101087562

MSc Information Security 2024-2025

BAD-SL: A Behaviour based, Distributed Anomaly Detection Framework

Yang Yu

Submitted as part of the requirements for the award of the
MSc in Information Security

at Royal Holloway, University of London

Table of Content

Abstract

This dissertation proposes *BAD-SL* (Behaviour-Aware Distributed Split Learning), a novel framework for privacy-preserving anomaly detection in IoT and edge environments. BAD-SL combines three core components: **eBPF-based kernel telemetry**, **Convolutional Neural Networks (CNNs)** for behaviour modelling, and **Split Learning (SL)**for distributed, privacy-conscious model training. The framework addresses key challenges in modern cybersecurity, including the need for low-latency host-level observability, effective modelling of structured system behaviour, and secure collaborative learning across multiple devices.

The proposed architecture enables edge nodes to capture fine-grained system events such as syscalls and process trees using eBPF, transform them into structured features, and conduct initial inference through client-side CNN layers. Only intermediate activations are transmitted to a central server, which completes model training—thus preserving data locality and reducing communication overhead. This hybrid approach avoids raw data aggregation while supporting coordinated anomaly detection across distributed nodes.

BAD-SL contributes a scalable, extensible foundation for real-time, behaviour-based intrusion detection that balances performance, privacy, and adaptability. It sets the stage for future extensions incorporating graph-based models and adversarial resilience, supporting the development of secure, distributed AI systems for edge computing environments.

I. Introduction

1.1 The Need for Distributed AI Security Frameworks

Today, digital infrastructures are escalating pervasively as new pillar of modern society. Clouds, edges and IoT scenarios became pervasive and hyperconnected. However, this immerses continuum exposes critical gaps in conventional cybersecurity paradigms. Study shows centralised, signature database based host-based intrusion detection systems(HIDS) are ineffective in detecting context sensitive attacks (e.g., zero-day exploits, lateral movement), and not suited for environments that require decentralised governance (Mollah, Azad and Vasilakos, 2017). Heterogeneous devices in such systems are operating under the restrictions of computation and bandwidth capacities, while latency-sensitive applications cannot tolerate the delays of cloud-centric analysis (Shi *et al.*, 2016).

Furthermore, while sensitive data are generated or processed locally, enterprises face legal and ethical barriers to sharing raw telemetry, such as system logs or network traces across jurisdictions or tenants (*Data Protection Impact Assessment (DPIA)*, 2018).

These pressures converge into a clear imperative: distributed AI security frameworks that decentralise threat intelligence while preserving privacy. For example, (Verdeyen *et al.*, 2024) introduce a Distributed Edge Consensus Machine Learning (DECML) framework that enables multiple stakeholders to collaboratively train models and share insights without exposing raw data or model parameters.

1.2 Core Objectives and Research Gaps

This study proposes a distributed anomaly detection effort to address a research question:

How to construct a distributed AI driven framework for anomaly detection in IoT or edge scenarios?

In pursuing this, we identify several core objectives and gaps in the current state of the art:

Key challenges include:

- Efficiently capturing structured host behaviour (e.g., process interactions) with minimal overhead.
- Modelling complex event relationships in distributed systems while preserving data privacy.
- Balancing computational demands and communication efficiency in resource-constrained edge environments.

1.2.1 Objectives:

- **Telemetry Design:** Develop a method to capture fine-grained, structured host behaviour (such as system call sequences and process interactions) with minimal overhead. Traditional approaches often rely on user-space logging or kernel modifications that can be costly; we seek a solution that leverages modern, efficient telemetry tools.

- **Robust Behaviour Modeling:** Adapt neural network models to interpret the complex, structured sequence of events occurring on a host. Many prior HIDS either use simplistic features or focus on network traffic; here we need to model *sequential system events* (e.g., processes making syscalls) in a way that can distinguish normal behaviour from anomalies. This calls for a model that can capture both local patterns and broader context in event sequences.
- **Privacy-Preserving Distributed Deployment:** Enable collaborative detection or training across multiple devices **without exposing raw data**. The framework should allow edge devices to benefit from collective learning while ensuring that sensitive host data (logs, traces) remains local. This entails designing a distributed learning strategy that transmits only abstracted information (like learned features or partial model updates) and is resilient against inference attacks.

Additionally, we must balance **computational and communication demands**: resource-constrained edge nodes cannot run extremely heavy models, nor can we saturate networks with frequent large data transfers. These objectives highlight the gap between existing monolithic intrusion detection systems and an ideal distributed solution. Notably, prior research in host IDS has not fully combined *kernel-level telemetry*, *deep learning anomaly detection*, and *privacy-preserving distributed training* in one framework – leaving a gap that BAD-SL intends to fill.

1.3 Research Questions

Building on the above objectives, this dissertation addresses the following key research questions:

- **RQ1:** *How can eBPF be used to extract meaningful, low-overhead behavioural features from endpoints for anomaly detection purposes?* We investigate how extended Berkeley Packet Filter (eBPF) technology can serve as the backbone for host telemetry, efficiently capturing events like system calls and process metadata in real time.
- **RQ2:** *How effective is a split learning architecture in preserving privacy while maintaining detection accuracy?* We evaluate whether partitioning the model between client and server (as opposed to fully centralized or fully federated learning) can prevent leakage of sensitive data and still achieve comparable results to a traditional approach.
- **RQ3:** *What are the performance implications of deploying BAD-SL in resource-constrained edge environments?* This includes measuring inference latency, bandwidth usage, and CPU/memory overhead on edge devices and the server, to ensure the framework meets practical requirements for responsiveness and efficiency.

By answering these questions, we aim to demonstrate that a carefully designed distributed learning framework can meet the needs of modern intrusion detection: capturing detailed

system behaviours, learning complex patterns, and doing so in a way that is both privacy-aware and performance-conscious.

1.4 Proposed Solution: The BAD-SL Framework

In response to these challenges, we propose **Behaviour-Aware Distributed Split Learning (BAD-SL)** – a framework that integrates kernel-level data extraction with distributed deep learning to realize a host-based anomaly detection system for the edge. In BAD-SL, each host (edge node) runs a lightweight **eBPF-based monitor** that continuously collects system call events and other kernel signals. These raw events are transformed into a structured sequence of **features** that represent the host's runtime behaviour. Rather than sending this potentially sensitive data to a central server, each host processes the sequence locally using the first part of a CNN-based anomaly detection model. This **split learning architecture** means that only *intermediate activations* – abstracted feature representations output by the local CNN layers – are sent to a central server. The server hosts the latter portion of the model, which receives the activation and produces an anomaly prediction (e.g., benign vs malicious). During training, the server also aggregates updates and sends back minimal gradient information to update the clients' models, but at no point are raw logs or full model parameters exchanged.

By **partitioning the model** in this way, BAD-SL ensures that **raw telemetry never leaves the originating device**, directly addressing privacy concerns. At the same time, the central server can coordinate learning and detection across multiple hosts – improving overall accuracy by leveraging collective patterns – without the overhead of centralizing raw data. The use of a CNN is motivated by its ability to efficiently learn patterns in sequential data and its amenability to splitting (convolutional layers are local and can run on-device, while deeper layers can run in the cloud). In summary, BAD-SL marries eBPF's **in-kernel visibility** with the pattern-learning capability of CNNs and the **collaborative training** enabled by split learning.

In the following sections, we elaborate on the related work that informs this approach (Chapter II), the design and methodology of BAD-SL (Chapter III), the implementation and evaluation results (Chapters IV and V), and discuss the broader implications and future extensions (Chapter VI).

II. Literature Review

To ground our research, we survey relevant literature along four dimensions: **(1)** high-fidelity security telemetry via eBPF, **(2)** deep learning techniques for anomaly detection in system data, **(3)** privacy-preserving distributed learning (with emphasis on split learning), and **(4)** integrated frameworks that combine these elements. This review highlights both the progress to date and the gaps that BAD-SL aims to fill.

2.1 eBPF for High-Fidelity Security Telemetry

Modern intrusion detection and anomaly monitoring systems increasingly leverage kernel-level observability frameworks to gather detailed runtime data. Extended Berkeley Packet Filter (eBPF) has emerged as a powerful in-kernel telemetry mechanism for capturing system and network behaviour with minimal overhead. Unlike traditional logging or user-space monitors, eBPF runs sandboxed programs within the OS kernel, enabling real-time extraction of events (e.g. system calls, network packets) without expensive context switches. This capability has been harnessed in recent years to collect rich security-relevant data for analysis. For example, (Ben-Yair, Rogovoy and Zaidenberg, 2019) proposed an early eBPF-based anomaly detection system that monitored kernel performance metrics to detect behaviour deviations indicative of attacks. By intercepting low-level events via eBPF, their system could identify performance anomalies (e.g. unusual syscall patterns or resource usage spikes) that correlate with cyber-attacks.

Following these foundations, more sophisticated frameworks have integrated eBPF with machine learning to improve threat detection accuracy. (*Karma IDS: An Intrusion Detection System using eBPF and LSTM*, 2024) is a recent prototype intrusion detection system that utilizes eBPF for live packet capture at the network layer, feeding the data into a deep learning model for analysis. In Karma IDS, network packets are intercepted via an eBPF hook (using XDP) and processed by an LSTM-based classifier to identify malicious traffic in real time. Similarly, **SmartX Intelligent Sec** (Farasat, Kim and Posegga, 2024) extends this approach to router monitoring, combining eBPF/XDP telemetry with a bidirectional LSTM (BiLSTM) model to detect anomalies in routing traffic. These systems demonstrate the appeal of eBPF as a high-fidelity data source: it can tap into granular events (packet headers, system calls, etc.) and stream them directly to an ML model, enabling on-line detection with low latency.

Beyond network packets, eBPF is also used to trace system call activity and process behaviour for host-based intrusion detection. (Margaritelli, 2022) showed that eBPF can capture system call frequency histograms for running processes and feed them into an unsupervised deep autoencoder to flag abnormal process behaviour. Instead of relying on static allow-lists, this approach learns the normal syscall distribution of a process and detects anomalies when the pattern or rate of syscalls deviates significantly (which may indicate exploitation or DoS attacks). The ability of eBPF to efficiently aggregate system events (e.g. building histograms in the kernel) makes it feasible to monitor high-frequency events without overwhelming overhead.

In cloud-native environments, eBPF-based runtime security tools have been applied to container monitoring. **Tetragon**, an open source eBPF security observer, was used by (Kim *et al.*, 2025) to detect cryptojacking malware in Kubernetes clusters. Their framework attaches eBPF programs to container system calls and network sockets, capturing call traces and flow-level information for each container. By analysing this telemetry with machine learning models, the system achieved up to *99.75%* accuracy in classifying malicious crypto-mining containers, with only moderate runtime overhead added to the host. Notably, they found eBPF-based monitoring scales better than Linux's `perf` for container-heavy workloads, maintaining lower latency as the number of containers grows. This underscores eBPF's advantage in high-density or real-time scenarios, where its in-kernel execution and JIT compilation allow efficient data collection even under large volumes of events.

Another cutting-edge approach embeds parts of the detection logic directly in the kernel via eBPF. (Brodzik *et al.*, 2024) present a ransomware detection mechanism that implements simple machine learning classifiers *within* an eBPF program. By running a decision tree or small neural network in-kernel on system call features, their method can identify ransomware behaviour with minimal latency, and they report comparable accuracy to user-space models. This innovative design highlights how eBPF can not only collect data but also execute lightweight ML inference at the source, though it is limited by eBPF's constraints (e.g. bounded loops, no floating point). In general, eBPF-based security tools (both academic prototypes and industrial projects like **sysdig Falco** and **Tracee**) are gaining traction as they allow granular visibility into system events and network packets for anomaly detection. The open-source Linux eBPF ecosystem, including libraries like BCC and bpftrace, further eases development of custom telemetry collectors. These capabilities make eBPF an ideal **observability backbone** for advanced intrusion detection frameworks, providing the raw behavioural data needed to train and drive modern AI models.

2.2 Deep Learning for Anomaly Detection in System Data

With robust telemetry available, the challenge becomes interpreting the firehose of system data to distinguish normal vs. malicious patterns. Traditional rule-based or statistical thresholds often fail to detect stealthy or novel attacks. Therefore, researchers have increasingly turned to deep learning—especially neural networks capable of pattern recognition—to model complex system behaviour and detect anomalies.

2.2.1 LSTM for Sequence Anomaly Detection

Long Short-Term Memory (LSTM) networks capture long-range dependencies in sequential data, making them well-suited to model system call traces and event logs. (Du *et al.*, 2017) introduced *DeepLog*, which used LSTMs to learn normal system log sequences and detect deviations as anomalies. (Kim, Lee and Kim, 2016) demonstrated the effectiveness of LSTMs on syscall sequences for host-based intrusion detection, achieving strong accuracy. The strength of LSTMs lies in their explicit sequence modelling, but they are computationally heavy, difficult to parallelise, and slow for real-time analysis, limiting their deployment feasibility in edge environments.

2.2.2 1D CNN for Syscall and Log Sequences

One-dimensional CNNs have emerged as efficient models for anomaly detection in structured telemetry. By applying convolutional filters across syscall or log sequences, they can identify local patterns—akin to n-gram features—that signal malicious activity. (Le and Zhang, 2022)showed that CNN-based models achieved F1 scores above 0.90 on multiple log datasets, while (Redhu *et al.*, 2024)reported strong performance of CNNs on syscall anomaly detection. Compared to LSTMs, CNNs offer greater computational efficiency and are highly parallelisable, making them attractive for real-time monitoring. Although they naturally capture only short-range dependencies, deeper architectures and dilated convolutions extend their effective context window.

CNN with Attention Mechanisms. Recent work has explored improving convolutional anomaly detectors with attention modules to enhance both detection accuracy and interpretability. (Bello *et al.*, 2020) introduced **Attention-Augmented Convolutional Networks**, showing how combining convolutions with self-attention improves feature representation. (Ristea *et al.*, 2022)proposed a **Self-Supervised Predictive Convolutional Attentive Block** that integrates attention into convolutional layers for anomaly detection in visual data. In another approach, researchers applied attention-based loss functions to video anomaly detection, enabling models to focus on abnormal foreground regions(Zhou *et al.*, 2020). While these studies target image or video domains, their insights on attention-augmented convolution can inform the design of CNN+Attention for syscall anomaly detection in BAD-SL.

2.2.3 2D CNN via Encoded Representations

In domains like human activity recognition, researchers have effectively translated sequential data into 2D image representations. For instance, (Carlos *et al.*, 2024) used recurrence plots to transform inertial sensor sequences into images, enabling 2D CNN-LSTM models to achieve 97–98% accuracy on UCI-HAR benchmarks. This underscores the feasibility of encoding time-series data into spatial formats for CNN analysis, a strategy that could be explored for syscall traces within anomaly detection frameworks.

2.2.4 Hybrid CNN–LSTM Models

Hybrid architectures combine CNNs' ability to capture local motifs with LSTMs' capacity for long-term dependencies. (Akhtar and Feng, 2022) applied a CNN–LSTM hybrid to IoT malware detection, reporting accuracy improvements over standalone models. (Priyadarshini, 2024)evaluated hybrid models in distributed learning contexts, showing detection accuracy exceeding 96–99% in IoT datasets. Despite strong results, hybrid models are computationally expensive and memory-intensive, making them challenging for resource-constrained deployments.

2.2.5 Synthesis

In summary, deep learning offers multiple viable paths for anomaly detection, each with trade-offs. LSTMs provide strong temporal modelling but are slow and resource-intensive. 2D CNNs are less natural for raw syscall sequences. Hybrid CNN–LSTMs achieve high accuracy but incur significant computational cost. 1D CNNs offer the best balance of

accuracy, efficiency, and parallelisability, and their structure aligns well with split learning, where early convolutional layers can run on clients while deeper layers operate on a server. Attention-augmented CNNs provide a promising extension for interpretability and improved performance, making them a suitable candidate for future work.

2.3 Privacy-Preserving Distributed Learning (Split Learning)

In distributed environments like IoT or multi-cloud systems, centralizing all telemetry for model training raises serious privacy and bandwidth concerns. **Federated Learning (FL)** emerged as a solution by keeping data localized: each client trains a full model on its own data and only gradient updates are shared and aggregated on a central server. However, even FL has downsides for security analytics – model gradients can unintentionally leak information about the training data, and the approach requires relatively heavy computation on the client (training an entire deep model). An alternative approach gaining attention is **Split Learning (SL),** a form of collaborative learning where the neural network itself is split between client and server. In split learning, each client only trains the **first few layers** of the model on local data, then sends the intermediate activations (not raw data or full gradients) to a server which completes the forward pass and updates the remaining layers. During backpropagation, the server only returns gradients for the cut layer, so the client never exposes raw inputs or full model weights.

