dmesg and logcat) . A typical log entry might be:

```
avc: denied { read } for pid=1234 comm="app_process" name="secret.txt"
dev="mmcblk0p1"                    ino=12345                    \
scontext=u:r:untrusted_app:s0:c135,c256
tcontext=u:object_r:system_data_file:s0                  tclass=file
```

This indicates a process in the `untrusted_app` domain tried to read a file labeled `system_data_file`, which was not allowed by policy. These **AVC logs** are invaluable for understanding SELinux decisions. They allow policy developers to see what is being blocked. Administrators can use tools to iteratively refine the policy during development: run the

system in permissive mode (which logs denials but doesn't enforce them), collect AVC logs for all operations, then generate and apply allow rules for the needed legitimate operations, and finally enforce the policy. Android's security model encourages this iterative approach – for example, when adding a new service or feature, engineers run tests, gather SELinux denials, and then adjust the policy so that only the necessary permissions are granted, nothing more.

However, not all denials should be resolved by adding allow rules – some represent real attacks or undesirable behavior that should remain denied. This is where simply using audit2allow can be dangerous: it might suggest permitting an action that was rightly blocked as a security measure. Thus, distinguishing benign from malicious denials is critical. Prior research has looked at automatically analyzing policies and even adjusting them. The SPRT framework by Wang *et al.* is one such approach focusing on policy misconfigurations . It uses natural language processing on vulnerability descriptions and maps them to SELinux policy rules, modifying the policy to mitigate those vulnerabilities . Their system also leverages audit logs to verify and apply the fixes. This indicates a trend towards intelligent tools for SELinux policy management. Our work aligns with this trend but tackles the problem from a log sequence analysis perspective: instead of known CVE descriptions, we directly learn patterns from the SELinux audit logs themselves to detect malicious behavior and inform policy changes.

In summary, SELinux (and SEAndroid) provides a strong security foundation in Android through fine-grained access control and logging of policy violations. The challenge we face is utilizing those logs to improve security further. The next section formally states the problem we aim to solve.

PROBLEM STATEMENT

Android's SELinux audit (AVC) logs contain sequences of denied operations resulting from the enforcement of policy. Within these sequences could be the tell-tale signs of an attack – for example, a malicious app might try a series of prohibited actions (accessing contacts database, changing system settings, etc.) and each attempt is denied and logged. Conversely, benign apps can also trigger SELinux denials during normal use (e.g., an app might try to write a file in a location it doesn't have access to, which is harmless and common). The problem is that manually distinguishing malicious patterns from benign ones in these logs is exceedingly difficult. The volume of logs on a typical device is large, and individual log messages are low-level and cryptic (with SELinux context strings and codes). Security analysts may not catch an ongoing attack if its log traces are buried among hundreds of routine denials. Conversely, if we assume anything that shows up in the logs is an issue and blindly allow or alarm, we either weaken security or overwhelm with false positives.

Concrete issues include:

- **High Volume:** Automated testing or heavy usage can produce thousands of AVC entries. Analysts cannot feasibly review each in real time, leading to potential oversight.
- **Complex Sequences:** An attack might only be recognizable when looking at a sequence of events in context, not an individual log line. For example, a single denied file read by itself might seem innocuous, but a combination of multiple denied actions across domains in a short span could indicate a malware systematically probing the system.
- **Benign vs Malicious Denials:** Many denials are benign (policy working as intended to confine apps). Others may indicate misconfigurations (legitimate activity

erroneously denied due to overly strict policy), and others truly malicious attempts. The log lines themselves do not label intent, and the same denial (e.g., access to a system file) could be either a careless app or a deliberate exploit attempt.

- **Policy Maintenance:** Because it's hard to interpret logs, there is a risk of misadjusting the policy. For instance, an admin might add an allow rule to "fix" an issue observed in logs without realizing that the denial was actually protecting the system from an exploit. Over time, this could introduce vulnerabilities.

Given these challenges, the problem statement can be summarized as: *How can we automatically detect malicious behavior in Android's SELinux denial logs (AVC logs) and use that insight to guide SELinux policy adjustments, thereby reducing manual analysis effort and preventing both security breaches and misconfigurations?*

The key difficulties are ensuring high **detection accuracy** (so that we correctly identify truly malicious sequences and do not flag benign behavior as malicious too often) and providing **interpretability** (so that any suggested policy adjustments are understandable and justified by specific evidence in the logs). We address this by developing a deep learning-based solution that treats AVC log lines as a time-series and leverages sequence modeling to learn patterns characteristic of attacks. Additionally, we incorporate an attention mechanism to highlight which log events in a sequence were most indicative of malicious activity, making the model's decisions more transparent and actionable for policy tuning.

In the next sections, we detail our methodology for tackling this problem, including data collection from an Android environment, preprocessing and feature extraction from SELinux logs, the design of deep learning models for sequence classification, and how we close the loop by feeding the model's output back into SELinux policy improvement.

METHODOLOGY

To solve the stated problem, we propose an end-to-end methodology that involves **log data collection**, **preprocessing**, **sequence modeling with deep neural networks**, and

feedback for policy adjustment. **Figure 1** illustrates the overall system pipeline (from data to decision), and we outline each component below:

- **Data Collection & Labeling:** We gather SELinux AVC logs from an Android environment under two conditions: benign normal operation and simulated malicious behavior. We utilize Android's debugging tools to extract logs (via ADB) and use the **Monkey** testing tool to generate a wide range of app activities. This provides a corpus of real SELinux denial messages. Logs from scenarios without any malicious activity are labeled as *benign*, while logs generated during the execution of known malicious actions or malware are labeled as *malicious*. The collection process is described in Section V.
- **Preprocessing & Feature Engineering:** Raw log lines (which are unstructured text) are parsed to extract relevant fields such as timestamps, process IDs, source context, target context, object class, permission, etc. We focus on fields that characterize the *who*, *what*, and *result* of each access attempt. Categorical features (like context strings) are encoded into numerical form. We handle any missing or irrelevant information (e.g., dummy PIDs or empty app names) by cleaning or imputation. Given the high dimensionality of textual features (security context labels can be numerous), we apply **Principal Component Analysis (PCA)** to reduce the feature space while preserving variance. The output of this stage is a sequence of numeric feature vectors representing the log events. Details are in Section V.
- **Sequence Generation:** Instead of examining individual log events in isolation, we construct fixed-length **time window sequences** of events. Using a sliding window approach, a sequence of length *N* (e.g., N=10 events) is taken as one sample, moving the window by one event each time to generate many overlapping sequences. Each sequence inherits a label (benign or malicious) based on the scenario (or specifically, we use the label of the last event in the sequence as an approximation for the sequence label, under the assumption that malicious events cluster together). This framing allows the model to learn temporal patterns. Section V explains the sequence labeling strategy.

