Anomaly Detection using Negative Selection Algorithms

Report on Assignment 2

Group 28

February 22, 2026

1 Introduction

Anomaly detection in system call sequences is a foundational approach to intrusion detection. Because Unix processes communicate with the kernel through system calls, these sequences can provide critical insight into the health of a system. The core premise is that legitimate processes (such as `sendmail`) generate predictable patterns during normal behavior. In contrast, anomalous system call sequences, such as a normally silent process suddenly opening hundreds of files, can be a strong indicator of a malicious exploit.

To tackle this, the Negative Selection Algorithm (NSA) is used to classify sequences as either normal or anomalous. The NSA is a technique heavily inspired by the biological immune system. In nature, T-cells that react to the body's own cells (the "self") are eliminated in the thymus during development; only those that tolerate the self survive to recognize and attack foreign, non-self material. The algorithm computationally mirrors this biological process as it generates a repertoire of detectors that explicitly do not match any strings within a labeled "self" training corpus. At test time, any input sequence that is matched by at least one detector in the repertoire is flagged as an anomaly.

The objective of this project is to train a classifier using normal system call sequences to accurately detect anomalies in test datasets. We will be working with historical benchmark data originally collected by Stephanie Forrest's group at the University of New Mexico. Given a training corpus of normal system call sequences, the goal is to train a classifier that assigns an anomaly score to each sequence in the test sets ensuring that anomalous sequences score higher than normal ones. The test sets contain both normal and anomalous behavior. Ultimately, the quality of our classification and the algorithm's discriminative power will be evaluated using Receiver Operating Characteristic (ROC) visualizations and the Area Under the Curve (AUC) metric.

2 Methodology

2.1 Dataset

For this analysis, we utilized two primary datasets, `snd-cert` and `snd-unm`, located within the provided syscalls directory. These datasets capture the sequences of system calls (such as `open`, `read`, or `write`) made by Unix processes communicating with the kernel. Each dataset is structured into several specific files to facilitate training and evaluation:

- **Training Data** (`.train`): A single file containing sequences recorded exclusively during the normal, healthy behavior of a process.
- **Alphabet** (`.alpha`): A file containing the discrete alphabet used to encode the different system calls.
- **Test Data** (`.test`): Three separate test splits containing a mixture of both normal and anomalous system call sequences to evaluate the classifier.
- **Labels** (`.labels`): Corresponding files that provide the ground truth for the test sequences, using a binary classification system where 0 indicates a normal sequence and 1 signifies an anomalous one.

A defining characteristic of these datasets is that the recorded system call sequences are of variable lengths. Because the negative selection algorithm is designed to process fixed-length strings, this variable-length nature introduces a critical preprocessing step. Before training or testing can begin, the sequences must be split into fixed-length chunks.

2.2 Negsel2

To perform the negative selection, we utilized the provided Java-based implementation, `negsel2.jar`. This tool operates on the principle of r-contiguous detectors. By providing a ”self” training file, a specified chunk size n, and a matching threshold r, the algorithm generates a repertoire of detectors of length n. During the testing phase, the tool evaluates each input string against this trained repertoire. It calculates an anomaly score based on the number of detectors that match the input. We utilize specific command-line switches, such as the `-c` flag which instructs the tool to count the matching patterns, and the `-l` flag which converts this raw count x into a more manageable logarithmic format using the formula $log_2(1+x)$. Consequently, a higher score indicates a greater deviation from the normal training data, signaling a higher degree of ”non-self-ness.” The tool is invoked via the command line using the following structure, where the `-alphabet` switch ensures the basis alphabet matches the unique system calls found in our dataset:

```
java -jar negsel2.jar -alphabet file://<alpha> -self <train> -n <n> -r <r> -c -l <
<input>
```

2.3 Data Preprocessing and Chunking Strategy

Because the sequences stored in the intrusion detection files are no longer of a fixed length, the raw data cannot be fed directly into the algorithm. The `negsel2` tool requires fixed-length strings on its standard input, meaning we must pre-process each sequence into a set of fixed-length chunks of size n. To handle this, a two-step preprocessing and scoring strategy was developed.