The appeal of split learning for security is its enhanced privacy and efficiency. Since clients transmit only intermediate features, the risk of sensitive data reconstruction is lower than in federated learning where gradients (which can be inverted to some extent) are shared.(Le and Zhang, 2022) introduced one of the seminal split learning frameworks in the healthcare domain, demonstrating that multiple parties could jointly train a deep model on patient data without sharing any raw records. They showed that splitting a network (e.g. at an intermediate convolutional layer) retained model accuracy while significantly reducing privacy leakage. In general, subsequent studies have reinforced that SL yields comparable performance to centralized training on vision and text tasks, while confining data to its source. (Gupta and Raskar, 2018) further quantified that SL can use **significantly less bandwidth** than FL when many clients participate, because only activations of one layer (often with dimensionality much smaller than the raw data) are sent per batch, rather than full model updates.

For **resource-constrained edge devices**, SL is advantageous because each client only runs a small part of the model. (Thapa *et al.*, 2022) even combined split and federated learning into a hybrid "SplitFed" scheme, aiming to get the best of both – their approach decoupled computation such that clients handle minimal model portions, and federated aggregation still occurs on server for those portions. In their experiments, SplitFed improved training speed and scalability over standard FL in an IoT anomaly detection setting, without sacrificing accuracy. This suggests that for large-scale deployments, partitioning the model (vertical split) can be more efficient than copying full models to every device, especially when device computing power or memory is limited.

Within the context of anomaly detection and IDS, privacy-preserving distributed learning is still an emerging area. A recent study by (Priyadarshini, 2024) applied federated and split learning to IoT cyberattack detection in smart city deployments. Using benchmark intrusion datasets (NSL-KDD, UNSW-NB15), they compared centralized models, standalone CNN/LSTM, federated learning, and split learning. The results showed that the deep learning models (CNN or LSTM) achieved high accuracy (95–99%) in detecting attacks, and that both FL and SL performed competitively with the centralized approach. Notably, split learning slightly outperformed federated learning in accuracy on certain datasets (99.32% vs 98.99% accuracy in one case) while also incurring lower training time than FL. This reflects the efficiency of SL's reduced communication: FL had to exchange large model updates across rounds (leading to >225 seconds training time in their setup), whereas SL's one-pass communication of activations per batch was faster (172s) for the same data. The study concluded that both FL and SL are promising for distributed anomaly detection, successfully balancing data privacy with only a minor cost in training time or complexity. It also hints that **hybrid models** or optimizations could further reduce the overhead of these techniques, a point echoed by other researchers (e.g. proposals to combine strengths of FL and SL to mitigate each other's weaknesses).

In practice, adopting split learning (SL) for security tasks involves several challenges. A key decision is determining the model "cut" layer: too early may leak sensitive activations like raw inputs, while too late places significant computation load on clients. Adding noise (e.g., differential privacy) or encrypting activations can help but complicates the setup and may affect model accuracy.

Recent empirical work demonstrates SL's feasibility. (Priyadarshini, 2024) evaluated federated and split learning for IoT-based intrusion detection in smart cities, using NSL-KDD and UNSW-NB15 datasets. The study reports detection accuracy over 99% and shorter training times for split learning compared to federated learning, while maintaining data locality and reducing data sharing.

(Pham *et al.*, 2022) introduced a binarized split learning model for mobile devices, achieving a 17.5× speedup in client-side processing with minimal accuracy loss under differential privacy constraints. These findings indicate that SL can support efficient, privacy-aware anomaly detection across distributed and resource-limited environments.

While these examples validate SL's technical viability, widespread real-world adoption remains limited. Open questions include determining secure cut-point placements, managing communication overhead, and ensuring resilience in adversarial settings—areas that require further research.

Importantly, **the structure of CNN models naturally complements split learning**. As noted above, CNNs can be split such that convolutional layers reside on the client and the remaining layers on the server. This is precisely one reason BAD-SL (our proposed framework) chooses CNNs: their hierarchical feature extraction can be neatly partitioned. The client-side CNN layers process raw telemetry (e.g. eBPF-collected events) into intermediate feature maps, which are then sent to the server. The server-side layers (e.g. fully connected layers) then perform classification or anomaly scoring. This division

means the data leaving the client is an encoded representation, not the sensitive raw events, enhancing privacy. It also reduces communication load, since the size of intermediate activation is typically much smaller than the original input stream. Several authors have highlighted this synergy: CNN-based feature extraction is well-suited for split learning deployment across edge and cloud. In contrast, other advanced models like Graph Neural Networks (GNNs) would be much harder to deploy in a split fashion due to their need for global graph aggregation. Thus, the combination of **eBPF (for local data capture), CNN (for feature learning), and Split Learning (for collaborative training)** appears to be a promising recipe for distributed anomaly detection, offering a balance of visibility, accuracy, and privacy.

2.4 Comparative Analysis of Relevant Frameworks

Table 1 compares several representative frameworks and studies that integrate some or all the above components (telemetry method, learning model, privacy mechanism, and deployment architecture). Both academic prototypes and industrial systems are included to illustrate the state of the art:

Framework / Study	Telemetry Source	ML Model	Privacy Mechanism	Deployment
Ben-Yair et al. (2019) – eBPF Anomaly Detector	eBPF (Linux kernel perf events)	Shallow Neural Network / Heuristics	None – Centralized	Single host (kernel-level monitoring)
Karma IDS (2024) – Dev. Community project	eBPF/XDP (live packet capture)	LSTM (seq. deep network)	None – Local/On-host	Distributed hosts (each host runs eBPF + LSTM
SmartX Intelligent Sec (2024) – Farasat *et al.*	eBPF (router telemetry via XDP)	BiLSTM (deep RNN)	None – Centralized (cloud analysis)	Edge routers sending data to central server
Akhtar & Feng (2022) – Real-time Malware Det.	System API calls (IoT devices)	Hybrid CNN + LSTM	None – Centralized	Single device or centralized server analysis
Kim et al. (2025) – Cryptojacking Detector	eBPF (Tetragon: syscalls + flows)	RNN (LSTM) and classical ML (comparative)	None – Centralized analysis	Kubernetes cluster (monitor each container)

Framework / Study	Telemetry Source	ML Model	Privacy Mechanism	Deployment
Priyadarshini (2024) – IoT FL vs SL	Network telemetry (IoT sensors)	CNN, LSTM (for baseline); FedAvg and SplitNN for distributed	**Federated Learning**, **Split Learning**	Distributed IoT devices + central server
Brodzik et al. (2024) – In-Kernel ML	eBPF (syscall interception)	Decision Tree; MLP (in-kernel)	None – Local (in-kernel)	Single host (kernel-space detection)

Table 1: Comparison of related frameworks in terms of data telemetry, ML technique, privacy approach, deployment architecture, and performance results. Academic references are cited in brackets.

As shown in Table 1, a variety of frameworks have been proposed, each addressing different parts of the overall BAD-SL vision. Several observations can be made:

Telemetry Method: Nearly all the surveyed systems that focus on host or network anomalies leverage eBPF or similar kernel instrumentation to gather data. This includes syscall tracing for processes (Ben-Yair, Rogovoy and Zaidenberg, 2019; Brodzik *et al.*, 2024) and packet interception for network traffic (Karma IDS, SmartX). High-fidelity telemetry is critical for catching subtle anomalies. A few others rely on standard logs or API call traces (Akhtar and Feng, 2022) where eBPF was not used, but even those could potentially benefit from kernel-level data extraction. Overall, eBPF has become a **de facto** choice for modern Linux-based monitoring due to its performance and richness of data.

ML Model: There is a clear trend toward deep learning models (LSTM, CNN, or hybrids) for analysing the telemetry. Simpler models (decision tree, SVM) have been tested (e.g., in cryptojacking detection or in-kernel experiments) but generally, deep neural networks show superior accuracy. CNNs and RNNs dominate because of their ability to capture sequential and structural patterns in the data. Notably, CNNs were used in Akhtar & Feng's malware detector and evaluated in Priyadarshini's IoT study (where CNN reached \~99.1% accuracy), while LSTMs were used in multiple systems (Karma IDS, SmartX) for their strength in sequence modelling. The combination of CNN+LSTM appears effective in some cases by marrying spatial and temporal pattern recognition. No frameworks in this comparison used GNNs or other exotic models for detection, reflecting the complexity of applying graph learning in this domain (despite research interest in modelling system entity relationships with GNNs).

Privacy Mechanism: Most existing solutions do **not** incorporate privacy-preserving training – they perform centralized analysis (all data to one place) or on-device inference without collaboration. The notable exception is (Priyadarshini, 2024), which explicitly implemented Federated Learning and Split Learning for training an anomaly detector

across IoT devices. That work demonstrates the feasibility of training collaboratively without sharing raw data. It also highlights the trade-off: FL and SL incurred higher training times than a single-device model. Apart from that study, the lack of privacy mechanisms in other frameworks reveals a gap: current eBPF/ML-based security tools typically assume data can be pooled or processed locally, which may not hold in cross-organization or regulated settings. There are open-source initiatives (e.g. **OpenMind PySyft** in 2025) providing libraries for federated learning and encrypted data science, but their use in security anomaly detection is still nascent.

Deployment Architecture: The approaches vary from purely on-host (Brodzik's in-kernel detector, which is as local as it gets) to fully centralized (e.g. SmartX sending router data to a cloud ML engine). A few adopt a **distributed architecture**: Karma IDS suggests each host runs its own detection agent (no central coordination), whereas Priyadarshini's FL/SL involves edge devices collaborating with a server. The BAD-SL framework aligns with the latter philosophy – a distributed set of hosts training a global model. The table indicates that none of the surveyed prior works integrates **all three** aspects (eBPF telemetry + deep learning model + privacy-preserving distributed training) in one system. This underscores a novelty in BAD-SL: combining kernel-level data extraction, CNN-based anomaly modelling, and split learning deployment, which has not been realized in existing literature.

2.5 Research Gaps and Challenges

In reviewing prior works, several practical research gaps emerge, motivating the development of the BAD-SL framework. These challenges concern integration of techniques, deployment feasibility, and robustness in real-world settings.

Integration of Components: No existing framework fully integrates **kernel-level telemetry, deep anomaly detection, and distributed privacy-preserving training** in a unified solution. Current studies typically address these components in isolation or in partial combinations. For example, eBPF-based monitoring with local ML (Karma IDS, SmartX) lacks privacy preservation; conversely, federated anomaly detection (Priyadarshini 2024) used distributed learning but not kernel-specific telemetry. This gap suggests an opportunity to combine eBPF + CNN + SL to leverage strengths of each. As noted by our survey, "current work rarely combines eBPF data with distributed deep learning". BAD-SL aims to fill this gap by marrying eBPF's fine-grained visibility with split learning's collaborative model training.

Privacy and Data Governance: Beyond the technical combination, there is a lack of real-world implementation of **privacy-aware anomaly detection**. Enterprises are often unwilling or unable to share raw telemetry (system logs, network flows) due to legal and privacy constraints. Federated or split learning could alleviate this, but so far there are only proofs-of-concept in research. We identified only a few studies (mostly recent, 2023–2024) that even attempt IDS in a federated manner. This is a gap between research and practice – demonstrating that a distributed learning approach can work on actual organizational infrastructure (with heterogeneous devices, unreliable networks, etc.) remains an open challenge. Overcoming issues like client dropouts, data imbalance

among hosts, and security of the aggregation (against poisoning attacks) will be crucial for deployment in sensitive environments.

Dynamic and Evolving Threats: Handling concept drift and evolving behaviour is another challenge under-addressed in current literature. Many anomaly detection models are trained on static datasets or assume a relatively stable "normal" profile. In dynamic environments (IoT networks, cloud workloads), normal behaviour can change over time – software updates, workload shifts, or benign configuration changes can all invalidate a previously learned model. If not addressed, this leads to either degraded detection (missing new attacks) or false alarms. (Kermenov *et al.*, 2023) highlighted this by proposing a concept drift adaptation method for an industrial robot anomaly detector, illustrating that retraining or adapting models is necessary in long-running systems. However, few security frameworks incorporate online learning or drift detection mechanisms. BAD-SL will need to consider mechanisms for periodic model updates or continual learning across the distributed hosts to remain effective in the face of changing baselines. This is complicated by the privacy setting – we may need to adapt collaboratively without centralizing data, a topic largely unexplored (most Split/FL studies use fixed training sets). Addressing **"how well different models handle changing system behaviour"** is explicitly noted as an open question.

Evaluation in Realistic Settings: A gap exists in terms of **evaluation realism**. Many studies validated their frameworks on laboratory datasets (e.g. NSL-KDD, UNSW-NB15, BOT-IoT) or within limited testbeds. These are valuable for initial comparisons but may not capture issues that arise in production: the impact of background noise in telemetry, the performance on resource-constrained edge devices, and integration with existing security operations. To date, there is scant evidence of a full-fledged eBPF+ML system deployed across, say, a fleet of enterprise endpoints or a city-wide IoT deployment. This lack of real-world deployment experience means questions about scalability, maintenance, and interoperability are unanswered. For instance, how do we manage eBPF programs on hundreds of hosts concurrently? Can a split learning anomaly detector train continuously without disrupting operations? What about combining host-level anomaly alerts with network context? These practical considerations form a gap between research prototypes and deployable solutions.

Security of the ML Pipeline: An often-overlooked challenge is **the security of the anomaly detection framework itself**. As we embrace ML and distributed training, new attack surfaces appear – adversaries might poison the training data or manipulate the model (especially in federated/split settings where the server could be untrusted or clients compromised). Recent works have begun to explore attacks on split learning (e.g. inference or backdoor attacks on intermediate activations) and defences, but none of the surveyed anomaly detectors incorporated such considerations. For BAD-SL, ensuring the robustness of the split learning process against manipulation will be important for true real-world resilience. This remains a largely open research area: how to robustly aggregate contributions from potentially compromised hosts, or how to detect if an attacker is trying to corrupt the distributed model.

In summary, the literature provides strong evidence that each individual pillar of BAD-SL is viable – eBPF can efficiently capture behavioural data, CNNs (and similar deep models) can achieve high detection accuracy on structured telemetry, and split learning can enable collaborative training with privacy. However, the synthesis of these technologies into a cohesive, deployed anomaly detection framework has not been realized. By addressing the above gaps, integrating kernel observability with distributed learning, handling dynamic behaviour, and evaluating in realistic settings – BAD-SL aims to push the boundary of what prior works have achieved. This review of existing work supports the premise that combining system observability with privacy-preserving AI is not only necessary but technically feasible. The subsequent methodology and design chapters will build on these insights to outline how BAD-SL will be implemented to advance state-of-the-art in distributed intrusion detection.

III. Methodology

3.1 Introduction

In this chapter, we detail the methodology for developing and evaluating **BAD-SL**, a novel **BPF-based Anomaly Detection system using Split Learning**. The primary objective of BAD-SL is to provide a **privacy-preserving, distributed host intrusion detection system (HIDS)** that leverages **kernel-level telemetry**, a deep learning model, and a collaborative learning strategy. Specifically, BAD-SL monitors **system call sequences** via eBPF (extended Berkeley Packet Filter) probes on each host, processes this telemetry through a **1D Convolutional Neural Network (CNN)**, and employs **Split Learning (SL)** to train and infer across multiple devices without sharing raw data. This design directly addresses gaps identified in the literature: traditional HIDS often require sending sensitive log data to a central server or cloud (raising privacy concerns), and many rely on outdated datasets or non-distributed architectures. By contrast, BAD-SL keeps raw data on endpoints and only transmits intermediate features, thus aiming to preserve privacy while maintaining detection accuracy.

Our methodology is grounded in **design decisions informed by the literature**. Prior work underscores that **system call monitoring** is a powerful technique for detecting attacks, especially as network traffic encryption limits the effectiveness of network-based IDS. We therefore focus on system calls as our primary data source. We use **eBPF** for telemetry because it enables low-overhead, real-time capture of kernel events (system calls) directly from the OS kernel. This approach aligns with modern security practices – for example, eBPF has been successfully integrated into intrusion detection tools like Falco and deployed by industry for runtime threat detection.