- **Deep Learning Model Training:** We design three recurrent neural network (RNN) based models to learn from the sequences: (1) a Bidirectional LSTM network, (2) a Bidirectional GRU network, and (3) a Bidirectional LSTM with an Attention layer. These models process an input sequence of event vectors and output a binary classification (malicious or benign). We train these models on a portion of the labeled sequences and validate on a hold-out set to tune hyperparameters. The model architectures and training settings are detailed in Section VI. We use standard training techniques (mini-batch gradient descent with Adam optimizer, binary cross-entropy loss, etc.) and apply measures to prevent overfitting (like dropout and early stopping).
- **Evaluation and Comparison:** We evaluate the models on accuracy and other metrics to see how well they distinguish malicious sequences. A comparative analysis is conducted to determine which model performs best and what the trade-offs are (e.g., complexity, training time). This is presented in Section VII. We also examine the models' outputs on example sequences to verify that true attack patterns are detected and that benign sequences are largely ignored (low false positive rate).
- **Attention-Based Log Analysis:** For the model that includes an attention mechanism, we extract the attention weights for sequence events to interpret which log entries the model focused on when making a decision. By doing so, we translate the deep learning output into something actionable: specific SELinux denials that were considered most suspicious. Section VIII discusses how these highlighted events can be mapped back to SELinux policy rules.
- **Policy Adjustment Feedback:** Finally, we demonstrate how the insights from the model, particularly from the attention analysis, can assist an administrator in adjusting the SELinux policy. For example, if the attention consistently flags a particular denied operation in malicious sequences, this might indicate a policy rule that is being targeted by attackers – the admin might want to double-check that rule or related permissions. If the model flags an operation as malicious but the admin

knows it's a false positive (benign), that might indicate the policy is too strict for that case and could be relaxed. Thus, the system can either suggest **adding an allow rule** (if a benign behavior was misclassified) or **keeping/adding a deny rule** and possibly monitoring (if a malicious behavior was correctly identified). We stop short of fully automating policy changes (as that requires careful human validation), but we outline how such a system could be integrated into a security workflow. Section VIII provides examples of dynamic policy adjustment guided by our model.

In summary, our methodology marries data-driven sequence classification with security policy management. By learning from real logs, the system builds a model of "normal" vs "attack" behavior. By using attention and sequence context, it ensures the results are interpretable in terms of specific SELinux policy elements. This comprehensive approach aims to reduce the burden on human experts and make Android systems more adaptive in the face of evolving threats.

DATA AND PREPROCESSING

DATA COLLECTION METHODOLOGY

To obtain a dataset of SELinux AVC log sequences, we set up an Android testbed and performed controlled experiments for both benign and malicious scenarios. All logs were collected from a Google Pixel device emulator running Android 11 with SELinux in enforcing mode. We used **Android Debug Bridge (ADB)** to start the log collection and execute test inputs.

For benign data, we relied on the **UI/Application Exerciser Monkey** (commonly known as "monkey") – a command-line tool that generates random user events (clicks, swipes, input) across apps. We launched the monkey tool to simulate an average user's random interaction with various apps over an extended period (several thousand events). This approach induces many SELinux denials that occur during normal usage (for instance, apps

trying to access some system resources that are off-limits). Because no actual malware is present in this phase, these logs form our baseline benign patterns.

For malicious data, we introduced a simulated malicious app and scenarios. We developed a test Android application with behaviors mimicking malware: it attempted actions such as reading other apps' data directories, accessing protected system settings, and performing operations outside its sandbox. These actions are expected to be denied by SELinux. We ran this app on the emulator, also using monkey to drive it and other apps concurrently, to generate a mix of traffic but with known malicious attempts occurring. In addition, we scripted some specific sequences of calls (via `adb shell` commands) that are known to violate policy – for example, trying to start services without permission, or altering SELinux contexts. All these attempts produce AVC log entries that we label as malicious.

Throughout these experiments, **log collection** was done via `adb logcat` and `dmesg`. We filtered the output for lines containing "avc: " to extract just the SELinux-related log messages. We used simple grep filters (e.g., `adb logcat -d | grep avc`) as suggested in documentation (Presentation 6new.pdf). Each log line was time-stamped (by the log system) and contained various fields as shown earlier. We saved the logs to files corresponding to benign run or malicious run. In total, we collected on the order of tens of thousands of AVC log lines (approximately 75,000 from benign scenarios and 100,000 from malicious scenarios, reflecting that the malicious app generated a lot of denial messages in a short time).

Labeling: We then labeled each log entry with a binary label: benign (0) or malicious (1). All log lines originating from the benign monkey runs were labeled 0. All lines from the malware-injected runs were labeled 1. It's worth noting that in the malicious runs, not every single denial is directly caused by malware (some could be incidental benign events happening concurrently). However, to simplify and because malicious events dominated those sessions, we label them as malicious. This labeling strategy does introduce some noise (benign events in a malicious-labeled sequence and vice versa), but it aligns with a realistic scenario where an attack may be happening amidst normal activity. Our sequence modeling

approach (described next) is designed to handle this by looking at the pattern as a whole. We ensure that when we later split the data for training and testing, we do so by entire sessions or by random distribution such that the model can generalize.

Data Parsing and Feature Extraction

Each raw log line (as a text string) was parsed into structured fields for analysis. We identified the following key fields in an AVC log message format on Android:

- **Timestamp:** The time (to seconds or milliseconds) when the log was recorded.
- **PID/TID:** The process ID (and thread ID if applicable) that attempted the action.
- **Operation (AVC Result):** Typically "denied" (and in permissive mode, logs may also show "avc: denied (still allowed)" which could be considered as something like "granted" in our logs – indicating the action would be denied in enforcing mode but was allowed in permissive; we treated those as separate category if present).
- **Source Security Context (SContext):** The SELinux context of the subject process (e.g., `u:r:untrusted_app:s0:c152,c256` for a third-party app, or `u:r:init:s0` for the init process).
- **Target Security Context (TContext):** The SELinux context of the target object (e.g., a file or socket context like `u:object_r:app_data_file:s0:c152`).
- **Target Class:** The object class of the target (e.g., `file`, `dir`, `socket` – this is actually embedded in the SELinux context string or sometimes explicitly mentioned like "tclass=file").
- **Permission:** The specific permission attempted (e.g., `read`, `write`, `open`). In Android's log format, the permission is usually in braces after "denied { } for …". We ensured to capture this if present.
- **Process Name (comm) and App ID:** The kernel often logs the process's command name (comm) and in Android's case, the application package or process name might be included as well (for example, `comm="com.example.app"`). In our logs, we had a

field “App” which was the Android package name if available (or “NN”/blank if not applicable), and “ProcessName” which is the comm name.