2.3.1 Baseline

As an initial approach, the training file was passed as-is to `negsel2` via the `-self` flag, and chunking was applied exclusively to the test sequences. For classification, we split the sequences into chunks. Specifically, each test sequence was divided into non-overlapping substrings of length n using the following Python function:

```python
def chunk_sequence(seq: str, n: int) -> list[str]:
    return [seq[i:i+n] for i in range(0, len(seq) - n + 1, n)]
```

Each chunk was scored individually by the negative selection algorithm. To merge these counts, we averaged the individual chunk scores to produce a single composite anomaly score per test sequence. Mathematically, this is represented as: $score(seq) = mean\{negsel\_score(chunk) \mid chunk \in chunks\_n(seq)\}$ Sequences shorter than n produced no chunks and defaulted to a score of zero.

While this approach yielded reasonable results on the `snd-cert` test split 1 (achieving an AUC of 0.9374 at $n = 15$, $r = 5$), a critical practical limitation became apparent. For $n \geq 15$, runtimes exceeded one hour per parameter combination, making a thorough parameter sweep infeasible. The assignment brief explicitly warns that if running the algorithm on one set of parameters takes too long, you might run into time trouble. A more effective use of the data was required.

2.3.2 Chunked Training File for Optimization

To address the severe runtime bottleneck, the pre-processing logic was extended to the training data. The training sequences were also pre-processed into a set of fixed-length, non-overlapping chunks of length n. To optimize the process further, only the unique set of these chunks was written to a temporary file and passed to `negsel2` as the `-self` corpus.

```python
def write_chunked_train(train_seqs: list[str], n: int) -> Path:
    chunks = set()
    for seq in train_seqs:
        chunks.update(chunk_sequence(seq, n))
    tmp.write("\n".join(chunks) + "\n")
    return tmp
```

This drastically reduced the size of the "self" set presented to the algorithm. This smaller self set translates to fewer detectors needing to be matched against, which substantially lowered the runtime. Crucially, this refined approach also produced noticeably higher AUC scores on snd-unm (detailed in Table ...), suggesting that an explicitly compact and distinct set of normal patterns is highly effective at discriminating normal sequences from anomalous ones in this dataset.

2.3.3 Parameters

To tune the negative selection algorithm, two hyperparameters must be chosen: the chunk size n and the contiguous matching threshold r. To find the optimal balance for intrusion detection, we designed a grid search over the following parameter space:

- $n \in 7, 10, 15$
- $r \in 2, 3, 4, 5, 6$

Performance was initially estimated and tuned on test split 1 for both datasets, and subsequently validated on test splits 2 and 3 to ensure the classifier's generalizability.

3 Results

In this section, we evaluate the performance of the negative selection algorithm using the Area Under the Curve (AUC) metric. The AUC quantifies the algorithm's ability to successfully discriminate between normal system call sequences and anomalous ones. We present the findings in two phases: the initial baseline approach (using unchunked training data) and the optimized approach (where both training and test data were preprocessed into fixed-length chunks).

3.1 Unchunked Training Sequences

For our baseline evaluation, the training file was fed into the `negsel2` tool without explicit chunking, while the test sequences were split into non-overlapping chunks. We used the parameters $n = 10$ and $r = 4$ as a foundational starting point. To determine the optimal n and r parameters, we conducted a grid search on the first test split of the `snd-unm` dataset, because it's smaller than the `snd-cert` dataset, thus it's computationally more efficient.

We validated a stable parameter combination ($n = 10$, $r = 4$) across all three test splits for both `snd-cert` and `snd-unm`. The algorithm demonstrated solid baseline performance (Table 1), achieving AUC scores around 0.94 on `snd-cert` and ranging from 0.93 to 0.97 on `snd-unm`.

Table 1: Baseline Validation across Test Splits (Unchunked Training, $n = 10$, $r = 4$)

Dataset	**Split 1**	**Split 2**	**Split 3**
`snd-cert`	0.9350	0.9456	0.9459
`snd-unm`	0.9344	0.9778	0.9789

Table 2: AUC Grid Search on `snd-unm` Split 1 (Unchunked Training Data)

Chunk Size (n)	$r = 2$	$r = 3$	$r = 4$	$r = 5$	$r = 6$
7	0.8912	0.9396	0.9396	0.9388	0.9236
10	0.9106	0.9344	0.9344	0.9344	0.9222
15	NaN	NaN	NaN	0.9724	-

The results in Table 2 indicate that a larger chunk size generally improves discriminative power, provided r is scaled appropriately. However, a significant limitation was observed at $n = 15$ with low r values ($r \leq 4$), which resulted in NaN (Not a Number) errors. This likely occurs when the matching criteria are so loose that no distinct detectors can be generated, or all sequences are classified identically, making it impossible to compute an ROC curve.