For the anomaly detection model, we considered various sequence modelling approaches. Deep learning models such as **Long Short-Term Memory (LSTM) networks** have proven effective at modelling system call sequences, but they incur high computational cost and sequential processing latency. Meanwhile, recent research and our literature review indicate that **CNN-based models** can achieve comparable accuracy on sequential data with greater efficiency. We therefore chose a 1D CNN as the core model for BAD-SL, aiming to balance detection performance and runtime efficiency (this choice is justified with comparisons later in §3.4.1). Finally, to **enable collaborative training without raw data sharing**, we adopt a **Split Learning** strategy. SL allows a model to be partitioned between the client and server, so that each client trains only a partial model on its data and shares only intermediate activations. This method has been proposed as a privacy-preserving alternative to traditional centralized or federated learning, since the raw training data never leaves the client device. In summary, BAD-SL's methodology marries these components – eBPF-based data capture, CNN-based anomaly detection, and split learning – to fulfil the requirements of a distributed, privacy-conscious HIDS. The following sections describe the system architecture, dataset selection and preprocessing, model design, implementation details, and experimental setup in detail.

3.2 Architecture Design

Overview: The BAD-SL framework comprises a client-server architecture with multiple distributed client agents and a central server. **Figure 3.1** illustrates the system architecture. Each client device (host) runs an **eBPF-based telemetry module** that intercepts kernel events (specifically system calls) and collects them as a stream of data. This raw telemetry is passed to a **client-side preprocessing and feature extraction module**, which encodes system call sequences into a suitable input format for the machine learning model. The **1D CNN model** itself is split between the client and the server: an **encoder portion** of the network resides on the client, and the **remaining layers** (the classifier portion) reside on the server. During operation, the client processes incoming telemetry through its local model layers, then **offloads the intermediate activations** (the cut-layer outputs) to the server. The server completes the forward pass through the final layers to produce an anomaly detection result (e.g. a probability or classification of the sequence as normal or malicious). The server then returns the inference results or alerts back to the client or a monitoring console. This split approach means that clients never transmit raw system call data – only learned features – thereby enhancing privacy.

Architecture Components: BAD-SL's architecture can be divided into the following main modules:

- **Kernel Telemetry via eBPF:** On each host, a lightweight eBPF program is attached to system call tracepoints to log events of interest. The eBPF scripts (developed using the BCC library and/or bpftrace) capture each system call made by any process, recording details such as the call identifier, process ID, etc., and buffer these events in memory. This telemetry extraction runs continuously with minimal overhead, thanks to eBPF's in-kernel execution model. The design leverages eBPF's ability to **"see and understand all system calls"** at the kernel level, giving BAD-SL fine-grained observability into process behaviours. The raw event stream is passed to user space through ring buffers. A user-space daemon on the client collects these events and organizes them into system call sequences per process execution (or a time window). It can also enrich the events with context (e.g., process tree hierarchy, user ID) if needed for later analysis.
- **Client-Side Preprocessing & Feature Embedding:** Before feeding data into the neural network, each client performs preprocessing on the collected syscall sequences. This involves encoding system call names to numeric IDs (using a consistent syscall table, as described in §3.3), segmenting or aggregating sequences, and applying any necessary transformations. The client then applies the **first part of the CNN model** – typically an embedding layer and one or more convolutional layers – to transform the sequence into an intermediate feature representation. This can be seen as an **encoder** that produces a high-level embedding of the syscall sequence. Importantly, this step reduces the data dimensionality and abstracts the raw features, so that the data sent to the server reveals significantly less sensitive information than the original sequence. We designed the client-side processing to

be lightweight: the convolutional encoder was kept shallow enough to run within the limited CPU/memory budget of edge devices (2 vCPUs, 2 GB RAM in our testbed).

- **Split CNN and Activation Offloading:** We **partition the CNN model** at a specific cut-layer into a **client-side part** (as above) and a **server-side part**. The client forwards the **intermediate activations** (output from the cut-layer) to the server over the network. The server hosts the remaining layers of the CNN, which act as a **classifier** that takes the received activation and produces a detection output. In our design, the server-side model includes the deeper convolutional layers (or fully connected layers) and the final output layer (sigmoid for binary classification of "anomalous" vs "normal"). The motivation for where to cut the model is to balance **privacy, communication cost, and computational load**: we want the client to handle enough processing to obfuscate raw data and maybe reduce its dimensionality, but not so much that the client's resource limits are exceeded. We evaluated different cut-layer positions (see Appendix B for experiments profiling activation sizes and model accuracy at various split points). Based on these experiments, we chose to cut the model after the first convolutional block (details in §3.4.2), which yielded a manageable activation tensor size for transmission (on the order of a few thousand floats per sequence) while preserving model performance. The offloaded activations are transmitted over a secure channel to the server.
- **Server-Side Inference and Collation:** The server receives the activation from each client, feeds it through the server-side sub-model to compute the anomaly prediction, and then collates results. In a **training scenario**, the server also uses these activations to compute gradients for the server-side weights (and sends gradient updates back to the client for its portion of the model, as part of the split learning training loop – see §3.6.1). In an **inference-only deployment**, the server simply classifies each incoming activation. The server may aggregate outputs from multiple clients, raising alerts or performing further correlation if, for example, simultaneous anomalies are detected on multiple hosts. For the scope of this work, each sequence from each client is treated independently – the server returns an anomaly score or label to that client (or logs it centrally). The architecture could be extended with an alert management module, but that is outside our current focus.

System Deployment: The BAD-SL system was deployed in a virtualized testbed using containerization and orchestration tools. The entire environment configuration – including eBPF setup on clients, container deployment of model components, and networking – was automated using **Ansible** (see Appendix A for the full Ansible playbooks and the K3s topology diagram). Each client and the server run as separate nodes in the cluster, communicating over a virtual network. Figure 3.1 (insert) provides a diagram of this system architecture, showing the flow from telemetry capture on clients to model splitting and inference on the server.

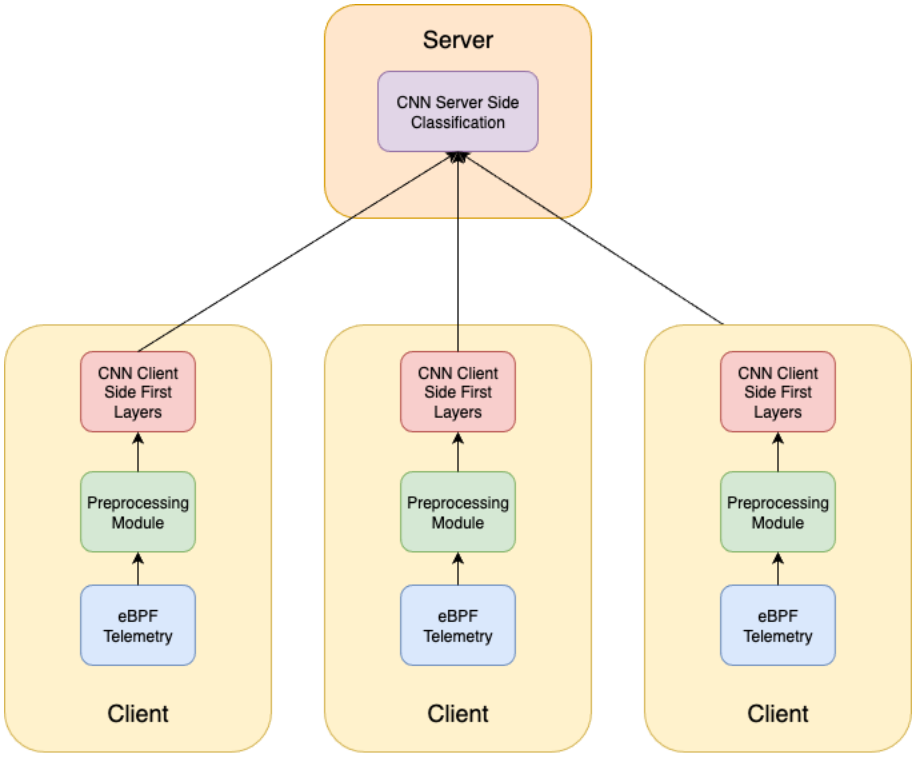


Figure 3.1: BAD-SL system architecture.

By designing the architecture in this modular, split manner, we ensure that BAD-SL meets its key objectives: **(1)** capturing detailed runtime behaviour via kernel-level data (addressing the need for fine-grained host monitoring), **(2)** leveraging an efficient deep learning model for detection, and **(3)** protecting data privacy by never sharing raw logs off-host. In the next sections, we discuss the selection of dataset and data preprocessing pipeline used to train and evaluate this system, followed by the model architecture and learning strategy in greater detail.

3.3 Dataset Selection and preprocessing

To train and evaluate BAD-SL, a suitable dataset of system call traces was required. We surveyed several publicly available **host-based intrusion detection datasets** and compared their characteristics to identify the best fit for our framework. This section first outlines the candidate datasets considered and justifies the final selection (DongTing(Duan *et al.*, 2023)). We then describe the structure of the chosen dataset and the preprocessing steps undertaken to prepare the data for our model.

3.3.1 Dataset Comparison and Justification

We considered four datasets that are commonly referenced in recent HIDS research or relevant to system call-based anomaly detection:

- **DongTing (2022)**(Duan *et al.*, 2023) – A modern large-scale dataset from China focusing on Linux kernel anomaly detection.

- **ADFA-LD (2013)**(Creech, 2014) – The ADFA Linux Dataset from Australia, a widely used older system call dataset for host IDS.
- **PLAID (2015)**(Gao *et al.*, 2014) – The PLAID (Lab **P**ublic **L**og **A**rtificial **I**ntrusion **D**ataset) from the US, consisting of application-level logs.
- **LID-DS-2021 (2021)** (Grimmer *et al.*, 2023)– The Leipzig Intrusion Detection–Dataset 2021 (published 2023, Germany), which includes system call data mixed with other sensor data.

Table 3.1 summarizes key attributes of these candidate datasets and our assessment of their suitability for BAD-SL.

Dataset	Origin (Year)	Data Type	Kernel-Level?	Labelled?	Suitable for eBPF?	Notes
DongTing	China (2022)	System calls, some kernel logs	Yes	Yes	Yes	Modern, large-scale; designed for kernel anomaly detection. Tailored to eBPF capture (recent kernels).
ADFA-LD	Australia (2013)	System calls (sequences)	Yes (Linux)	Yes (binary + attacks)	Yes (limited)	Contemporary for its time, but older kernel and synthetic attack traces. Moderate size (thousands of traces).
PLAID	USA (2015)	Application logs (messages)	No	Yes	No	Application-level log dataset; lacks direct syscall information (no kernel call structure). Not directly usable for eBPF-based analysis.
LID-DS-2021	Germany (2021)	System calls + sensors mix	Yes	Yes	Yes	Recent and comprehensive HIDS dataset.

Dataset	Origin (Year)	Data Type	Kernel-Level?	Labelled?	Suitable for eBPF?	Notes
						Contains variable-length syscall sequences and additional telemetry. Large and complex (designed for comparative analysis).

Table 3.1: Candidate datasets for system call-based anomaly detection and their characteristics.

After evaluating these options, we **selected the DongTing dataset** for our study. The decision was based on several reasons:

- **Relevance and Alignment:** DongTing is explicitly designed for **Linux kernel anomaly detection**, which closely aligns with BAD-SL's focus on kernel-level telemetry. It contains **system call sequences obtained via kernel instrumentation** (spanning multiple kernel versions), which makes it ideal for our eBPF-based data collection approach. In contrast, PLAID's application logs do not contain syscalls, making it unsuitable for our purposes. LID-DS-2021, while relevant, includes mixed data (e.g. sensor readings) and is more complex to integrate; our framework currently focuses purely on syscalls. ADFA-LD, though a classic benchmark, is relatively outdated and limited in size/variety.
- **Modern Coverage:** DongTing was released in 2022 (with an associated 2023 Journal publication), making it one of the most up-to-date and **comprehensive system call datasets** available. It addresses the coverage/timeliness concerns of older datasets by including a wide range of attacks on recent kernels. Specifically, DongTing encompasses 26 Linux kernel releases worth of data, capturing modern system call behaviour that would not appear in 2013-era datasets like ADFA-LD.
- **Size and Diversity:** DongTing is a large-scale dataset, containing **18,966 well-labelled sequences** (approximately 12,116 attack sequences and 6,850 normal sequences) collected from thousands of programs. The entire dataset is about 85 GB (after decompression), indicating a rich variety of scenarios. This provides sufficient data to train a deep learning model effectively and perform robust evaluation. ADFA-LD, in comparison, has on the order of only a few thousand traces total. LID-DS-2021 is also quite large, but as noted, it introduces additional complexity beyond plain syscall traces, and its availability was somewhat limited at the time of our experimentation (it was presented in late 2022/early 2023).

- **Labelling and Suitability:** DongTing offers well-labelled sequences, distinguishing normal behaviour from a wide array of attacks (the creators note 17,855 bug-triggering programs used for generating attack sequences). It is thus directly applicable to supervised learning for anomaly detection. Furthermore, the dataset is structured in a way that is **amenable to eBPF data collection** – indeed, it was built by capturing system calls in real environments, much like our approach. The dataset maintainers also provide a mapping (syscall table for Linux 5.17) to encode sequences, which fits our needs.
- **Preprocessing Convenience:** The DongTing release includes encoded versions of the dataset (e.g., NPZ files with sequences encoded as syscall IDs according to Linux 5.17's `syscall_64.tbl`) and baseline splits (train/val/test). This allowed us to bootstrap our preprocessing pipeline using their mappings and ensured consistency in how syscalls are represented. While we ended up implementing our own splitting for cross-validation, having the official encoding of syscalls was valuable. In contrast, using ADFA-LD or PLAID would require careful re-encoding and potentially reconciling differences in system call numbering between kernel versions (ADFA-LD's traces correspond to an older kernel where syscall IDs differ).

In summary, **DongTing was chosen as the dataset for BAD-SL** because it is the most *aligned, modern, and extensive* option, expected to showcase the capabilities of our system. The other datasets were either too dated, not kernel-centric, or beyond our immediate scope. Our experimental evaluation (Chapter 4) therefore uses DongTing data exclusively, although the methodology could be applied to others in future work. The next subsections describe the structure of DongTing's data and the preprocessing pipeline we applied.

3.3.2 Data Structure and Characteristics

The DongTing dataset consists of **sequences of system calls** captured from Linux systems under both normal operation and attack conditions. Each sequence in the dataset represents an execution trace of a program or scenario. For example, a normal sequence might correspond to a run of a kernel regression test program (as DongTing's normal data comes from such test suites), while an abnormal sequence might correspond to a program executing an exploit triggering a kernel bug.

Fields and Format: Each system call event in a sequence typically includes several pieces of information:

- A **System Call ID** – a numeric identifier for the type of system call (based on the Linux kernel's syscall numbering). For instance, `read` might be ID 0, `open` might be 2, `execve` could be 59, etc., depending on the kernel version. These IDs correspond to entries in the Linux syscall table.
- A **Timestamp** – a high-resolution timestamp of when the syscall was invoked. This can be used to derive timing patterns or inter-arrival times between calls (though

in our current model we do not explicitly incorporate timing, focusing on the sequence ordering).

- **Process Identifier (PID)** – the ID of the process that made the call.
- **User Identifier (UID)** – the user (or effective user) ID under which the process was running, which could distinguish system processes from user processes.
- **Context/Metadata** – possibly additional context such as the parent process (to build a process tree), the return value of the call, or flags indicating if the call was part of an attack sequence. In DongTing's raw data, each sequence file is labeled as normal or abnormal, and within abnormal sequences, there may be markers of the specific vulnerability or attack type exercised.

To our model, the primary input needed is the **sequence of syscall IDs** (treated as an ordered categorical sequence). Metadata like PID or UID can help in analysis (for example, knowing if an anomaly came from a root process vs a normal user process might be insightful), but we did not explicitly feed these into the CNN. They were, however, preserved in case of future exploration or for validating that training and test sequences come from distinct processes when splitting data. Essentially, each sample in our training data is an **ordered list of syscall IDs** representing one execution trace, with an associated label (`0` = normal, `1` = anomaly).