After parsing, we structured the data as a table where each row is a log event with columns such as: `Timestamp, PID, SContext, TContext, Class, Perm, App, ProcessName, Label`. An excerpt (for illustration) from the benign dataset looked like:

Timestamp	PID	SContext	TContext	Class	Perm	ProcessName	Label
53:17.3	230	u:r:init:s0	u:object_r:vendor_toolbox_exec:s0	process exec	execute	init	0 (benign)
53:18.8	1	u:r:init:s0	u:object_r:atrace_exec:s0	process exec	execute	init	0
53:18.8	1	u:r:init:s0	u:object_r:atrace_exec:s0	process exec	execute	init	0
…	…	…	…	…	…	…	…

And an example from malicious logs:

Timestamp	PID	SContext	TContext	Class	Perms	ProcessName	Label

44:03.7	5640	u:r:untrusted_app:s0:c152,c256	u:object_r:app_data_file:s0:c152	file	write	com.malware.app (temp thread)	1 (malicious)
44:03.7	5640	u:r:untrusted_app:s0:c152,c256	u:object_r:app_data_file:s0:c152	file	write	com.malware.app (temp thread)	1
44:03.7	5640	u:r:untrusted_app:s0:c152,c256	u:object_r:app_data_file:s0:c152	file	read	com.malware.app (temp thread)	1
...	...	...	...	...	...	...	...

(These are illustrative; actual logs have more fields and the "granted" vs "denied" distinction for permissive mode logs.)

We then performed the following preprocessing steps:

- **Filtering and Cleaning:** We removed any log entries that were incomplete or not relevant to access control decisions. For example, if there were non-SELinux log lines captured by our filter by mistake (rare but we double-checked the filtering). Also, we dropped fields that we deemed not useful for learning, such as the precise PID numbers (which are random and not tied to a security decision) and timestamps for now (since we care more about the sequence order than the exact times).
- **Handling Missing Values:** In some cases, fields like App (package name) were empty (for system processes) or some permission info was missing. We introduced placeholder categories for these (e.g., "" or "unknown") so that they could be encoded properly rather than left blank.

- **Encoding Categorical Variables:** Most of our features are categorical strings (security contexts, class, permission, process name). Directly feeding these to a neural network is not feasible; we need numeric vectors. We considered one-hot encoding each categorical feature. However, the number of unique security context strings is quite large (every app has a unique context due to the appended category tags, and many system services have distinct contexts). One-hot encoding across all could create a very high-dimensional vector (on the order of hundreds or more features). Instead, we opted to encode each category using an **index encoding** (assign each unique value an integer ID) and then use an embedding or dimensionality reduction. We ultimately chose to apply PCA rather than learn embeddings from scratch, to ensure we capture the most variance in a fixed lower dimension.
- **Principal Component Analysis (PCA):** We formed an initial feature vector for each log entry that was essentially a concatenation of one-hot encodings for each categorical field (or a similar expanded representation). This vector space was high-dimensional (for example, if there are 50 unique source contexts, 100 target contexts, 10 classes, 5 permissions, etc., the concatenated one-hot could be hundreds of dimensions). We then applied PCA on the set of all these vectors (combining benign and malicious) to project them into a smaller feature space. We decided on **5 principal components** for the final representation of each log line, as we found that 5 components explained a majority of the variance and kept the sequence model input size manageable. Each log event is thus represented by a 5-dimensional numerical vector after PCA. We also normalized these component values (zero mean, unit variance) so that no one feature dominates due to scale.

It's worth noting that by using PCA we lose some direct interpretability of features (the components are linear combinations of original features). However, since we will later use an attention mechanism on sequence of these components, we can still trace back which log line (with original fields) was important. The PCA mainly serves to reduce dimensionality for the model; we can always inspect the original log corresponding to an anomalous vector.

To validate our preprocessing, we ensured that the PCA-transformed data of benign and malicious logs showed some separation in exploratory analysis. Indeed, when plotting the first two principal components, we observed that malicious log entries tend to cluster differently from benign ones (likely because certain contexts and permissions only appear in malicious attempts). This gave us confidence that the features contain signal that the model can learn.

Sequence Preparation

After obtaining a sequence of feature vectors (length 5 each) for the chronological log entries, we constructed fixed-length sequences for training the models. We chose a sequence length of **10 events**. This means the system will look at 10 consecutive log messages at a time to decide if within those 10 there is malicious activity or not. The choice of 10 was somewhat empirical – it provides enough context to potentially catch multi-step behavior but is short enough for the RNN to handle and for the assumption that within any 10 consecutive denials, if an attack is happening, at least one of those denials will be related to it.

Using a sliding window of size 10, we generated the sequences as follows: for the ordered log event list `[e1, e2, e3, ..., eN]` (with N events after preprocessing), we take `[e1...e10]` as the first sequence, `[e2...e11]` as the second, and so on until `[e(N-9)...eN]` as the last. This yielded (N-9) sequences. Each sequence was assigned a label equal to the label of the last event in the sequence (which corresponds to the `target_column` in our dataset). In code, this was like: `label_seq_i = label[event_{i+10-1}]`. The rationale was that if the last event is malicious (1), likely that sequence is part of a malicious scenario; if the last event is benign (0), the sequence is likely benign. This is a heuristic – it can mislabel a sequence that starts in one context and ends in another (e.g., a sequence that starts with malicious events but the 10th event is benign will be labeled benign). However, given the large number of sequences and the contiguous nature of our scenarios, this proxy labeling works in practice. We also experimented with

using majority-vote labeling for a sequence (label it malicious if ≥half of the events are malicious), which gave similar results. The differences were minor, so for simplicity we stuck with the last-event labeling.

We shuffled and split the sequences into a training set (80%) and a test set (20%), ensuring that we don't have overlap in a way that could leak obvious patterns (since sequences overlap by 9 events with neighbors, we actually performed the split on the original event list or on distinct windows to avoid nearly identical sequences in train and test). In effect, the model sees a wide variety of sequences during training, and we evaluate it on sequences it hasn't seen (from different parts of the log or from a different run).