3.2 Chunked Training Sequences

To overcome the runtime bottlenecks warned about in the assignment brief and improve detection accuracy, we applied the chunking preprocessing step to the normal training sequences as well. Only the unique chunks were passed as the ”self” corpus.

This preprocessing strategy yielded dramatic improvements. As seen in the grid search on the `snd-unm` dataset (Table 3), the AUC scores increased significantly across the board, achieving near-perfect separation (AUC = 0.9900) at parameters like $n = 7, r = 5$ and $n = 15, r = 5$. Compacting the training data into a localized set of unique chunks (e.g., only 391 unique chunks for $n = 10$ and 374 for $n = 15$) not only sped up execution time but also sharpened the boundary between ”self” and ”non-self”

Table 3: AUC Grid Search on `snd-unm` Split 1 (Chunked Training Data)

Chunk Size (n)	$r = 2$	$r = 3$	$r = 4$	$r = 5$	$r = 6$
7	0.9300	0.9700	0.9700	0.9900	0.9900
10	0.9700	0.9804	0.9804	0.9804	0.9804
15	NaN	NaN	NaN	0.9900	0.9900

Table 4: Optimized Validation across Test Splits (Chunked Training, $n = 10$, $r = 4$)

Dataset	**Split 1**	**Split 2**	**Split 3**
`snd-cert`	0.9728	0.9821	0.9839
`snd-unm`	0.9804	0.9821	0.9846

4 Discussion

5 Conclusion

5.1 How to create Sections and Subsections

References

Appendices

Task 1: Using the Negative Selection Algorithm

This section contains the questions and answers from "Your task" on page 2 of the second assignment.

1. *Compute the area under the receiver operating characteristic curve (AUC [1][2]) to quantify how well the negative selection algorithm with parameters $n = 10$ and different values of r ($r = 1$ until $r = 9$) discriminates individual English strings from Tagalog strings by using the files english.train for training and english.test as well as tagalog.test for testing. Which value of r belongs to what AUC in figure 1?*

   Plot 1 has an r-value of 1, plot 2 has an r-value of 7, and plot 3 has an r-value of 4.

2. *How does the AUC change when you modify the parameter r? Specifically, what behaviour do you observe at $r = 1$ and $r = 9$ and how can you explain this behaviour? Which value of r leads to the best discrimination?*

   When increasing the r-value pass 3 the AUC curve "flattens", making it slightly worser at discrimination. At r-values 1, 8 and 9, the AUC value is close to 0.5, making it almost no better than random guessing.
   For $r = 3$ you get the best AUC, with a score of 0.8311.

3. *The folder 'lang' contains strings from 4 other languages. Which languages can be best discriminated from English using the negative selection algorithm, and for which is this most difficult?*

   The best AUC scores for all languages where found with a r-value of 3. The AUC scores of each language compared to english are given below:

   - **Middle-english:** 0.5424.
   - **Plautdietsch:** 0.7747.
   - **Hiligaynon:** 0.8397.
   - **Xhosa:** 0.8893.

   We observe that Middle-English is the hardest language to discriminate from English. This is a pretty logical conclusion since English is a direct descendant of Middle-english and both languages share vocabulary and many patterns. The best AUC value of 0.5424 is close to random guessing and the algorithm produces only a few matches, meaning that it is very difficult to distinguish Middle-endligh from regular English.

   On the other hand, Xhosa, Hiligaynon, and Plautdietsch were much easier to discriminate. Xhosa was the easiest to discriminate with an AUC score of 0.8893, followed by Hiligaynon with an AUC of 0.8397. These languages are very distant from English as they use different consonants and combine characters differently. Plautdietsch is the hardest out of these three to distinguish from English with an AUC score of 0.7747. It could be because is a dialect of German and shares some similarities with the English language.

4. *We train the repertoire on strings of a certain length, but we could in theory input strings of other lengths. Explain in your own words the issue with providing strings that are too long/short.*

   If we use shorter strings, it will be more difficult to find matches as the window would be much smaller. This would likely result in many 0 or non-matches. Longer strings would have the opposite effect. With longer input strings, we would be able to find more matching patterns. However, the anomalous part of a long string might be overshadowed by the high number of normal matches, making it harder to classify accurately.