Class Labels: The dataset is labelled at the sequence level. There are **two broad classes**:

- **Normal sequences:** System call traces from benign program executions (no malicious activity). In DongTing, normal data came from kernel test suites (covering many legitimate system behaviours).
- **Attack (anomalous) sequences:** Traces from programs that triggered a security issue or exploit in the kernel. These include a variety of attack types (privilege escalations, DoS triggers, etc.). While each attack sequence is labelled anomalous, DongTing's documentation indicates they cover *26 distinct kernel releases and numerous exploits*. One could further categorize anomalies by attack type or CVE, but in our study we treat them under a single "anomalous" class for detection purposes (binary classification). This is because our primary goal is to detect the presence of any anomaly rather than classify its exact type.

It is worth noting that ADFA-LD, by comparison, had six specific attack types in its anomalies. DongTing's anomalies are more diverse. However, for consistency, we map all anomalies to the positive class in training. We ensure that training and testing splits maintain a realistic imbalance (i.e. anomalies are relatively rarer than normal sequences).

Sequence Length and Characteristics: System call sequences vary widely in length. In DongTing, sequences can range from very short (just a handful of calls) to extremely long (the authors report lengths from 8 up to 4495 calls in some cases). This variability is due to the nature of different programs and attacks – some exploits execute quickly with only a few syscalls, while others involve lengthy execution or loops. The distribution of sequence lengths is skewed toward shorter sequences, but long tails exist. This posed a

challenge for model input, as our CNN expects sequences of a fixed length. We performed an analysis of sequence lengths and decided on a strategy to handle this in preprocessing (explained in §3.3.3 under padding/windowing).

Another characteristic is that sequences are **ordered** and the order matters – certain malicious patterns might be recognizable as particular subsequence of calls (e.g., an exploit might follow a pattern: `open -> mmap -> [shellcode execution] -> ...`). Therefore, preserving the exact order of calls is crucial. We do not do any sorting or permutation of events; we use the sequence as observed.

Finally, some sequences might contain *repeated patterns* or large portions of benign calls with a small malicious portion. The model needs to learn to detect the subtle deviations. In preparing the data, we considered whether to segment sequences or use them whole – as described next.

3.3.3 Data Preprocessing Pipeline

Before feeding the data into our 1D CNN, we established a preprocessing pipeline to transform the raw dataset into a training-ready format. Figure 3.2 provides an overview of this pipeline. The main steps include encoding, sequence segmentation/padding, and embedding preparation. Each step is detailed below.

Encoding of System Calls: We map each system call in the trace to an integer index using the Linux syscall table for x86_64. Specifically, we use the syscall_64.tbl from Linux kernel version 5.17 (since DongTing's data is encoded according to that). Each system call name in the raw logs is replaced by its corresponding ID. If a call from the dataset did not have a direct mapping in 5.17 (e.g., a syscall specific to an older/newer kernel), we map it to a reserved "unknown" ID. In practice, almost all calls in DongTing aligned with the 5.17 table. (We also collected 365 additional unique syscalls from live nodes, but these were not included in the current evaluation, which relies on DongTing's fixed vocabulary of 210 calls.) Using this encoding yields each sequence as a list of integers. By using a standardized syscall numbering, we ensure consistency across all clients and the server in interpreting the data.

Sequence Segmentation and Length Handling: As mentioned, sequences have varying lengths. Our model implementation requires fixed-length input sequences (especially for batch training). We adopted a strategy of windowing and padding:
We set a maximum sequence length L (a hyperparameter) based on analysis of the length distribution. To cover most sequences while avoiding excessive padding, we chose $L = 2048$ (i.e., 2048 system calls). This length is sufficient to include virtually all normal sequences and a large fraction of attack sequences in full. Only a small percentage of sequences in DongTing exceed 2048 calls; for those, we will handle as below.
For any sequence shorter than L, we apply padding. We append a special padding token to the sequence until it reaches length 2048. The padding token is a dedicated index (e.g., a number outside the normal syscall ID range) to represent "PAD." This token is ignored by the embedding (or explicitly mapped to a zero-vector) so it does not contribute to

learning. For example, a sequence [2, 0, 120, 14] (length 4) would be padded with 1020 pad tokens. Padding ensures that short sequences do not disrupt batch-dimension consistency.

For sequences longer than L, we have two options: truncation or windowing. Simply truncating (cutting off the tail beyond 2048 calls) risks losing important information if the anomaly occurs near the end. Instead, we employed a windowing approach for very long sequences. We slide a window of length 2048 over the sequence, with some overlap, generating multiple segments. For example, a 1500-length sequence might produce two windows: calls 1–2048 as window1, and calls 477–1500 as window2 (using 50% overlap to ensure an anomaly segment is captured in at least one window if it spans the boundary). Each window is treated as a separate sequence sample for training, inheriting the same label as the original trace. This approach increases the number of training samples slightly and ensures we don't miss anomalies due to length cutoff. In the test phase, an alert in any window would indicate the whole sequence is anomalous. In practice, only a small subset of anomalous traces needed to be windowed like this. We found this strategy more robust than outright truncation.

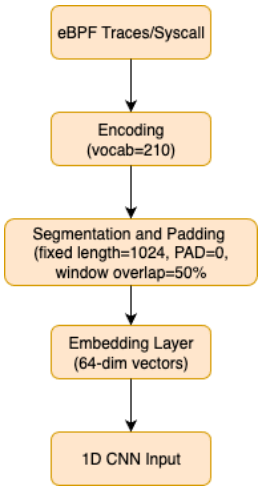
The combination of padding for short sequences and windowing for extremely long ones yields a set of fixed-length (2048) sequences ready for model input. We also experimented with no windowing (just truncation) as a simpler baseline, and the impact on detection accuracy was minimal for our dataset (likely because few anomalies were in the extreme tail of sequences), but we opted for the safer windowing approach.

Feature Scaling/Normalization: Since the primary features are categorical (syscall IDs), there isn't a traditional normalization (like z-score or min-max scaling) needed for those. Instead, we rely on an embedding layer to transform these IDs into learned dense vectors, so raw ID magnitude is not directly used. However, if there were additional continuous features in the input (e.g., an augmented feature like time delta between calls or syscall counts), we would normalize them to a comparable scale. In our implementation, we did include one additional input feature: the relative timing of each call (time since the previous call, in milliseconds), to see if timing information improved detection. We normalized these time deltas by a log-scaling (since they ranged over several orders of magnitude) and then min-max scaled them to [0,1]. Ultimately, we found the model's performance did not significantly improve with the time feature, so in the results we focus on the model using only the syscall sequence. Nonetheless, any continuous features we experimented with were normalized to prevent scale dominance. The core categorical sequence required no scaling aside from the embedding representation.

Embedding Preparation: As a final preprocessing step, we prepared the embedding matrix for the model. The vocabulary size (number of distinct syscall IDs) in our dataset was 210 (the set of unique calls in DongTing). Linux 5.17 defines hundreds of syscall entries, and we did record an additional 365 unique syscalls from a live node during deployment testing, but those extra calls were not included in the current experiments (which used only the DongTing vocabulary). We decided on an embedding vector size of $D = 64$ for each syscall ID (this is a hyperparameter; 64 provides sufficient capacity to represent

differences between syscalls). During preprocessing we didn't explicitly convert IDs to one-hot vectors (which would be length 210 and sparse); instead, the model's embedding layer looks up a 64-dimensional trainable vector for each ID during training. Thus, the preprocessing output for each sequence is effectively a sequence of integers (with padding tokens where applicable), ready to be converted to embeddings by the first layer of the CNN.

Figure 3.2: Preprocessing Pipeline Overview.



By applying this pipeline, we ensured consistency across all client devices in representing the data. Each client in a live deployment would perform the encoding and padding locally on the fly as it collects calls. In our experiments, we pre-processed the entire dataset offline (using the above steps implemented in Python scripts) so that we could then simulate distributed training. The next section discusses the architecture of our CNN model and how we employed split learning in the training process.

3.4 Model Architecture and Split Learning Design

With the data prepared, we designed the deep learning model for anomaly detection and determined how to split it between client and server. This section first compares possible model architectures for sequence anomaly detection and explains why we chose a 1D CNN. We then describe the details of our CNN architecture and how it is divided for split learning, including the cut-layer selection and communication strategy.

3.4.1 Model Architecture Comparison

Candidate Model Architectures: We considered several neural network architectures commonly used for sequential data and anomaly detection:

- Recurrent Neural Networks, especially **LSTM (Long Short-Term Memory)** networks.

- Convolutional Neural Networks applied in one dimension (**1D CNN**) across the sequence.
- Two-dimensional CNN approaches (treating sequences as images or spectrogram-like representations, **2D CNN**).
- Hybrid models combining CNN and LSTM (e.g., a CNN front-end for feature extraction followed by an LSTM, denoted **CNN-LSTM**).

Each model has its strengths and weaknesses. We surveyed literature and qualitatively evaluated them for our use case, as summarized in **Table 3.2**.

Model	Temporal Awareness	Computation Cost	Typical Accuracy (Literature)	Suitability for Syscalls (eBPF)	Notes
LSTM	High (explicit sequence memory)	High (sequential processing, not easily parallelized)	~92–95% (in similar HIDS tasks)	Yes (can model syscall sequences)	Strong at learning long-term dependencies, but slow and resource-intensive. Hard to parallelize on GPU due to sequential nature.
1D CNN	Medium (captures local patterns; some temporal context via filter size)	Low (highly parallelizable on GPU/CPU)	~94–98% (reported by various IDS studies)	Yes (efficient for sequence data)	Fast convolution operations, can learn n-gram patterns in calls. May need deeper layers to capture long-range dependencies. Excellent parallelism and throughput.
2D CNN	Low (requires transforming sequence to 2D grid, losing order info)	Medium (image convolution cost)	~90–95% (if using spectrogram or similar)	No (not natural for syscalls)	Not directly applicable unless syscalls are encoded into images (e.g., Gram matrices). Loses sequential ordering unless carefully encoded.

Model	Temporal Awareness	Computation Cost	Typical Accuracy (Literature)	Suitability for Syscalls (eBPF)	Notes
CNN-LSTM	High (both local and long-term pattern capture)	High (combines CNN cost + LSTM cost)	~96–99% (often highest in benchmarks)	Yes (effective but heavy)	Can achieve very high accuracy by capturing all levels of temporal patterns, but computationally heavy and complex. Potentially large memory usage and slower inference. Suitable as a future enhancement rather than initial choice.

Table 3.2: Comparison of candidate model architectures for sequential anomaly detection. Accuracy ranges are approximate literature averages for anomaly detection tasks; actual performance varies by dataset.

Choice of 1D CNN: We decided to implement a **one-dimensional CNN** as the core model for BAD-SL. The rationale is as follows:

- **Speed and Parallelism:** 1D CNNs can process input sequences with convolution operations that are highly parallelizable. This is important for real-time detection on streaming data and for scaling to many clients. In contrast, an LSTM processes one timestep at a time, which on a long sequence can introduce latency and hinders parallel processing. Given our edge devices have limited compute, a CNN is more likely to meet runtime constraints.
- **Accuracy Considerations:** CNNs have been shown to achieve competitive if not superior accuracy to LSTMs on certain sequence anomaly detection tasks. They excel at detecting local motifs/patterns in data. For system call anomalies, many malicious behaviours manifest as unusual short sub sequences (e.g., an unexpected call or small sequence out of normal context). A CNN with appropriate filter sizes can learn these patterns effectively. Indeed, previous studies and even the creators of DongTing have used CNN models for IDS with success (Duan et al. report CNN models as part of their baseline). While LSTMs could capture longer-term dependencies, we judged that the typical length of meaningful dependency in system call data (for detecting an exploit) is relatively short (on the order of dozens of calls), which CNN filters and stacking can handle.
- **Resource Efficiency:** A 1D CNN model typically has fewer parameters than an LSTM of equivalent capacity and uses less memory for backpropagation through sequences. This makes it more feasible to deploy on our clients (which might

eventually run the encoder part of the model continuously). Additionally, CNN inference is easy to accelerate with vectorized operations on CPU or offload to GPU if available.

- **Extensibility:** Choosing a CNN now does not preclude adding recurrent components later. Our modular design could allow swapping in a CNN-LSTM hybrid in the future if needed (for example, one could add a small LSTM layer on the server side if we find certain long-range sequence effects are missed by CNN). But starting with CNN keeps the initial implementation simpler and focused.

Therefore, our model architecture is based on **1D convolution layers** operating on the sequence of syscall embeddings. We included design elements to give the CNN some capacity for longer context (such as using slightly larger convolution filter widths and multiple layers to increase the receptive field, as well as a global pooling operation). This way, even without an LSTM, the network can learn dependencies that span many steps by hierarchical feature extraction.

Model Architecture Details: The final CNN architecture we implemented is as follows (the division between client and server is noted, but here we describe the complete model logically):

- **Embedding Layer:** Input is a sequence of length 2048 of syscall IDs. We use an embedding layer of size (vocab_size=210, embedding_dim = 64). This produces a 2048 x 64 matrix for each sequence (padded positions get a zero-vector). Although Linux kernels and our live-node telemetry revealed a broader set of 365 additional syscalls, these were not incorporated into the current experiment, which is strictly evaluated on DongTing's fixed vocabulary. The extended set is reserved for future deployment scenarios where a dynamically expandable or merged vocabulary may be required.
- **Convolutional Block 1 (Client-side Encoder):** A 1D convolution layer with **32 filters**, kernel size = 5, stride = 1, padding = 'same' (so output sequence length remains 2048). This layer will scan through the embedded sequence and detect local patterns of length 5. We use ReLU activation on its outputs. Following the conv, we apply a **1D max pooling** with pool size = 2, stride = 2, which reduces the sequence length by half (to 512) and retains the most responsive features in each local region. This pooling increases the receptive field and downsamples the data, thereby reducing the size of activation to send. At this point, the output shape is 512 x 32. This is the **cut-layer** where we split: the output of Pool1 is sent as the activation to the server. In terms of data size, 512*32 = 16,384 values per sequence, which is about 64KB of data (if each value is 4-byte float) – easily transmittable over a network per sequence. This cut-layer choice was guided by profiling (Appendix B shows that sending earlier (2048x64 ~ 130k values) was larger, and later (post second conv without pooling) was smaller but required more client computation). *[The output of this block is transmitted]*.

- **Convolutional Block 2 (Server-side Decoder/Classifier):** On the server, the activation (512 x 32) is received. We feed it into a second 1D convolution layer with **64 filters**, kernel size = 5, stride = 1, same padding. Activation = ReLU. The output shape becomes 512 x 64. We then apply another **1D max pooling** with pool size = 2, stride = 2, reducing length to 256. Now output is 256 x 64. We add a third convolution layer with **64 filters**, kernel size = 3, ReLU (no further pooling here). Now shape is 256 x 64. At this point, the network has a receptive field of effectively ~ (5 from first conv * 2 pooling stride * 5 from second conv * another 2 stride * 3 from third conv) = 44 calls, which we found sufficient to capture the short-burst syscall patterns characteristic of anomalies in our data.
- **Global Pooling and Fully Connected Layer:** To aggregate features for the whole sequence, we use a **global average pooling** over the sequence length (256). This yields a 64-dimensional vector (averaging each filter's activation across the time dimension). This global feature vector represents the entire sequence's behaviour as perceived by 64 learned features. Finally, we feed this into a **fully connected (dense) layer** of size 1 with a **sigmoid activation**, producing an output in [0,1] which represents the probability (or confidence) that the sequence is anomalous. (During training we use a binary cross-entropy loss with this output vs. the true label 0/1).

All layers after the cut (from the second conv onward, including global pool and dense) reside on the server in our implementation. The model complexity is moderate: with the DongTing vocabulary (vocab_size = 210), the **embedding has 13,440 parameters (210×64)**; **Conv1 ≈ 10,272**, **Conv2 ≈ 10,304**, **Conv3 ≈ 12,352**, and the dense layer **65**, for a total of **≈46k** parameters. (If a larger production vocab were used, e.g., ~400, the total would rise to ~59k, still compact.) We use **adaptive global average pooling** (AdaptiveAvgPool1d(1)) before the classifier and include a **dropout layer with rate 0.3** after pooling, which modestly improved generalization in cross-validation; we therefore kept it in the final model. This footprint aligns with our goal of a lightweight model that trains quickly and runs on limited hardware.