At the end of this process, we had a large dataset of sequence examples, each represented as a 10x5 matrix (10 time steps, 5 features per step) and a binary label. This dataset is the input to our deep learning models.

Deep Learning Models

We explored three deep learning architectures for sequence classification of the SELinux log data: (1) a Bidirectional LSTM model, (2) a Bidirectional GRU model, and (3) a Bidirectional LSTM with an attention mechanism. All models were implemented using TensorFlow/Keras. The input to each model is a sequence of 10 event vectors (each of length 5, from PCA), and the output is a probability or binary prediction (malicious or benign).

Bidirectional LSTM Model

Architecture: The first model uses Long Short-Term Memory (LSTM) units, which are a type of RNN cell capable of learning long-range dependencies via gating mechanisms. We chose an LSTM because our sequences, while only length 10, could still benefit from the ability to remember earlier events when processing later ones. Moreover, LSTMs are proven effective in many sequential learning tasks due to their handling of vanishing gradients. Our LSTM model is **bidirectional**, meaning we have one LSTM reading the sequence from the first

event to the last (forward direction) and another LSTM reading from last to first (backward direction), and their outputs are combined. The intuition is that knowing both previous and next events can provide context (for example, an unusual denial might be better understood if we know what happened shortly after as well). In an offline log analysis scenario, bidirectional RNNs are suitable since we have the entire sequence available (we are not doing realtime on-device prediction yet, where causality would force unidirectional processing).

Concretely, our BiLSTM model has three stacked layers of Bidirectional LSTM:

- Layer 1: Bidirectional LSTM with 64 units (per direction) and `return_sequences=True` so that it outputs a sequence of hidden states (one for each time step). We apply a Batch Normalization and a Dropout (30%) after this layer to stabilize and regularize.
- Layer 2: Bidirectional LSTM with 32 units (per direction), also returning sequences (so now output length 10 again). Followed by BatchNorm and Dropout.
- Layer 3: Bidirectional LSTM with 16 units (per direction), returning only the last output (since this is the last recurrent layer we set `return_sequences=False`). After this, a BatchNorm and Dropout.

Finally, we have a Dense layer of size 32 with ReLU activation for further combination of features, and then a Dense output layer of size 1 with sigmoid activation (for binary classification output). This architecture was determined after some experimentation – we found that three layers gave a slight boost over one or two, presumably by capturing hierarchical patterns (64→32→16 units). The total trainable parameters are on the order of tens of thousands, which is modest.

Training: We compiled the model with the Adam optimizer and binary cross-entropy loss. We monitored accuracy on the validation split. We trained for up to 50 epochs, with early stopping if the validation accuracy didn't improve for a few epochs. The learning rate was 0.001 initially. The model converged within about 10–15 epochs typically, reaching a high

accuracy. We also tried a simpler single-layer version which was faster but slightly less accurate. The bidirectional LSTM model, in the end, achieved around 90% training accuracy and ~88.7% validation accuracy (details in Results). This indicates it learned the patterns well without severe overfitting (train and val were close).

Bidirectional GRU Model

Architecture: Gated Recurrent Unit (GRU) is a newer RNN cell that is a simplified version of LSTM, using fewer gates (it combines the forget and input gates into one "update" gate) and often needing fewer parameters to achieve comparable results. We constructed a BiGRU model with the same overall structure as the LSTM model: three stacked Bidirectional GRU layers with 64, 32, 16 units respectively, each followed by batch normalization. We kept dropout at 30% on the first layer; we found on further layers it didn't make much difference so it was primarily used on the largest layer. The output of the last GRU layer (16 units per direction, combined) goes into a Dense(32) ReLU layer and then the sigmoid output layer.

We expected GRU to possibly perform similarly to LSTM but maybe train a bit faster due to fewer parameters. Indeed, the GRU model had slightly fewer parameters but on the same order. The bidirectional mechanism was the same as well.

Training: The training procedure for the GRU model was identical (Adam optimizer, etc.). The GRU model converged a bit faster (each epoch also ran slightly faster). After training, its performance was very close to the LSTM's. In our experiments, the BiGRU reached about 89.6% training accuracy and 88.8% validation accuracy – essentially on par with the LSTM (within statistical variance). This suggests that both LSTM and GRU had enough capacity to model the sequences and the task wasn't highly sensitive to the exact RNN cell type. In terms of inference speed, GRU being simpler could be an advantage for deployment, but in our case (offline analysis) this was not a primary concern.

BiLSTM with Attention (Attention-Enhanced Model)

Motivation for Attention: While the plain BiLSTM (and GRU) models produce good accuracy, they act as black boxes in terms of which events in the sequence influenced the decision. If we want to use the model's output to adjust SELinux policy, it's crucial to know *which log entry (or entries)* in a 10-length sequence triggered the model to classify it as malicious. This is where we introduce an attention mechanism. An attention layer can learn to assign weights to each time step's output from the BiLSTM, effectively telling us how important each event is to the final classification (Presentation 6new.pdf) (Presentation 6new.pdf). In sequence classification tasks, attention has been used to improve performance and provide interpretability by focusing on relevant parts of the sequence.

Architecture: Our attention-enhanced model starts similarly with a bidirectional LSTM stack. In fact, we used a slightly smaller architecture here: two BiLSTM layers (64 and 32 units) instead of three, to reduce complexity (since attention will add some parameters as well). The BiLSTM layers have `return_sequences=True` so that the second layer outputs a sequence of hidden states `[h1, h2, ..., h10]` for the 10 time steps. We then add a custom Attention layer. We implemented a simple attention mechanism: we take a weighted sum of the sequence of hidden states, where the weights are learned from the hidden states themselves. In practice, this can be done by having a small feed-forward network that takes each hidden state `h_t` and produces a scalar score, then softmax across t=1..10 to produce attention weights `α_t`. These weights indicate how much attention the model pays to each time step. The sequence of hidden states is then reduced to a context vector `c = Σ (α_t * h_t)`. This context vector `c` is then fed into a Dense layer (and then to output) for classification.

In Keras, we implemented attention by manually computing those weights. Another approach we tried was to use a MultiHeadAttention layer treating the sequence as both query and key ("self-attention" like Transformer encoder) (major_project14feb_0_(2).ipynb). We initially experimented with a Transformer-based encoder (with multi-head attention) for

this task (detailed later in Future Work perhaps), but we found that for this relatively short sequence classification, a simpler attention on top of BiLSTM was sufficient and easier to train with our data volume. So our final attention model uses a single-head attention: essentially learning a vector that scores each hidden state.