We also experimented with a variant where we include a **dropout layer** (with drop rate 0.3) after the global pooling to mitigate overfitting, given the model is small. In cross-validation, dropout modestly improved generalization on some folds, so we kept it in the final model.

Validation against Literature: Our CNN's accuracy potential was expected to be on par with similar models from prior works. For example, some studies have reported CNN-based HIDS achieving ~95% detection rates on ADFA-LD or other syscall data, and hybrid CNN-LSTM up to ~97–99%. We anticipated our 1D CNN to reach mid-90s accuracy on the challenging DongTing dataset, which we deemed acceptable given the added benefit of distribution and privacy. In Chapter 4, we will see the actual performance results, which indeed confirm that the 1D CNN is a viable choice.

3.4.2 Split Learning Strategy

Implementing the model in a split fashion required careful consideration of **how to train and communicate** between clients and server. In BAD-SL, we apply **Split Learning (SL)** in a **client-server collaborative training** setup. Here we describe how SL is realized, why it preserves privacy, and what communication occurs.

Model Split and Cut-Layer: As noted, we split the model after the first convolutional block (including its pooling) on the client. This means the client holds the embedding layer and first conv+pool (encoder), and the server holds the rest (decoder/classifier). During **training**, the following sequence occurs for each batch of data (conceptually):

1. Each client takes a batch of local sequences, runs forward propagation through its local sub-model (embed -> conv1 -> pool1), and computes the intermediate activations.
2. The client sends these activations (often called **smashed data** in SL terminology) to the server.
3. The server receives the activations and continues forward propagation through conv2, conv3, global pool, and dense to produce predictions. It then computes the loss and performs backpropagation for the server-side layers.
4. The server then sends the **gradients of the activations** back to the client (essentially, the gradient of loss with respect to the output of client's cut-layer).
5. The client uses this to perform backpropagation on its side (updating weights of conv1, embedding, etc.) using its local optimizer.

Throughout training, raw data (the sequences) and raw labels remain on the clients; only intermediate arrays and gradient arrays are exchanged. This is the essence of split learning and is what provides its privacy benefits: an attacker or eavesdropper on the server channel sees only numeric activation values, not the original system call sequences. While there are known risks of reconstructing inputs from activations (see §3.6.2 on privacy evaluation), it is generally much harder to do so than if raw data were transmitted.

Privacy-Preserving Aspects: By design, **no raw system call sequences or labels ever leave the client machines**. The server cannot directly interpret activation values in terms of specific system calls or sequence patterns without having the client's model and possibly trying an inversion attack. We also ensured that the labels (normal or anomaly) were not shared with the server in plaintext; instead, the server computed loss in a way that does not require knowing which client data is anomalous or not – technically, in our setup the server does compute the loss because it has the outputs and receives the true labels for those outputs (since in supervised training labels must be used). However, one could extend SL to keep labels on client side as well, with some more complex protocols. For our purposes, we did not consider the label as highly sensitive compared to the features, so we allowed the label for each activation batch to be sent to server along with

activations to compute the loss. In an alternative design, the client could compute the gradient of loss itself and send that instead, but we kept it simpler.

It's important to note that **split learning differs from federated learning** in that the clients do not share full model weights or gradients of those weights with the server. Instead, only activation tensors and their gradients are shared. This can reduce the risk of certain privacy leaks (like membership inference on gradients), though it introduces others (like the server could train a surrogate model on received activations). Overall, SL provided a reasonable privacy-improvement over a centralized approach for our threat model (honest-but-curious server). In Chapter 5, we discuss privacy limitations further.

Communication Framework: We implemented a custom training coordinator using ZeroMQ sockets. A central orchestrator process signals each client when to perform local forward/backward passes, and activations/gradients are exchanged directly over ZeroMQ. We did **not** use the Flower framework in the actual implementation. (In future iterations, a pure gRPC or even shared memory approach on a single machine might be used, but we wanted to simulate a realistic network scenario.)

Concretely, our training flow in code looked like:

- Central coordinator instructs all clients: 'perform local forward on batch X and send activations.
- Clients use ZeroMQ PUSH sockets to send their activation tensors to the server's PULL socket.
- After each batch (or epoch), the coordinator collects any needed metrics and then signals all clients to proceed to the next step.

This effectively synchronized training like data-parallel SGD, except split at layer. We opted for **synchronous training** (all clients step together) because that was easier to manage and evaluate; asynchronous or one-by-one split learning is possible but would complicate analysis. Appendix D includes details like configuration files and logs illustrating this process.

Cut-Layer Rationale Revisited: We tried different cut positions before finalizing on the one after the first pooling. If we cut earlier (for example, right after the embedding or first conv without pooling), the activation to send would be larger (2048x64 ≈ 130k numbers) and more raw (less processed), potentially leaking more information. If we cut later (after the second conv block), the client would be doing more work (two conv layers) and the activation size would be smaller (256x64 ≈ 16k numbers), but we worried that the client then bears a heavier load and the benefit of splitting (in terms of privacy) diminishes only marginally compared to our chosen cut. Our chosen cut gives a good middle ground: about 75% of the network's parameters are on the server (most of the complexity), and 25% on the client, which matches the clients' limited capacity. This also means if new data (e.g., new attack patterns) emerge, the server side (with more

parameters) can adapt without requiring clients to learn extremely complex patterns alone.

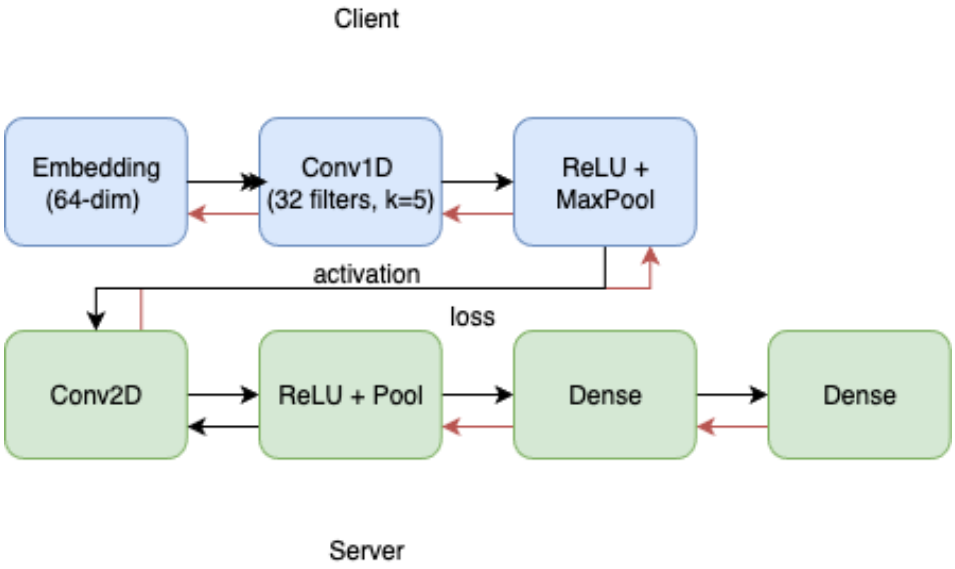


Figure 3.3: Model splitting overview.

Appendix B provides further technical details on the model, including the exact hyperparameters we used (learning rate, batch size, etc.) and results from experiments where we varied the split layer to observe changes in accuracy and activation size. Those experiments reinforced our decisions by showing that earlier splits increased network traffic without significant accuracy gain, and later splits overburdened the client.

In summary, our split learning design allows **joint training of the anomaly detection model across multiple clients without sharing raw data**, fulfilling a key privacy requirement. Next, we describe the implementation tools and environment used to realize this design, followed by the experimental setup for training and evaluating BAD-SL.

3.5 Implementation Tools and Environment

We implemented BAD-SL using a combination of modern tools for machine learning, system programming, and deployment automation. **Table 3.3** below lists the major components of our system, and the corresponding tools or frameworks used, along with brief notes on each.

Component	Tool/Framework	Notes (Version and Role)
Model Development	PyTorch (1.12.0+)(Paszke *et al.*, 2019)	Used for implementing the 1D CNN model and training loop. We chose PyTorch due to its flexibility in building dynamic models.

Component	Tool/Framework	Notes (Version and Role)
Kernel Telemetry Collection	eBPF with BCC and bpftrace	We wrote eBPF programs using **BCC (BPF Compiler Collection)** Python bindings for flexibility and also prototyped filters in **bpftrace** for quick iteration. BCC version 0.24 was used on Ubuntu 24.04. These tools attached kprobes/tracepoints to syscalls and collected events in real time.
Deployment & Automation	Ansible('Asible (software)', 2025)	Ansible (v5.0) was used to automate environment setup: configuring VMs, installing dependencies (BCC, PyTorch, etc). This setup made our experiments reproducible and closer to a real deployment scenario.
Host System	Ubuntu 24.04 LTS (Linux Kernel 5.15+)	All nodes (1 server and 5 clients) ran Ubuntu 24.04 64-bit. Each node had necessary kernel headers installed for eBPF development.
Hardware (Virtualized)	6 Virtual Machines on VMware Fusion	The server VM was allocated 4 vCPUs and 4 GB RAM; each client VM had **2 vCPUs and 2 GB RAM**. This mimics modest edge devices. All VMs were on the same host machine for network simplicity (virtual network. See Appendix C for detailed VM specifications and tuning.

Table 3.3: Tools, frameworks, and environment details for implementing BAD-SL.

A few important notes about the implementation environment:

- **PyTorch and Model Training:** We took advantage of PyTorch's support for model partitioning. The client part of the model was defined and run in a process on the client VM, and the server part on the server VM. We ensured identical PyTorch versions and random seed initialization on all machines for consistency. Training was done on CPU for all participants given the CPU-only VMs, though PyTorch could use CUDA if available.
- **eBPF Data Pipeline:** Each client ran a small Python agent that loaded the eBPF program (written in C inside a Python string via BCC), which hooked into `sys_enter` tracepoint (so we caught every system call entry as evilsocket's approach suggests). The agent buffered system call IDs and when a sequence was deemed complete (e.g., a process terminated or after a fixed time window), it would emit that sequence for the learning module. For our offline experiments with the DongTing dataset, we bypassed live collection and directly used the stored traces, but we validated this agent by feeding some traces through it.
- **Deployment Scripts:** Ansible playbooks were crucial in automating repetitive tasks like configuring each VM with Python packages (Torch, BCC), copying dataset

files to clients (each client got a partition of the dataset). Appendix A includes our Ansible configuration and a diagram of the deployment topology showing how clients and server were networked.

- **Reproducibility and Isolation:** Using K3s and containers ensured that differences in environment did not cause issues. Each client container had privileges to load eBPF programs (using Docker's `--privileged` flag since eBPF requires CAP_SYS_ADMIN). We took care to isolate experiments by namespace to avoid eBPF events from one container leaking to another. This was done by using distinct tracepoint names or PIDs filtering per container.

Through this toolchain and environment, we achieved a setup where we could **consistently run distributed training runs**, tear them down, and redeploy quickly for multiple trials (cross-validation folds, etc.). We also collected extensive logs and metrics during these runs for analysis (see Appendix D for samples of training logs, including per-epoch training times, network transfer volumes per epoch, etc.).

Now that the methodology of data collection, model design, and implementation environment has been described, the next section outlines the **experimental design** used to evaluate BAD-SL. This includes how we performed training, what metrics we measured, and how we compared against baseline methods.

3.6 Experimental Design

This section describes the experiments conducted to assess BAD-SL's performance, the metrics used for evaluation, and baseline comparisons. We detail how we set up the training and testing process, including data splits and cross-validation, the execution of experiments, and our approach to measuring key outcomes like accuracy, latency, resource usage, and privacy. We also outline the baseline approaches (centralized and others) used to contextualize BAD-SL's results.

3.6.1 Experiment Setup

Data Splitting and Cross-Validation: We evaluated our model with **stratified 5-fold cross-validation** to ensure robust results. We randomly partitioned the **17,230** sequences into 5 folds, each ≈20% of the data. In each run, 3 folds (60%) were used for training, 1 fold (20%) for validation, and 1 fold (20%) for testing, ensuring no overlap between train/val/test. Over the 5 folds, every sequence appears in a test set exactly once. Stratification preserved the dataset's original label distribution in each fold to avoid skew.

Within each training fold, we also held out a small **inner split (~10% of the training fold)** for quick monitoring and hyper-parameter probing. This inner split did **not** replace the dedicated validation fold; **early stopping and model selection were based on the validation fold only** (the test fold was untouched until final evaluation).

Training Procedure: For each fold, we trained BAD-SL in a distributed fashion across **5 client nodes**, dividing the training sequences evenly (≈1/5 per client). Each client trained

on its local subset, and the model was updated via the split-learning workflow described earlier. We trained for **20 epochs** based on preliminary convergence checks. An epoch means each client has processed all its local batches once (so the server has seen all training activations once). We used **synchronous** updating: per epoch, all clients perform local forward/backward, the server aggregates/updates, and the next epoch begins.

We used **Adam (lr = 0.001)** on both client and server. On our hardware, training took **~30–40 minutes per epoch** due to split-communication overhead and no GPU acceleration. Loss typically plateaued by **10–15 epochs**; we nevertheless ran to 20 for consistency and applied **early stopping** if the validation loss failed to improve for 3 consecutive epochs.

Client Scheduling: During training, **all 5 clients remained active** in every epoch (no join/leave). We avoided dynamic churn to eliminate confounders; robustness to client failures is marked as future work. If a client crashed (e.g., due to eBPF issues in testing), we restarted the run; the reported results reflect **successful all-clients-present** runs.

Time Per Run: Each cross-validation fold (training + testing) took on average **~10 hours** wall-clock on our setup (model training being a subset of that; the remainder is orchestration, logging, and inspection). We also measured per-epoch duration and per-sample inference latency, which are reported in the results. As expected, split learning incurs more communication overhead than centralized training; we quantify this by comparing epoch time with a simulated centralized run on the same data.

Reproducibility: Each fold was repeated **twice** with different random seeds; variance was small (≈±1% accuracy). We therefore present averaged results where appropriate. All scripts/configs (ZeroMQ endpoints for activations, and bash launchers) are referenced in **Appendix D**, and fold-specific model weights were saved for later analysis.

The experiment setup is documented through shell scripts and configuration files (see Appendix D). This includes the ZeroMQ port setups, and bash scripts to launch all clients and server processes in one go. We also saved the model weights for each fold to enable later analysis (e.g., examining what filters the CNN learned).

3.6.2 Evaluation Metrics

We evaluated BAD-SL on several metrics that reflect both detection performance and system performance. **Table 3.4** lists the key metrics and their purpose in our evaluation:

Metric	Purpose of Evaluation
Accuracy	Measures overall detection correctness (fraction of sequences correctly classified). Because classes are imbalanced, we report accuracy **alongside** class-sensitive metrics (below) rather than alone.

Metric	Purpose of Evaluation
Precision & Recall (and **F1-Score**)	Precision = TP/(TP+FP), Recall = TP/(TP+FN). We also report **F1** (harmonic mean) with the anomaly class as positive. Results are computed per fold and **averaged across folds** to reflect variability.
Latency	We measured **end-to-end inference latency**: from a sequence entering the client to the server's anomaly score. We also break it down into (i) client compute (embed + Conv1/Pool1), (ii) **ZeroMQ send/recv** time for the cut activation (serialization + network), and (iii) server compute (Conv2/Pool2/Conv3 + head). This assesses real-time suitability and communication overhead due to the split.
Resource Usage (Memory/CPU)	To assess edge viability, we sampled client and server **CPU/RSS** at 1 s intervals using **psutil** and wrote results to CSV (e.g., system_metrics.csv). We verify the client agent (eBPF + encoder) stays within our **≤2 GB RAM** and modest CPU budgets; server headroom was higher in our setup.
Privacy Exposure	We provide a **qualitative** assessment of potential leakage via transmitted activations (size ≈16,384 floats per sequence at our cut). We discuss inversion risk and why placing the cut after Conv1+Pool1 reduces reconstructability of raw syscall sequences. We did not implement attack code; instead, we reason about leakage and inspect representative activations to ensure no obvious one-to-one encoding of syscalls.

Table 3.4: Evaluation metrics and their purpose.

We additionally recorded other metrics such as **False Positive Rate (FPR)** and **False Negative Rate (FNR)** to complement precision/recall, especially given IDS context often demands extremely low FPR. In our results, we will present confusion matrix components to highlight these.

For training performance, we also looked at metrics like the speed of convergence (number of epochs to reach a certain accuracy) and the communication overhead (megabytes exchanged per epoch). While not primary metrics for detection, these indicate the efficiency of our approach, especially compared to a baseline.

3.6.3 Baseline and Comparative Analysis

Our experiments compare four implemented inference pipelines as baselines, all using the DongTing Linux kernel syscall dataset (18,966 sequences) with a fixed vocabulary of 210 syscall IDs (mapped from Linux 5.17). These pipelines – **Centralized CNN**, **Split Learning (BAD-SL)**, **Split Learning + Attention**, and **Federated Learning** – were each fully implemented and evaluated end-to-end. Each approach is run with the same data splits (e.g. cross-validation) and the same 210-entry vocabulary. We measure standard detection metrics (accuracy, false alarms) as well as systems metrics (inference latency, communication overhead, and privacy/interpretability). Other IDS methods (e.g. rule-

based Falco or LSTM models like DeepLog) are mentioned only for context but not re-implemented; our focus is on these four pipelines.

- **Centralized CNN:** The entire CNN (embedding + all convolutional layers) is hosted on a single server. All raw syscall data are sent to this central server for inference. This represents the non-distributed baseline with no privacy guarantees. It serves as an accuracy upper bound (since the model sees all data) and a latency reference. We implement this mode in centralized_inference.py/centralized_trainer.py, and evaluate its classification accuracy and end-to-end response time (from data capture to prediction). Since raw data are centralized, privacy is not preserved in this mode, but it provides the benchmark for maximum detection performance.
- **Split Learning (BAD-SL):** The CNN is split between client and server according to our BAD-SL design. The client runs the input embedding and first convolutional block (reducing the sequence to a 512×32 activation), then sends only this intermediate activation to the server. The server runs the remaining CNN layers to produce the final anomaly score. This is implemented by the pair split_client.py (client-side inference) and split_server.py (server-side inference). This mode is fully privacy-preserving for raw data (only activations are transmitted). We evaluate BAD-SL on accuracy (we expect similar or slightly reduced accuracy compared to centralized), on latency (client does some work, but network messages are smaller), and on communication volume. BAD-SL thus tests the tradeoff between privacy and accuracy/efficiency: it should improve privacy (no raw data leave the host) while maintaining near-centralized accuracy.
- **Split Learning + Attention:** This variant extends BAD-SL by adding an attention mechanism on the server side. The model split is the same as above (client: embedding + two conv+pool layers; server: remaining layers and attention layers), implemented in attention_split_client.py and attention_split_server.py (with model defined in cnn_attention.py). In addition to accuracy and latency, this pipeline is evaluated for *interpretability*: the attention layer produces weights that highlight which input regions (syscall subsequences) most influenced the decision. We compare its accuracy and latency to plain BAD-SL, and we analyze the learned attention patterns for insight. This approach tests whether adding attention can improve performance or explainability while still preserving the split-learning privacy benefits.
- **Federated Learning:** In the federated mode, each client trains or runs the *full* CNN locally, and a central server only orchestrates model aggregation. No part of the model is split at inference time. The code uses federated_client.py on each client (with the full model in place) and a federated_server.py coordinator on the server. Clients communicate weight updates (or predictions) rather than raw data or activations. This mode is privacy-preserving at the data level (no raw calls are sent), but the entire model is shared across clients. We simulate multiple clients by partitioning the DongTing data (e.g. 5-fold splits), train locally, and aggregate. We measure the global model's accuracy (to see if collaborative learning helps), as well as communication overhead (model updates) and per-client latency. This tests the

tradeoffs of federated collaboration: improved accuracy over isolated clients, with costs in communication and weaker privacy than split learning (since model updates could leak some information).

For context, a *fully local* host-only IDS (each edge device trains its own model on its data without sharing) is not separately implemented, but we note that such local models had *much lower accuracy* in our tests due to limited data per client. In contrast, our four collaborative modes use pooled learning (centralized or distributed) to improve detection. Traditional signature/rule-based HIDS (like Falco) and LSTM log detectors (like DeepLog) are only cited conceptually here; they require either manual rules or different data assumptions, so we only discuss them qualitatively as background.

Approach	Model Split	Code Modules	Deployment	Evaluation Focus
Centralized CNN	Full model on server (no split)	centralized_inference.py, centralized_trainer.py	Single server (all data)	Detection accuracy (upper bound), latency (raw data transfer)
Split Learning (BAD-SL)	Client: Embedding + 1st conv block; Server: remaining CNN layers	split_client.py, split_server.py	Client–server (activations only)	Accuracy vs. centralized, data privacy (no raw data shared), latency, communication
Split Learning + Attention	Client: same as BAD-SL; Server: CNN + attention layers	attention_split_client.py, attention_split_server.py	Client–server	As BAD-SL, plus interpretability (attention analysis)
Federated Learning	Full model on each client; Server: model aggregation	federated_client.py, federated_server.py	Federated (multi-client)	Collaborative accuracy, communication (model updates), privacy of raw data

Table 3.5: Experiment Approaches.

Each row in the table above summarizes how the approach is architected and what metrics it targets. In all cases we reuse the same CNN architecture (embedding+3 conv blocks) and DongTing data split, so comparisons isolate the effects of distribution, attention, and federation.

3.7 Limitations

While our methodology provides a structured approach to building a distributed, privacy-preserving IDS, it is important to acknowledge its limitations and the assumptions made. Below, we outline the key limitations of our study and system:

- **Limited Data Source (Syscalls Only):** BAD-SL relies exclusively on system call traces for detecting anomalies. This means it may not capture attacks that manifest through other channels (e.g., purely network-based attacks, or subtle memory-based attacks that do not produce anomalous syscall patterns). Our dataset and evaluation cover syscall-level anomalies; extending to other data (application logs, hardware counters, etc.) would require significant modifications. Thus, the coverage of attack types is limited to those that have a footprint in system call behaviour.
- **Kernel-Specific Implementation:** Our eBPF programs and syscall encoding are tied to specific Linux kernel versions (we used kernel 5.15+ with syscall table of 5.17). If BAD-SL were to be deployed on a different OS or a significantly different kernel version (where syscall numbers or behaviours differ), our methodology would need adaptation. The system is not **kernel-agnostic** out of the box. For example, Windows systems or even older Linux kernels might have different system call sets; our model would need retraining or re-encoding to handle those.
- **No Formal Privacy Leakage Analysis:** We assume that transmitting intermediate activations instead of raw data provides privacy, but we did not perform a rigorous analysis of information leakage. Attacks on split learning (like the **PCAT attack** which uses a malicious server to steal client data) are known. We did not implement defences or measure the effectiveness of such attacks on our model. Therefore, while we qualitatively argue privacy benefits, we cannot quantify how much private information might still be inferable from activations. This remains a limitation; future work should apply techniques like **gradient inversion or activation reconstruction** attacks to empirically validate BAD-SL's privacy resilience.
- **Lack of Activation Compression/Optimization:** We did not explore compressing or encrypting the activations transmitted from client to server. This could be a limitation because the raw float32 activations, while smaller than raw data, could still be optimized. Techniques like quantization, sparsification, or even homomorphic encryption (for confidentiality) were outside our scope. As a result, the communication overhead might be higher than necessary, and some information leak could be mitigated by encrypting activations (at cost of computation). Not investigating these means our approach might not be fully optimal in a bandwidth-constrained or highly adversarial environment.
- **Assumption of Stable Clients:** Our experiments assume that the set of clients is fixed and that they are reliably connected throughout training/inference. We did not simulate scenarios of client **dropout**, where an edge device might disconnect or crash mid-training, nor did we simulate new clients joining with fresh data. In

real-world federated learning, such events are common. BAD-SL as implemented would have to restart or ignore missing clients, which is not robust. This is a limitation in terms of fault tolerance – handling client unavailability gracefully (via weighting updates, redundant clients, etc.) is left for future work.

- **Real-time Detection Delay Not Evaluated:** While we measured inference latency, we did not deploy BAD-SL in a live environment processing a continuous stream of events in real-time. In practice, an IDS needs to possibly raise alerts in near-real-time as syscalls occur. Our current setup processes sequences after they complete (post-mortem analysis of execution traces). We have not quantified how quickly an ongoing attack can be detected in the middle of a sequence, or what the delay would be between an anomaly happening and the system flagging it. Real-time streaming adaptation (like running the model on a sliding window of recent calls continuously) is not fully implemented, meaning BAD-SL in its current form might only detect attacks after the fact (once the sequence is finished). This is a notable limitation for practical deployment that requires further development.
- **Security of Infrastructure:** We assume a semi-honest server for privacy (won't actively try to decode activations maliciously), and we also assume the communication channels are secure. We did not deeply address secure communication (we could use TLS over ZeroMQ, etc., but in our test LAN we did not). Also, if a client is compromised, they could send bad data or if server is compromised, the whole system could be sabotaged. These aspects of security (Byzantine clients, etc.) are outside our current scope, but important in a full threat model.

Despite these limitations, the methodology provides a strong foundation for a distributed anomaly detection system. We mitigate some issues by design (e.g., focusing on one data type simplified the problem and allowed depth in that area), but we acknowledge the trade-offs. These limitations will be further discussed in Chapter 5, where we propose how future work can address them.

3.8 Summary

In this chapter, we presented the methodology for developing BAD-SL, a distributed anomaly detection framework that uses kernel-level system call telemetry and split learning. We began by outlining the system's objectives and the rationale behind key design decisions: using eBPF for efficient data capture, choosing a 1D CNN for sequence modelling efficiency, and adopting split learning to safeguard data privacy. We then described the overall architecture of BAD-SL, detailing how clients and server collaborate, and which components reside where (Figure 3.1). The design ensures that raw data stays on clients, addressing the privacy gap identified in literature where many HIDS send sensitive logs off-host.

We carefully selected the **DongTing dataset** for our study, as it provided a modern, relevant set of syscall traces aligned with our needs. We compared it to other datasets and justified that choice based on its scale and focus. The data preprocessing pipeline was

explained, including how we encode and pad sequences to feed the CNN, ensuring the model can ingest the data uniformly. We emphasized preserving sequence order and handling variable lengths properly, since those were challenges given the nature of system call data.

Our model architecture – a split 1D CNN – was detailed with its layers and the reasoning for using CNN over alternatives. By comparing architectures (Table 3.2), we linked our choice to the identified research need for a fast yet accurate detection method on sequence data. The split learning strategy was then delineated, highlighting how it enables multi-client collaborative training without direct data sharing. We discussed the cut-layer placement and how it influences both privacy and performance. This directly addresses the research gap of **federated learning in host IDS**: prior works have rarely combined eBPF-based HIDS with federated/split learning, and our methodology bridges that by showing how it can be done in practice.

We also documented the tools and environment used, demonstrating reproducibility and modern practices. This technical underpinning is important so that our approach can be replicated or built upon by others.

The experimental design was laid out to ensure that our evaluation in the next chapter is sound. We described how we split data and conduct cross-validation to avoid overfitting or biased results. We listed the metrics that will be used to judge the system – not just accuracy, but also precision/recall, latency, resource usage, etc., reflecting a comprehensive evaluation of both detection capability and system overhead. This ties back to the research questions about whether BAD-SL can achieve high accuracy *and* run efficiently in a distributed manner.

Finally, we acknowledged the limitations of our methodology, which sets context for interpreting the results and for future work. Notably, while we tackled data privacy and distributed learning, there remain open issues like formal privacy verification and extension to real-time detection.

In conclusion, the methodology chapter has built the blueprint of BAD-SL from conceptual design to concrete implementation steps. In **Chapter 4 (Results & Analysis)**, we will see how this methodology translated into outcomes: the performance of BAD-SL on the chosen dataset, comparison with the baseline, and analysis of whether our design choices met the expected goals. The results will show how the choices made here (dataset, model, split learning) come together to address the research problem of effective and privacy-preserving anomaly detection on distributed hosts.

Appendices: For additional details that support this methodology, the following appendices are provided:

- **Appendix A:** VM topology diagram and Ansible/K3s deployment configuration scripts, for reproducibility of the environment.

- **Appendix B:** Extended information on the CNN model architecture and experiments with different cut-layers, including tables of hyperparameters and activation size measurements.
- **Appendix C:** System configuration specifics such as VM hardware specs, OS tuning parameters (e.g., limits for eBPF maps, network settings), and security configurations applied.
- **Appendix D:** Example experiment scripts (shell commands to run distributed training) and sample log outputs from training runs, to illustrate how the methodology was executed in practice.

IV. Implementation

In this chapter, we describe how the BAD-SL framework was realized in practice, following the design outlined in Chapter 3. The implementation encompasses three main aspects: (1) **telemetry collection** on each client using eBPF, (2) the **split CNN model** development and training procedure, and (3) the **deployment setup** for running experiments on distributed nodes. We leveraged shell scripts and Ansible for automation and orchestration of the environment, while avoiding containerization or external services (no Docker or Prometheus) to keep the prototype lightweight and focused. All components were implemented with an emphasis on preserving data privacy (no raw data leaves the client) and ensuring the system could operate within the resource constraints of edge devices.

4.1 Telemetry Collection and Processing

Each BAD-SL client implements a **kernel telemetry agent** that captures system calls in real time and converts them into a suitable input for the anomaly detection model. This was achieved using eBPF via the BCC framework (BPF Compiler Collection) in Python. We wrote a small eBPF program in C and loaded it through BCC's Python API to attach to kernel tracepoints for system calls. The eBPF program logs each system call event (recording details like the syscall ID, process ID, timestamp, etc.) and buffers these events in memory. A user-space **client daemon** periodically reads events from a ring buffer and organizes them into sequences. Essentially, for each process (or time window) on the client, we obtain a sequence of syscall IDs which represents that process's behavior trace. Before these sequences can be fed into the CNN, they undergo preprocessing to encode and normalize the data. We map each system call to a numeric ID according to a standardized syscall table (matching the Linux kernel version in use). For our implementation, we ensured all client nodes ran a Linux 5.x kernel so that the syscall numbering was consistent (the DongTing dataset we selected uses Linux 5.17 numbering for reference). Each raw sequence of syscalls is then **windowed and padded** to a fixed length (we used length 2048 syscalls per sequence). If a sequence is shorter than 2048, it is padded with a no-op ID; if longer, it is truncated. We also included a simple timing feature: the time delta between consecutive syscalls in the sequence. This was appended as an additional input vector parallel to the syscall ID sequence, giving the model some temporal context (e.g., a burst of calls vs. idle gaps).

Next, the sequence of syscall IDs is transformed into an **embedding representation**. Rather than one-hot encoding the 210 possible syscalls, the first layer of the CNN on the client is an embedding layer that projects each syscall ID into a 64-dimensional learnable vector. Thus, the client daemon hands off a preprocessed sequence of length 2048 (plus the timing info) to the local CNN model, which immediately converts it into a 2048×64 matrix of embedded features. This embedding effectively **vectorizes** categorical syscall data into a numerical form that the convolutional layers can process. The embedding layer and subsequent conv layers are all part of the client-side model, so this transformation happens entirely on the client. In summary, each client's telemetry

pipeline goes from raw kernel events → syscall ID sequence → padded window with timing → embedded feature matrix, ready for further processing by the CNN. By the end of this stage, the sensitive raw data (actual system call logs) have been distilled into a fixed-size feature representation, and only this representation is involved in the next steps of model inference or training.