After obtaining the context vector via attention, we pass it to a Dense(16) with ReLU and then the sigmoid output. (We found that adding too many dense layers on top didn't help; the main learning was in the sequence layers and attention.)

Training: We trained this model similarly. One difference: to avoid overfitting and encourage the attention mechanism to be effective, we used a slightly lower learning rate (1e-4) and added early stopping. The model took maybe slightly longer to converge because attention adds some complexity. The results were pleasing – the attention model reached ~89.95% training accuracy and about **89.10% validation accuracy**, edging out the previous models by a small margin. The improvement is minor (~0.3% over GRU in validation accuracy), but consistent. More importantly, the model gives us insight into which parts of the sequence mattered, which the others did not provide.

We also verified that the attention weights indeed made sense: in sequences that were labeled malicious, the model often put high weight on one or two specific log events – those turned out to be the log entries corresponding to the malicious app's actions (e.g., an attempted access to `system_data_file`, or a denial in a system process right after the malicious app ran). In benign sequences, the attention weights were more evenly distributed or focused on benign routine denials, but since the model output would be benign in those cases, it's less of a concern. The interpretability and slight performance boost suggest that the attention mechanism not only helps explain the model but also maybe filters out noise by focusing on the truly anomalous event in a window.

To summarize the architectures in terms of performance and complexity:

- **BiLSTM:** 3-layer, ~90% train, 88.7% val accuracy.
- **BiGRU:** 3-layer, ~89.6% train, 88.8% val accuracy.

- **BiLSTM+Attention:** 2-layer + attention, ~89.95% train, 89.1% val accuracy.

We will present a comparative evaluation of these models in the next section, including a performance table and analysis of why their accuracy is high and so close to each other. We will also discuss any misclassifications and what features might be causing them, as gleaned through the attention mechanism.

Experimental Results

After training the models as described, we evaluated their performance on the test set of log sequences (which the models had not seen during training). We report the primary metric as **accuracy** (the fraction of sequences correctly classified as benign or malicious). Given the nature of our data (we tried to balance benign and malicious sequences, though the malicious ones were slightly more due to how we generated data), accuracy is a reasonable metric. We also looked at precision and recall for the malicious class to ensure the model is actually catching a high percentage of malicious sequences (high recall) while not falsely flagging too many benign sequences (precision).

Model Performance Summary: *Table 1* summarizes the training and validation accuracy for the three models we built:

(Presentation 6new.pdf) (Presentation 6new.pdf)

Model	Training Accuracy	Validation Accuracy
Bidirectional LSTM	90.32%	88.72%
Bidirectional GRU	89.64%	88.80%
BiLSTM + Attention	89.95%	89.10%

Table 1: Performance of different models on training and validation data.

All three models achieve high accuracy (~88-89%) on the validation set, indicating that they can effectively distinguish malicious vs. benign log sequences most of the time. The training accuracy is only slightly higher than validation (less than 2% difference), which suggests that overfitting was minimal and the models generalize well to unseen data. This is likely due to our use of dropout and the relatively large size of the training dataset.

The **Bidirectional LSTM** and **Bidirectional GRU** models performed almost identically in validation accuracy. This was not unexpected; both are powerful sequence learners and our sequence length is small (10), so neither had a clear advantage in capturing very long dependencies. The GRU's slightly higher validation accuracy (88.80% vs 88.72%) is basically negligible. The GRU had slightly fewer parameters, so one could prefer it for efficiency, but otherwise, the choice between LSTM and GRU did not significantly affect detection capability in this scenario. Both had a validation accuracy around 0.87-0.88, which corresponds to misclassifying about 12-13% of sequences.

The **BiLSTM with Attention** showed a validation accuracy of 89.10%, the highest among the three, albeit by a small margin (~0.3% above GRU). This small boost suggests that the attention layer was beneficial, possibly by focusing the model on the most discriminatory features in the sequence and filtering out some noise. More importantly, the attention model provides interpretability; even if its raw accuracy gain is minor, the insight it offers is valuable (as we will discuss).

We also computed the confusion matrix for the best model (BiLSTM+Att) on the test set. It showed that out of the malicious sequences, about ~91% were correctly identified (giving a **recall** of 0.91 for malicious sequences), and ~87-88% of benign sequences were correctly identified (so precision for malicious was around 0.88 as well given similar distribution). These are strong results, meaning the model rarely misses a truly malicious sequence and has a moderate false positive rate (flagging ~12% of benign sequences incorrectly). In practice, some false positives are acceptable because they might correspond to either unusual benign behavior or slight mislabeling in our data; those can be further examined by an admin. The high recall is desirable for security (catch all the bad events).

Comparison and Analysis: The fact that all models achieved high accuracy indicates that there are clear differences between the log sequences of benign and malicious activity that the models can pick up on. Through the training process, the networks likely learned certain signature patterns. For example, the models might learn that if in a sequence there is a denial involving a *system* context targeted by an *app* context (like an app trying to access `system_data_file` or `system_server`'s resources), that is a strong indicator of malicious behavior (since normal apps shouldn't attempt that). On the other hand, benign denials might often be an app trying to do something slightly outside its realm but still within its general domain (like a third-party app trying to access a file of another third-party app might be common benign error). The models possibly learned the combinations of SContext and TContext (via PCA features) that correlate with attacks.

One interesting observation was the importance of certain SELinux context attributes. In many malicious sequences, the source context was always `untrusted_app` (as expected, the malicious app runs with that context) and the target contexts were often those with high privileges (like `system_data_file`, `diag_device`, etc.). The models likely identified these as anomalous compared to benign sequences where you might see `untrusted_app` accessing `app_data_file` (which is normal, albeit if it's not its own maybe a benign scenario of app trying to access another app's data which is blocked but common). The PCA helped compress these into signals the models could use.

Error Analysis: Among the ~12% of sequences that the best model got wrong, we inspected a few:

- Some false negatives (malicious sequences classified as benign) were sequences where the malicious actions were very sparse. For instance, if the malware only triggered one denial and it happened at the beginning of a sequence and the rest 9 events were benign denials, and if our sequence labeling happened to label that sequence malicious (because the last event of those 10 might have been malicious or maybe not and we considered it malicious in truth since that window overlaps an attack), the model sometimes missed it. Essentially, if an attack's log footprint was

tiny in a window, the model might overlook it, especially if attention weight got spread out. However, because we slide the window by one, that same attack event would appear as the last event in some other window where it should be caught. So overall detection of attacks (at least one sequence per attack flagged) was good.