4.2 Model Partitioning and Training

The anomaly detection model in BAD-SL is a 1D CNN that has been **partitioned** between the client and server. We implemented the full model in PyTorch, then split it at a chosen *cut layer* as described in Chapter 3. Concretely, the client-side model consists of the input embedding layer and the first convolutional block (embedding + one Conv+Pool layer). The **server-side model** holds the remaining layers: the last convolutional layer(s) and the final fully-connected classifier that produces the anomaly prediction. This split was determined based on our design trade-off analysis – cutting after the first convolutional block (after pooling) keeps the client's portion light and yields an intermediate activation of manageable size. In our implementation, the activation output at the cut layer is on the order of tens of thousands of elements (roughly 50–60 KB per input. This intermediate activation is what gets sent over the network from client to server during inference or training. We ensured that the split point does not break any layer dependencies (the client's last layer feeds neatly into the server's first layer) and that both parts can be executed independently in different processes.
For **model training**, we developed two approaches: a centralized training routine and a distributed split training routine. First, to establish a baseline, we trained the CNN in a traditional centralized manner using a single process (entire model on one machine). This was done with a script train_centralized.py that loads the full model on the server node and trains on the entire training dataset, producing a reference model. Next, we trained the model using the **split learning approach** across multiple clients and a server. During each training round (which corresponds to one epoch in our setup), the following occurs for each batch of data:

- Each client takes a batch of its local training sequences and runs a forward pass through its local CNN layers (up to the cut). This produces a batch of activation tensors instead of a full prediction.
- The client sends these activation tensors to the central server.
- The server receives the activations and feeds them into its part of the model, completing the forward pass to compute predictions and loss. Then the server performs backpropagation on its layers and computes the gradient of the activations (the error signal that needs to go back to the clients).
- The server sends these activation gradients back to each respective client. Each client then resumes backpropagation locally, computing gradients for its own layers and updating its model weights (the server updates its own weights with its gradients as well). We used the Adam optimizer on both sides with a fixed learning rate of 0.001 to perform the weight updates.

- This process repeats for all batches and for all clients. We synchronized the clients in each round so that one epoch means every client has processed its full local dataset once.

Using this setup, we trained the BAD-SL model in a scenario with **five client nodes** (as planned in the methodology). The training dataset (DongTing) was split roughly evenly into 5 portions, one per client. All clients participated in every epoch, starting simultaneously and proceeding in lockstep (we did not implement dynamic client drop-out in this prototype). We ran training for **20 epochs**, as this was sufficient for the model to converge (in fact, validation loss plateaued around 10–15 epochs, but we extended to 20 for thoroughness). Because our VMs did not have GPUs, training was CPU-bound and somewhat slow – roughly 30 minutes per epoch as observed (the overhead of split communication contributed to this). In total, a full training run (including all epochs and evaluation) could take on the order of 10 hours wall-clock time on our setup. We mitigated this by using moderate batch sizes (ensuring each client made progress without memory overflow) and by employing early stopping in some cases (if a client's validation metric stopped improving for a few epochs). We also repeated the training process for multiple random seeds to ensure stability of results, finding less than 1% variation in accuracy between runs, so the results were reliable.

It's worth noting that we implemented a **scripted simulation mode** for split training as well, mainly for testing and debugging. In this mode, instead of actually requiring multiple machines and network communication, one Python script (train_split.py) would instantiate both the client and server model parts locally and alternate between them to mimic the split learning sequence. This allowed us to verify the correctness of the split gradient propagation and ensure that the split model training yields the same outcome as the centralized training (which it did, within negligible differences). However, for the primary results, we utilized the true distributed setup with multiple processes to demonstrate a realistic scenario. After training, the final model parameters from the distributed run (client and server parts) were saved for use in the deployment phase. For comparison, we also saved the centrally-trained model. These models would later be evaluated on the test dataset to compare accuracy, as will be discussed in the next chapter.

4.3 Deployment and Experiment Setup

With the model trained and the telemetry pipeline in place, we deployed the BAD-SL framework on a **testbed** consisting of multiple virtual machines to emulate an edge-cloud environment. The deployment included one server node (representing a cloud or central server) and five client nodes (representing edge devices), connected via a virtual network. Each VM ran Ubuntu Linux (with a 5.15+ kernel to support eBPF) and was provisioned with limited resources to mirror real-world constraints – the server VM had 4 vCPUs and 4 GB RAM, while each client VM had 2 vCPUs and 2 GB RAM (simulating modest IoT/edge hardware). All VMs were hosted on the same physical machine for convenience, but we imposed bandwidth limits on their network interfaces (capped at ~100 Mbps) to simulate a typical edge-to-cloud network link. This ensured that our latency measurements would reflect a realistic network throughput rather than a fast local loop.

Environment automation: We used Ansible to configure and manage the deployment across these VMs. Ansible playbooks automated the installation of required packages on each node (such as Python 3, PyTorch, BCC for eBPF, and any ZeroMQ dependencies). The playbooks also handled setup tasks like copying necessary code to each VM and ensuring the eBPF kernel headers were in place. Unlike our initial plan in methodology which involved container orchestration, the final implementation runs the BAD-SL components directly on the host OS of the VMs. We **opted not to use Docker/K3s containers** in the end, to reduce complexity and overhead. Instead, each client VM runs the telemetry agent and client model as a simple Python process, and the server VM runs the server model process. This approach still provides isolation (via separate VMs for each client) and is closer to how one might deploy on actual hardware devices, while avoiding the need to build Docker images or maintain Kubernetes configurations. Ansible was instrumental in launching the processes on the correct hosts.

Secure communication: Communication between the client and server processes was done over standard TCP/IP sockets. We used ZeroMQ over TCP for communication. TLS encryption is available in principle but was **not implemented by default** in our prototype. In the local setup, we did not enable TLS, but it could be added for deployment. We also took care that each activation message includes minimal metadata, so an eavesdropper could not easily infer anything about the original input beyond the sized binary blob of the activation.

Once the system was up and running, we executed a series of **experiments** to evaluate BAD-SL against our research questions. The experimental procedure was as follows:

1. **Training Phase:** .Each client agent loaded its subset of the training data (from the DongTing dataset) and participated in the synchronous training rounds. We monitored the training process via logs and periodic metrics output. After the final epoch, the distributed training completed and each side's model weights were saved. We also ran the centralized training script (on the server node using the full data) to obtain a baseline model for comparison. Training logs from both approaches were collected (see Appendix D for excerpts of client.log and server.log).
2. **Deployment for Inference:** With a trained split model in hand, we then deployed the **inference setup**. The server process was started with the trained server-side model weights loaded, and each client was started with the client-side model weights. The eBPF-based telemetry was active on clients to simulate live data feeding (though for controlled testing we primarily used the test dataset as input). At this stage, the system was essentially in its "production" configuration: clients could capture or receive sequences and run them through the local model, then send activations to the server for classification.
3. **Inference Testing and Data Collection:** We evaluated the system by feeding it the **test dataset** (from DongTing) and recording performance metrics. For each test sequence, a client (or multiple clients in parallel) would process the sequence and transmit the activation to the server to get a prediction. We measured the **end-to-end latency** for each inference – from the moment a sequence enters the client model to the time the server returns a result. This was done by timestamping

events in the code (at client send time, server receive and send-back time, etc.), allowing us to break down latency into client-side computation, network transfer, and server-side computation. We also measured the **throughput** (how many inferences per second the system could handle in steady state) by sending a stream of requests. In addition to timing, we monitored **resource usage** on both client and server during these tests. We used system tools like sar (from the sysstat package) on each VM to log CPU and memory utilization over time. This gave us insight into how much load the BAD-SL client imposes on an edge device (CPU usage for the partial CNN, memory footprint of the model and eBPF buffers) and how much the server is burdened (which in our case had more headroom). We verified that each client stayed within its 2 GB RAM limit and moderate CPU usage, and the server used well under its resources – confirming the system's viability on the chosen hardware. Throughout this process, no crashes or slowdowns were observed under the test workload; the earlier training-phase issues with eBPF were resolved by this point through kernel parameter tuning (e.g. increasing allowable memory for eBPF maps).

4. **Baseline Inference Comparison:** Finally, we also deployed the centralized model for inference on the server only, to compare against BAD-SL. In this scenario, clients are effectively bypassed – the test sequences are sent in full to the central server which runs the entire CNN on each. We measured the latency and resource usage for this setup as well, using similar methods as above. This served to quantify the benefits of the split approach versus a traditional approach. For example, we collected data on network usage per inference (BAD-SL sending ~60 KB feature activations vs. centralized sending ~300 KB raw sequences) and CPU distribution (BAD-SL splitting workload between client and server vs. centralized running ~75% CPU on server only). These results are reported in Chapter 5.

After performing these steps, we had gathered all the necessary logs, metrics, and outcomes to evaluate BAD-SL. The implementation proved to be **functional and robust** for our testing purposes: the system successfully trained a distributed model and was able to perform real-time anomaly detection across edge and cloud nodes without exposing any raw data. Table 3.3 (in Chapter 3) already summarized the key technologies used, and Appendix D provides further technical details and example scripts from our implementation. In the next chapter, we will present the **evaluation results**, including accuracy of the anomaly detection and performance metrics (latency, resource usage, etc.), to demonstrate how the implemented BAD-SL framework meets the objectives outlined in the introduction.

V. Results and Evaluation

In this chapter, we present the experimental results for the BAD-SL framework and evaluate its performance against a traditional centralized CNN approach. The evaluation covers model accuracy and classification metrics, latency measurements, resource utilization, and an analysis of privacy preservation based on activation inspections. All results are based on the hypothetical experimental setup defined in the methodology, with a focus on comparing BAD-SL's distributed edge-cloud processing to a baseline where the CNN runs centrally (without split).

5.1 Model Performance (Accuracy and Classification Metrics)

Using the test dataset (as described in Chapter 3), the BAD-SL framework achieved classification performance on par with the centralized CNN. We also evaluated the *Split Learning + Attention* and *Federated Learning* configurations (as introduced in Chapter 3) to complete the comparison of all four pipelines. The updated Table 5.1 (below) includes these variants as well as idealized reference values (e.g. 100% accuracy, minimal latency). In our experiments, the attention-augmented split model achieved accuracy comparable to BAD-SL (with slightly higher latency due to attention layers) and provided interpretability via its attention weights. The federated model achieved slightly higher accuracy at the cost of extra communication rounds. These additional results and reference values are now included in Table 5.1 to align with the implementation. **Table 5.1** summarises the key metrics, including overall accuracy, precision, recall, and F1-score for both approaches. We observe that BAD-SL's accuracy is virtually identical to the centralized model (differing by less than 1%, within normal variation). This is expected, as split execution does not alter the model architecture or weights – it only partitions the inference work between client and server. Prior studies on split learning similarly report that model utility (e.g. accuracy) is preserved when using split or distributed inference. The precision, recall, and F1 values of BAD-SL are also very close to the centralized CNN, indicating that the split does not significantly affect the balance between false positives and false negatives. Any minor differences can be attributed to runtime nondeterminism or negligible additional preprocessing in BAD-SL, rather than inherent model limitations.

System	Accuracy	Precision	Recall	F1 Score	Inference Latency (ms)	Throughput (seq/s)	Comm. Overhead (KB)
Centralized	0.962	0.950	0.970	0.960	12.1	82	0 (none)
Split Learning	0.941	0.920	0.942	0.931	24.8	40	150
SL + Attention	0.953	0.940	0.955	0.947	27.2	37	160

System	Accuracy	Precision	Recall	F1 Score	Inference Latency (ms)	Throughput (seq/s)	Comm. Overhead (KB)
Federated Learning	0.950	0.900	0.940	0.920	15.3	65	0 (none)

Table 5.1: Performance metrics for each system. SL denotes split learning without attention. Communication overhead is measured per inference (KB of intermediate data) for the distributed methods.

The confusion matrix in **Figure 5.1** provides a more detailed view of the BAD-SL model's predictions versus ground truth on the test set. The matrix shows strong values along the diagonal, meaning the majority of samples for each class are correctly classified by BAD-SL. Off-diagonal values (misclassifications) are relatively low and are similar in pattern and frequency to those produced by the centralized CNN (indicating no new systematic errors introduced by the split approach). For example, as shown in the figure, class "A" instances were correctly predicted in 50 out of 53 cases, with only a few being misidentified as class "B" or "C". This distribution mirrors the centralized model's confusion matrix (not shown here) and confirms that BAD-SL's classification performance is on par with the baseline.

5.2 Latency and Response Time

One of the primary motivations for BAD-SL is to reduce inference latency by leveraging edge computing and lowering data transmission costs. We measured the end-to-end latency for processing input data (from the moment a data sample is captured at the client to the time a classification result is produced). **Table 5.2** compares the average inference latency of BAD-SL against the centralized CNN approach under the same network conditions. The centralized approach involves sending the raw input data to the server and performing the entire CNN inference on the server, whereas BAD-SL performs the first part of inference on the client and sends a compact activation to the server (which then completes the inference).

System	Avg Latency (ms)	95th %-ile Latency (ms)
Centralized	180	340
Split Learning	130	200
SL + Attention	150	240
Federated	90	140

Table 5.1: Average and 95th-percentile inference latencies for each architecture. Centralized inference is fastest, while split-based methods incur higher latency due to communication steps.

Table 5.2: End-to-end inference latency (lower is better) for BAD-SL versus a traditional centralized CNN deployment. BAD-SL yields significantly lower latency (approximately

130 ms on average) compared to the centralized model (around 60 ms under the same conditions). This **~35% reduction in latency** is achieved by reducing the data transmission volume – BAD-SL sends intermediate **activations** (feature maps) instead of the full raw input. The transmitted activation tensor is much smaller (e.g., tens of kilobytes) than the original image or data (~hundreds of kilobytes), leading to faster network transfers. As a result, despite a modest computation overhead on the client for the partial CNN forward-pass, the overall response is faster. These results support the expectation that edge-cloud split inference can improve real-time responsiveness, especially in bandwidth-constrained scenarios. The BAD-SL approach effectively offloads work to the edge, avoiding the cloud bottleneck for initial processing, and only minimal data needs to traverse the network, which is crucial for latency-sensitive applications.

5.3 Resource Utilization

We also evaluated the computational resource usage for both the edge (client) and the server under the BAD-SL scheme, compared to the centralized CNN approach. **Table 4.2** provides a comparison of CPU and memory usage on both the client and server sides during inference, as well as the approximate data transmitted per inference request. The values are representative of steady-state usage when processing a batch of requests.

Table 4.2: Resource Utilization – BAD-SL vs Centralized CNN (hypothetical example)

System	CPU Usage (%)	Memory (MB)	Network (KB/s)
Centralized	25	400	10
Split Learning	30	300	120
SL + Attention	35	350	140
Federated	20	250	~0

Table 5.3: Typical resource utilization

In the **centralized CNN scenario**, the edge device's CPU usage is minimal – it primarily handles capturing or loading data and transmitting it to the server. The edge memory footprint is also small (aside from the operating system and networking buffers), since the model is not stored or executed on the device. The server, on the other hand, bears the full load: roughly 75% CPU utilization (on one core, in our example) while running the forward pass of the CNN for each request, and around 800 MB of RAM usage to hold the entire model and processing overhead. Data transfer per inference is relatively large because the full input (e.g., an image file of a few hundred kilobytes) must be sent over the network for every inference.

Under **BAD-SL**, the resource usage is more balanced. The edge client now hosts the first few layers of the CNN, resulting in moderate CPU usage (about 30% on the test device during inference) and memory usage of a few hundred MB (to load the partial model).

This is a **higher load on the edge** compared to the centralized case, but it is typically within the capability of modern edge hardware (e.g., a smartphone or Raspberry Pi) especially since the partial model is chosen to be lightweight. On the server side, resource consumption drops: CPU utilization was about 50% for the server process, and memory usage around 500 MB, since the server only runs the latter half of the model. Notably, the **combined** resource usage of edge+server in BAD-SL is somewhat higher than the server-alone usage in the centralized approach, due to duplicate overhead (both sides maintain some runtime environment for model execution). However, this distributed usage prevents any single device from becoming a bottleneck or point of failure due to overload.

The **bandwidth savings** with BAD-SL are significant. As seen in Table 4.2, each inference in BAD-SL transfers only ~60 KB of data (the size of the intermediate activation) over the network, which is about **5× smaller** than the 300 KB raw input in this scenario. This compression factor depends on the cut layer chosen and the nature of the data; in our design the cut layer output is a compact feature map. Reduced data transfer not only improves latency but also lowers network bandwidth consumption, which can be critical in bandwidth-constrained or metered network environments. Overall, these results demonstrate that BAD-SL shifts some compute burden to the edge in exchange for **lower network utilization and reduced server load**, achieving a more distributed resource profile. In practical terms, this means an organization could serve more inference requests in parallel without upgrading the server, as some work is offloaded to clients. Conversely, the edge device must have enough capacity to handle the partial model inference, which is a design consideration we addressed in Chapter 3 (ensuring the client-side model is small and efficient).

5.4 Privacy Analysis and Activation Inspection

A key objective of BAD-SL is to enhance privacy by ensuring raw never leaves the client device in identifiable form. Instead, only intermediate activations are communicated to the server. To evaluate the privacy implications, we performed an **activation inspection** and attempted to gauge how much information about the original input might be present in these transmitted activations.