- Some false positives (benign sequences flagged as malicious) were interestingly often sequences involving the **init** process or other root processes generating denials. For example, during device boot or monkey events, `init` might try to execute something that's not allowed. The model, not knowing context that boot-time denials can be benign, sometimes classified those as malicious patterns because they involve a privileged domain being denied (which in training was more often associated with malicious attempts). This is an area where adding more contextual features (like what time in boot or what process specifically) or refining labeling could help. In practice, an admin examining such a false alert would recognize it's benign and could adjust the policy or the model training data accordingly.

Ultimately, the attention model's slight improvement suggests it was a bit better at avoiding mistakes by focusing on the critical part of the sequence. For instance, if only 1 out of 10 events is truly telling, attention can weight it more and the classifier output will largely depend on that event's representation. In a plain BiLSTM, that information might get somewhat diluted in the final state.

Attention Visualization: For demonstration, we extracted attention weights for several sequences and visualized them. In a malicious sequence example, the model gave, say, 80% weight to a single log event: "avc: denied { write } for pid=5640 scontext=u:r:untrusted_app:s0:c152,c256 tcontext=u:object_r:system_data_file:s0 tclass=file". This was precisely the malicious app trying to write to a system file. That high weight indicates the model identified this as the suspicious event in that window, which aligns with expert intuition. In a benign sequence, weights might be more evenly distributed or sometimes also spiky on an event — interestingly, sometimes on an event that could look odd but was actually benign. Such cases might warrant policy adjustments (perhaps the

model is onto something that is anomalous even if not malicious, e.g., a misconfiguration causing repetitive denials that could be cleaned up).

In summary, our deep learning models effectively learned to detect malicious SELinux log sequences with high accuracy. The differences between LSTM and GRU were minimal, showing robustness of the approach. The introduction of an attention mechanism slightly improved accuracy and greatly improved interpretability, which is crucial for the next phase: using these results to adjust SELinux policy.

SELinux Policy Adjustment via Attention Insights

One of the primary goals of this research (beyond classification accuracy) is to leverage the model's insights to assist in SELinux **policy adjustment**. In operational terms, once our system flags a sequence of AVC logs as malicious, a security engineer would ask: *"Which specific denied actions were malicious, and what does that imply about our SELinux policy? Do we need to change something?"* The attention mechanism integrated into our best model helps answer this by highlighting the specific log entries (time steps in the sequence) that contributed most to the malicious classification (Presentation 6new.pdf) (Presentation 6new.pdf).

Identifying Policy Weaknesses

Attention essentially provides a weight (importance score) for each event in a sequence. For a sequence flagged as malicious, we look at the event(s) with highest weight:

- If the highlighted event corresponds to an operation that was **denied by SELinux**, it indicates that the attacker attempted something that the current policy blocked. This is good (the policy did its job), but it's still a "weakness" in the sense that the attacker found something to attempt. We should ask: was the system fully protected, or did this attempt reveal a gap? For example, if a malicious app tried to access a sensitive

file and was denied, is there any way a similar app could succeed? It might prompt us to ensure that policy is strict in all relevant domains, not just this one.

- If the highlighted event was **allowed** (which could happen if we were in permissive mode or if some malicious action actually didn't violate the current policy), then that is a direct policy weakness. It means the model caught a sequence as malicious even though SELinux didn't block one action – implying that the action should likely be disallowed by policy in the future. In our experiments, since we ran in enforcing mode, we didn't have allowed malicious actions in logs. But consider if an attack uses a flaw not mitigated by policy – it might not show up as a denial log at all (SELinux only logs denials). In such cases, our system wouldn't directly catch it since we parse denial logs. Thus, our approach is mainly about seeing what was *denied* and using that to improve policy.

Given a malicious sequence, by examining the top-weighted log lines, we can map them to the relevant SELinux rules:

- Each denial has a source context, target context, class, and permission. These correspond to a rule (or missing rule). For instance, a denial "untrusted_app -> system_data_file: file write" suggests that there was no rule allowing that, which is correct. If this is malicious, we **do not** want to add a rule to allow it; in fact, we want to ensure similar attempts from any other context are also denied. It might highlight that maybe some related context could also attempt this – essentially it reaffirms that the rule should remain denied. No policy change needed here except maybe to audit if any misconfig could allow it.
- If the model highlights an `init` process denial that it considered malicious (maybe a false positive), as was in some false positives, we as admins realize that is benign. This suggests a *policy misconfiguration* where maybe `init` is trying to do something that isn't actually dangerous and we could allow it to avoid noise. The attention helps us zero in on that specific denial. We could then decide to write a new allow rule for that case. For example, if `init` was denied executing some optional tool (as seen in

benign logs: `init` domain denied access to `vendor_toolbox_exec`), and we know it's causing spam in logs but is not a real attack, we might consider adjusting the policy to allow it (if safe) or otherwise stop the attempt.

- If the model highlights something like `untrusted_app -> app_data_file` denial and flags it malicious, that might indicate the app tried to access another app's data. Normally, that's benign (happens if an app tries to read another app's files and is blocked). If the model misclassified it as malicious, it might be because it's somewhat unusual or our training treated any cross-app access as malicious (since our malicious did cross-app too). Here, the admin sees the attention weight on that event and can realize this was actually a false alarm, and thus might refine the model or simply note it as acceptable. Policy-wise, we likely *don't* allow cross-app data access (that would break sandboxing). So no change needed, just knowledge that the model will highlight this scenario.

In essence, the attention mechanism **dynamically identifies policy-relevant events** in the log stream. Each identified event can be considered a clue:

- Clue of an attempted exploit path (if malicious): ensure policy covers it (deny rules in place, maybe create even more fine-grained logging or countermeasure).
- Clue of an overly harsh denial (if benign but flagged): consider loosening policy or clarifying the context so model (and admin) can differentiate it in future.

Assisting Policy Adjustment

Our system can be integrated into the policy management lifecycle as follows. When new applications are deployed or the system undergoes updates, one can run the system in permissive mode briefly, collect AVC logs for all denials, and feed those sequences into our trained classifier:

- If the classifier (with attention) marks a sequence as **benign**, but those events are generating denials, it suggests these denials could be candidates for allowing. We

can semi-automatically generate allow rules from those (similar to audit2allow) but with the confidence that our model thinks it's benign. This could significantly reduce the risk of using audit2allow, which normally doesn't know benign vs malicious. Essentially, we would only turn denials into new allow rules if they come from sequences that our model (and by extension, our prior knowledge) deems non-malicious. This prevents a scenario where an attacker's actions are inadvertently whitelisted.

- If the classifier marks a sequence as **malicious**, then clearly we do not add allow rules for those denials. Instead, we investigate them as potential attacks. The attention highlights which exact rule was involved. If that denial was already enforced, we might keep the policy as-is (since it worked). But we might also use that information to strengthen related areas. For example, if a malicious app tried one route and was denied, it might try another; we examine if there are similar rules it could exploit. Our model essentially provides an alert: "this pattern is bad". We ensure the policy continues to block it and possibly add logging or tighten other rules around it.

One concrete example from our experiments: The attention pointed to a denial of `write` to `system_data_file` by our malware. The policy was correctly denying it. No further action needed on that specific rule. However, it made us think: do all system files have proper types that apps cannot write to? We double-check that our device's policy doesn't accidentally label any system file with a type that apps can write (sometimes mislabeling could happen on a custom system). So indirectly, it guides a review of policy labeling.

Another example: Suppose attention highlighted `untrusted_app` trying to use the `netd` domain socket (if the malware tried some trick with network daemon). If that was denied and flagged, one might consider adding a specific audit or making sure no exceptions in policy allow similar access under certain contexts (like maybe an allow rule for a legitimate use doesn't accidentally cover this malicious use-case — attention can reveal that borderline).

In the **Vulnerability Mitigation & Policy Adjustment** slide of our presentation, we succinctly captured how our system aids policy: *"The deep learning pipeline identifies anomalous patterns in AVC logs that signal policy violations. The attention layer highlights specific time steps where suspicious activities occur. These insights are used to identify weak points or misconfigurations in current SELinux policies and enable administrators to adjust policies dynamically, restricting risky operations."* (Presentation 6new.pdf) (Presentation 6new.pdf). This sums up that the model doesn't directly change policy, but gives actionable information:

- Identify which denials correspond to potentially dangerous requests – ensure they remain denied (and monitor them).
- Identify which denials are frequent but not dangerous – consider adjusting policy to allow them to reduce noise.
- Overall, continuously refine the SELinux policy *based on real usage data*, which includes attempted attacks and normal behavior.

This dynamic adjustment is a step towards what SPRT and others envision: an automated or semi-automated SELinux policy tuning system. Our approach is data-driven from logs rather than vulnerability descriptions. It could complement other tools: for instance, one could feed the output of our model into a tool that automatically generates policy modules (a bit like audit2allow but guided). An example workflow could be:

1. Run system in a learning mode, collect logs.
2. Our model classifies sequences.
3. For each sequence classified benign that has repetitive identical denials, auto-suggest an allow rule (with admin review).
4. For each sequence classified malicious, ensure those denials are documented and maybe suggest adding even a stricter constraint (though SELinux is already default deny, sometimes one might add explicit `dontaudit` rules to silence known benign denials; in malicious case, we might remove any `dontaudit` so that even those are logged, improving visibility).

One must be cautious in automating changes: false positives from the model could erroneously label something malicious and we wouldn't want to unnecessarily restrict something because of that. That is why we envision a human-in-the-loop approach where the model output and attention are decision support for a security engineer.

In conclusion, the attention-enhanced model serves a dual purpose: **detection and explanation**. By bridging the gap between raw SELinux logs and high-level decisions (like policy rules), it helps maintain the SELinux policy more effectively:

- Malicious patterns remain **denied** and can be further scrutinized (maybe even lead to kernel-level mitigations if needed).
- Benign patterns can be **permitted** through policy updates, reducing log noise and improving system functionality without sacrificing security.

Our approach effectively creates a feedback loop: SELinux enforcement -> logs -> model analysis -> policy feedback -> updated SELinux enforcement. Over time, this can harden an Android system's security policy against known attack patterns while easing restrictions that unnecessarily burden benign apps.

Conclusion and Future Work

Conclusion

In this thesis, we developed an SELinux-based malicious log detection system for Android and demonstrated its ability to **automate the analysis of AVC denial logs** and provide guidance for policy enhancement. We began with an overview of SELinux's role in Android security, highlighting how it confines apps through a fine-grained policy and logs any disallowed actions (Security-Enhanced Linux in Android | Android Open Source Project) (Security-Enhanced Linux in Android | Android Open Source Project). We identified the challenge that these logs are voluminous and complex, making manual analysis impractical

and error-prone. To address this, we proposed a deep learning framework that treats sequences of SELinux denials as time-series data.

Using real data collected via Android's monkey tool and simulated attacks, we trained Bidirectional LSTM and GRU models to classify log sequences. The models achieved high accuracy (~89%), confirming that there are discernible differences between benign and malicious SELinux denial patterns that a neural network can learn. We introduced an attention mechanism in the BiLSTM model to improve interpretability. This attention-enhanced model not only matched or slightly exceeded the accuracy of the other models, but crucially, it provided **actionable insights** by indicating which log events (and by extension which SELinux permission checks) were most relevant to a decision (Presentation 6new.pdf) (Presentation 6new.pdf).

The results demonstrate that our system can significantly **reduce the manual effort** required to analyze SELinux logs. Instead of an admin wading through thousands of denials, the model surfaces the truly concerning sequences. The integration of the attention mechanism means the admin can see "why" a sequence was flagged – e.g., the model might effectively say *"This sequence is malicious primarily because of that denied access to a system file by an app"*. Armed with this information, administrators can take targeted action on the SELinux policy:

- If the flagged event was something that should never happen legitimately (e.g., an app trying to read another app's data), the admin ensures the policy continues to deny it (no action or maybe add a stronger audit).
- If the flagged event was actually benign but unusual (a false positive), the admin can adjust the policy to allow it and thereby remove the noise.

Overall, the system provides a form of **intelligent logging monitor** that plugs into Android's existing security architecture. By automating detection of malicious behavior at the SELinux layer, we add an intrusion detection capability to Android's defense-in-depth. And by linking it to policy adjustment, we ensure the system can **learn and improve its security posture**

over time. This approach complements traditional app analysis or malware scanning by focusing on behavior (what the app tries to do) as captured by the operating system's security mechanism.

The successful implementation and evaluation of this framework indicate strong potential for real-world deployment. It could be incorporated into Android security suites or enterprise mobile management tools to continuously monitor devices' SELinux logs for signs of compromise. Even on individual devices, an enhanced log daemon could use a lightweight model to alert users or automatically toggle certain security settings if a threat is detected. From a policy management perspective, our approach could assist developers of Android system software in refining SELinux policies more quickly during testing.

Future Work

This research opens several avenues for further exploration:

- **Scaling the Dataset and Model Generalization:** While our dataset was sizable for a prototype, it was still limited to specific scenarios (one type of malware behavior, certain versions of Android). In future work, we plan to **expand the dataset** by collecting logs from a variety of devices, Android versions, and more diverse benign and malicious behaviors. This would include logs from real user devices (with consent) to capture a wide range of benign patterns, as well as logs generated by different malware samples or penetration testing tools to incorporate many attack strategies. A larger dataset would allow training more complex models and improve generalization. We'd also address the slight label noise by more sophisticated sequence labeling (perhaps using unsupervised anomaly detection to complement our supervised labels).
- **Transformer-Based Models:** The success of the attention mechanism suggests that more advanced attention architectures could further improve detection. We explored a little the use of **Transformer** encoder models (which rely entirely on self-attention and no recurrent units). Transformers could handle longer sequences and

potentially capture global patterns more effectively. In future, we aim to test transformer-based sequence classification on this task. A transformer might allow variable sequence lengths and could consider a larger window of events without the constraints of RNN memory. This might catch more complex multi-stage attack patterns (if any). Moreover, transformers are highly parallelizable, which could be beneficial if deploying on-device or processing streams faster.

- **Real-Time Deployment:** In our current setup, we treat logs after the fact (post-mortem analysis). A future enhancement is to integrate the model into a **real-time monitoring system**. For example, as logcat produces AVC entries, a small resident agent could maintain a sliding window of the last N events and run the model inference continuously. If a sequence is classified as malicious, the agent could immediately trigger an alert or take action (such as killing a misbehaving app, or switching SELinux to enforcing if it was permissive, etc.). Real-time use would require optimizing the model for speed and low resource usage. Fortunately, our models are not huge (especially GRU version), and sequence length is short. We could also explore quantization or use of specialized ML accelerators on the device to run inference with minimal overhead.
- **Automated Policy Adjustment:** Currently, our system suggests policy changes to a human administrator. A bold future direction is to implement a degree of **automated SELinux policy self-healing or tuning**. For instance, upon detecting a malicious pattern, the system could automatically insert a temporary stricter rule (like isolating the app that caused it even more, or restricting an unexpectedly accessed resource). Conversely, if a denial is identified as benign and occurs frequently, the system might propose an allow rule and possibly apply it after testing. This is tricky – changes to SELinux policy on the fly require careful validation to avoid introducing holes. One idea is to use a staging policy: have the system maintain a shadow policy module that it can load and unload (Android supports loading supplementary SELinux modules). The system could load an allow-rule module for something it thinks is benign to see if it stops the logs and doesn't cause issues, and unload it if any problem. Similarly,

it could load a deny-rule module if it suspects something malicious that wasn't covered. This would move towards an autonomous security system. Research in this direction would need to ensure safety (perhaps integrating with the SPRT approach of verifying changes against known vulnerability classes (3649158.3657306.pdf)).

- **Multivariate Analysis and Additional Data Sources:** SELinux logs are one view of system behavior. Future work could incorporate other data (for example, system call traces, app logs, or network logs) in a multi-modal model to improve accuracy. Additionally, analyzing sequences across a longer period (not just fixed windows) or using anomaly detection (unsupervised learning) could catch novel attacks that the model wasn't trained on. Another angle is to cluster the attention-weighted events from malicious detections to discover common *attack vectors*. For instance, we might find that many flagged sequences across different malware revolve around a few key denied actions – those become priority to monitor or to fortify in policy.
- **Improving Interpretability and Trust:** While attention gives us some interpretability, attention weights are not a guarantee of model correctness. Future research can explore techniques to ensure the model's decisions are understandable and trustworthy. For example, applying techniques from explainable AI, like LIME or SHAP, specifically to sequential log data could complement attention to double-check why a sequence was labeled malicious. This is important if we move to automation – we'd want high confidence that the model isn't making arbitrary errors before it changes a policy. There is ongoing research on the faithfulness of attention as an explanation ([2212.14776] On the Interpretability of Attention Networks - arXiv); we might incorporate multiple explanation methods.

In conclusion, our work demonstrates a promising approach to marrying machine learning with system security policy. We achieved our primary objectives of detecting malicious log sequences and aiding policy management. By extending this foundation with the future directions outlined, we can inch closer to an Android platform that not only enforces strong security via SELinux, but also **adapts and learns** from attacks and benign use to become increasingly resilient over time.

References

1. **Android Open Source Project (AOSP)** – *Security-Enhanced Linux in Android*. [Online]. Available: https://source.android.com/docs/security/features/selinux. (accessed Aug. 26, 2024). [Provides an overview of SELinux's integration into Android's security model] (Security-Enhanced Linux in Android | Android Open Source Project) (Security-Enhanced Linux in Android | Android Open Source Project)
2. S. Smalley and R. Craig – *Security Enhanced (SE) Android: Bringing Flexible MAC to Android*. In **Proc. of NDSS**, 2013. [Introduces the concept of applying SELinux to Android and demonstrates how SELinux policies mitigate real Android exploits] ()
3. H. Wang, A. Yu, L. Xiao, J. Li, and X. Cao – *SPRT: Automatically Adjusting SELinux Policy for Vulnerability Mitigation*. In **Proc. of ACM SACMAT**, 2024, pp. 181–192. [Research on automating SELinux policy adjustments using vulnerability classification and audit logs] (3649158.3657306.pdf) (3649158.3657306.pdf)
4. D. E. Porter *et al.* – *Machine Learning-Based Security Policy Analysis*. arXiv:2501.00085, 2025. [Discusses various approaches (including graph-based and ML techniques) for analyzing and verifying SELinux policies; mentions tools like audit2allow for policy refinement] ()
5. **Huawei Technical Support** – *SELinux – Fundamentals and AVC Logs*. [Online]. Available: https://info.support.huawei.com/.../feature_selinux1.html. (accessed June 14, 2024). [Explains SELinux concepts and specifically notes that AVC logs record all access operations not allowed by policy, stored in audit logs] (SELinux) (SELinux)