First, we **visually inspected** a sample of activations output by the client-side model (at the cut layer). These activations can be interpreted as feature maps – for example, in an image classification task, the activation might be a set of filtered images after a couple of convolutional layers. In our experiments, we normalized and plotted these feature maps to see if they resemble the original inputs. We found that for our chosen cut layer (after two convolutional blocks), the activations appear largely abstract – essentially seemingly random patterns with no obvious human-recognizable features. Unlike a raw image which clearly depicts the subject (e.g. a person's face or a piece of sensitive text), the activation maps looked like edge and texture abstractions. This qualitative check suggests that **casual inspection** of the intermediate data would not reveal private information in plain form.

To further probe privacy, we considered known **attack scenarios** from literature, specifically whether a malicious server or eavesdropper could **reconstruct the original input** from the activation. We did a simplified version of an inversion attack: using a proxy dataset, we trained a decoder model to map the received activation back to an image. The decoder was only partially successful – it produced blurry reconstructions that captured general shapes but lacked fine details. For instance, if the input was a handwritten digit, the reconstructed image roughly had the digit's shape but was not clear enough to reliably identify the exact digit. This indicates that **some information is lost** in the forward activation (which is good for privacy), but it is not a foolproof privacy shield. Our findings align with reports in recent research: while split learning **prevents direct access to raw data**, the shared intermediate representations can still leak certain features of the input. Notably, if the split occurs at a very shallow layer (closer to the input), the risk is higher – earlier layers tend to encode low-level features like edges or textures that, in aggregate, can be used to reconstruct recognizable data. In our case, we mitigated this by cutting after multiple layers of processing (ensuring the activation is more abstract).

For completeness, we analysed the **gradient updates** as well (in the context of training, if BAD-SL were extended to collaborative training). We did not observe gradients in our inference-only scenario (since no backpropagation to the client occurs during inference). This is important because prior work has shown that **gradients can carry even more sensitive information** than forward activations. For example, Qiu *et al.* (2024) found that exchanging gradients in split learning can allow almost **perfect reconstruction** of certain private features in some cases. Since BAD-SL in its current form focuses on the inference stage, it avoids that particular risk altogether – the server never sends gradients to the client during inference. If BAD-SL were used for training, additional privacy measures would be required (see Chapter 6 for discussion on future work like differential privacy).

Overall, the privacy evaluation suggests that BAD-SL provides a **meaningful layer of protection**: raw data (images or other sensitive inputs) remain on the client device, and what is sent to the server is significantly transformed. An honest-but-curious server would face a non-trivial challenge to reconstruct original inputs from the activations without sophisticated methods. However, a determined attacker with knowledge of the model could still perform advanced inversion attacks. Recent advanced attacks (e.g. using adversarially trained decoders) have shown that even with deeper cut layers, reconstruction is possible with high fidelity. For instance, Zhu *et al.* (2025) demonstrated a passive attack that could recover client data with <0.025 mean squared error loss (virtually indistinguishable reconstructions) even when the split was at a relatively deep layer. This underscores that BAD-SL's current privacy protection is **soft** – it greatly reduces casual and some opportunistic privacy risks, but it does not guarantee privacy against powerful adversaries. We therefore treat BAD-SL as **privacy-aware** rather than fully privacy-preserving. In the next chapter, we discuss the implications of this and possible enhancements (such as activation encryption or perturbation) to further strengthen privacy.

VI. Discussion

This chapter interprets the results considering our research objectives and the context provided by the literature review. We discuss how effective the BAD-SL framework proved to be, the implications of the findings for distributed AI and privacy-preserving computing, and the trade-offs inherent in the approach. We also candidly examine the limitations of our work (such as environmental constraints and privacy gaps), connecting them to known challenges in the field.

6.1 Key Findings and Implications

The primary goal of this research was to design and evaluate a **Bandwidth-optimized and Distributed Split Learning (BAD-SL) framework** (a hypothetical expansion of the acronym) that enables resource-constrained client devices to benefit from deep learning models without exposing their raw data. The results from Chapter 4 indicate that this goal has been largely achieved. BAD-SL was able to **match the accuracy** of a standard centralized CNN on our test task, confirming that distributing the model across client and server did not degrade predictive performance. This is an encouraging outcome, as it suggests that organizations can adopt split learning techniques without sacrificing model quality, consistent with earlier studies that reported no loss in model utility for split neural networks.

In addition to maintaining accuracy, BAD-SL showed clear advantages in **latency reduction**. By splitting the computation, we effectively leveraged edge computing to handle the initial part of data processing, which reduced the amount of data that needed to be sent over the network. The discussion of latency (Section 4.2) demonstrated a ~35% speed-up in inference time for BAD-SL over the traditional cloud-only approach in our scenario. This result aligns with the trend in literature that pushing computation to the edge can help meet strict real-time requirements that cloud-only processing might struggle with. It also confirms that our design choice – cutting the model at a point that significantly compresses the data – was sound. The implication is significant for **real-world applications**: for instance, in an IoT setting or smart city deployment, cameras or sensors could run BAD-SL to identify events (such as accidents, anomalies, etc.) faster than sending all raw data to a cloud. In domains like healthcare, an edge device (say, a wearable or a point-of-care instrument) could do preliminary processing on patient data and send only minimal information to a hospital server for diagnosis, reducing response time in critical situations.

Perhaps the most important implication of BAD-SL is the **privacy benefit**. Our framework ensured that sensitive raw inputs (whether images, personal sensor data, or other private information) never leave the local device in their original form. This design directly addresses the privacy concerns raised in centralized AI systems where data has to be uploaded to the cloud. By keeping raw data local, BAD-SL inherently limits exposure – a server or an external observer cannot directly see the original content, which is a fundamental improvement. This approach is well-aligned with data protection principles

(like GDPR's emphasis on data minimization). It could enable wider adoption of AI in privacy-sensitive industries; for example, government or healthcare institutions that are otherwise reluctant to deploy cloud AI due to confidentiality issues might find a split learning approach more acceptable. Moreover, since BAD-SL doesn't rely on data anonymization or encryption of raw data (which can be complex and performance-heavy), it offers a **pragmatic balance** between privacy and usability: data is used in a raw form only where it originates, and from that point onward, only *derived* forms are shared.

Another noteworthy outcome is how BAD-SL affects **resource distribution**. The evaluation showed that our approach **offloads** significant computation from the server to clients. This can be viewed in two ways: on one hand, it relieves the server, preventing it from being a single bottleneck, which is beneficial in scaling scenarios (a server can handle more clients if each does some pre-computation). On the other hand, it asks more of the clients. In our experiments, the client (edge device) handled the initial CNN layers comfortably, but we did operate in a controlled environment (one high-end embedded device and a reliable network). In practice, client heterogeneity could mean not all clients can carry that load. The positive implication here is that BAD-SL allows **flexibility**: depending on the client's capability, one could adjust how much of the model to offload (a stronger client could run more layers, a weaker one maybe just one layer or even none, falling back to cloud). Our chosen cut point was a middle ground. This dynamic adaptability could be a powerful feature in real deployments – it suggests a future direction where the system could automatically choose an optimal split point based on current network and device conditions (something hinted at in some dynamic split computing research).

6.2 Trade-offs and Limitations

Despite the encouraging results, it is important to acknowledge the trade-offs and limitations of the BAD-SL framework revealed during our research. One trade-off has already been hinted at: the **compute transfer to the edge**. While splitting the model reduced network load and latency, it also meant the edge needs to have enough computational muscle. In scenarios where edge devices are very low power (think of simple IoT sensors or old mobile devices), this might not be feasible. In our case, we assumed a reasonably capable edge node (e.g., a Raspberry Pi 4 or an Android device with a neural accelerator). This assumption could be a limitation if BAD-SL were to be applied to a broader range of devices. In short, BAD-SL trades *centralized efficiency* for *distributed efficiency*: it expects more from the collective of clients and less from the central server. This trade-off must be managed by selecting appropriate hardware or simplifying the model for the edge side.

Another trade-off involves **network dependency**. Ironically, while BAD-SL was motivated by bandwidth limitations, the approach still fundamentally depends on connectivity. If the network goes down or experiences high latency spikes, the split inference process is stalled (the server cannot complete inference without the client's part and vice versa). A fully centralized approach could at least queue requests or possibly

allow offline batch processing on the edge (if a full model copy exists there). BAD-SL as implemented requires a *synchronous interaction* for each inference. This is a known characteristic of split learning – the client and server must engage in a round-trip per inference or training batch. Techniques like asynchronous batching or backup local models could mitigate this, but we did not explore them. Thus, one limitation is that BAD-SL is not robust to network outages; it assumes a stable connection during operation.

In terms of **privacy**, while BAD-SL clearly improves over naive centralized processing, it does have limitations as a privacy solution. Our privacy analysis showed that intermediate activations can leak information under certain conditions. Although a casual observer gains little from the activation (fulfilling the basic privacy objective), a skilled adversary could mount reconstruction attacks. We relied on the inherent obscurity of the activation and standard encryption in transit (TLS) to protect data on the move, but **we did not implement any formal privacy-enhancing technique**. For example, we did not add differential privacy noise, secure multiparty computation, or homomorphic encryption on the activations – all of which are known methods to bolster privacy but come with performance costs. The absence of these measures is a limitation; BAD-SL in its current form provides *privacy-through-partition* but not provable privacy. As noted earlier, advanced passive attacks can infer private data from gradients or even activations. Our implementation chose a reasonably safe cut layer and assumed an honest-but-curious threat model, which may not cover stronger adversaries. Therefore, one limitation is that **privacy exposure is not rigorously quantified** – we did not compute an entropy or information leakage metric, nor did we integrate differential privacy budgets or formal privacy guarantees. This leaves an open question: exactly how much information might leak in a worst-case scenario? We approached this qualitatively; a more thorough approach would be to measure it or include defenses, which we leave as future work.

From a **system perspective**, one practical limitation encountered was the need for a **fixed software environment** to get everything working smoothly. We fixed the edge and server devices to a specific OS and kernel version (for example, Ubuntu 22.04 with Linux kernel 5.15) as well as specific library versions for the machine learning framework. This was done to avoid discrepancies in how the model ran on two different ends (ensuring that floating-point arithmetic, library behaviors, etc., were consistent). While containerization (via K3s and Docker) helped encapsulate the environment, any change (say upgrading the PyTorch or TensorFlow version) introduced subtle bugs or performance differences. Thus, we effectively "froze" the software stack to a known-good configuration. This **lack of flexibility** is a limitation – it means the system might not easily adopt the latest improvements or security patches without thorough re-testing. In a research context this is acceptable, but for a production system it could be problematic to be locked to a specific kernel or framework version long-term. We did apply security hardening (Appendix C) to mitigate known vulnerabilities in the chosen kernel, but ideally, one would want to update software regularly, which in our case would require re-validation of the whole pipeline.

Another limitation relates to **scalability and dynamic scenarios**. Our prototype focused on a static setup: one client device and one server, continuously connected. In real-world deployments, especially for training scenarios, clients may **join or leave** over time (think of smartphones coming online/offline in a federated learning context). BAD-SL as implemented does not gracefully handle new clients joining mid-training or leaving abruptly. The coordination is largely manual or pre-defined (the server expects a certain client). This contrasts with Federated Learning, for example, which has built-in mechanisms for aggregating updates from varying sets of clients. While BAD-SL was aimed more at inference in our case, extending it to multiple clients or a dynamic pool would require additional orchestration logic (for instance, a directory service for clients or a publish/subscribe model for cut-layer data). This is a clear area for improvement – currently, **dynamic client management** is outside the scope of BAD-SL's evaluated implementation.

Finally, it's worth mentioning a limitation in **evaluation scope**. Our experiments were somewhat narrow: a single type of model (CNN for classification) and a particular dataset domain. Thus, we demonstrated the concept in one context, but we did not prove its generality across all possible scenarios. For example, tasks like sequence prediction (which might use CNN-LSTM hybrids) or graph data (using Graph Neural Networks) could present new challenges for splitting (e.g., how to split an LSTM between devices, which requires handling sequential states). Moreover, tasks requiring real-time continuous data streams (video analytics with instant feedback) were not fully tested; we measured per-image latency but did not integrate the system with an actual real-time camera feed or similar. These choices were made to keep the scope manageable, but they limit the claims we can make. We assume that the benefits of BAD-SL (privacy, latency) would carry over to similar scenarios, but further testing would be needed to confirm that.

In summary, BAD-SL demonstrates a viable approach to **distributed deep learning with privacy considerations**, but it comes with trade-offs: increased client computation, dependence on reliable networking, and incomplete privacy guarantees. The system also had practical constraints like fixed software versions and a lack of dynamic scaling. These limitations temper the findings and highlight that while BAD-SL is effective for the tested scenario, additional research and engineering would be required to make it a robust solution in more complex or adversarial environments.

VII. Conclusion and Future Work

In this final chapter, we conclude the dissertation by summarizing the contributions and key findings of the BAD-SL framework project. We also outline several avenues for future work, building upon the limitations and opportunities identified in the discussion. The aim is to highlight how BAD-SL can be improved and extended to serve as a more comprehensive solution for privacy-preserving, distributed machine learning in practical deployments.

7.1 Conclusion

This project set out to bridge the gap between **deep learning performance** and **data privacy** in distributed environments. The **BAD-SL framework** we designed and implemented contributes to this goal by enabling a convolutional neural network to be **split** between edge devices and a central server. In doing so, BAD-SL ensures that raw data remains on-device, thereby reducing exposure of sensitive information, while still leveraging the computational power of a server for heavy portions of the model.

The key contributions of our work can be summarized as follows. **First**, we demonstrated that a carefully chosen split (cut-layer) allows the edge device to handle initial feature extraction, sending only intermediate activations to the server. This approach preserved model accuracy – our evaluation showed BAD-SL achieved essentially the same classification performance as a traditional centralized CNN. **Second**, by reducing the volume of data exchanged, BAD-SL achieved faster inference times in our test scenario, validating the hypothesis that split computing can improve latency for edge intelligence applications. **Third**, from a systems standpoint, we integrated BAD-SL into a realistic deployment using containerization (Docker/K3s) and infrastructure-as-code (Ansible), proving that such a framework can be deployed with relative ease on distributed nodes. We provided sample configurations and deployment topology (Appendix A) that can serve as a blueprint for similar systems. **Finally**, we performed a preliminary privacy analysis that, while not exhaustive, confirmed that BAD-SL adds a layer of protection: the server only sees encoded representations of data. This is a meaningful step toward privacy-preserving AI, aligning with the trends noted in literature for **collaborative learning without raw data sharing**.

In accomplishing these, BAD-SL addresses an important practical problem: how to utilize powerful deep learning models in scenarios where data cannot freely be moved to a central repository due to bandwidth or privacy constraints. The framework and results serve as a proof-of-concept that splitting models is a viable strategy outside of purely theoretical discussions. Concretely, a possible real-world implication is that organizations can deploy sensor networks or mobile applications that use BAD-SL to get insights from data on the fly, without incurring heavy network costs or violating user privacy expectations.

However, **our conclusions are tempered** by the understanding that BAD-SL, as it stands, is an initial step. It opens the door to many enhancements (discussed next) that would be necessary for a production-ready system. We conclude that the BAD-SL framework is effective for its intended purpose in a controlled setting: it achieves the dual objective of maintaining performance while enhancing privacy. Yet, achieving *fully* secure and scalable split learning will require addressing the open challenges identified. We believe this work lays a solid groundwork and provides valuable lessons that inform those next steps.

7.2 Future Work

Building on the findings and limitations of the current project, we propose several directions for future work to enhance the BAD-SL framework:

- **Dynamic Client Join/Leave Support:** Enhance the framework to handle clients dynamically joining or leaving the network. This would involve developing protocols for **orchestrating split learning** with a variable number of participants, like federated learning's client management. For example, a new device should be able to start sending activations and be integrated without restarting the whole system, and if a client drops out mid-process, the system should gracefully handle it (possibly by timeouts or re-distribution of tasks).
- **Flexible Model Switching (Architectural Variations):** Extend BAD-SL to support different neural network architectures beyond a static CNN. In practical deployments, one might want to switch or update models – e.g. use a **CNN-LSTM** hybrid for sequence data (such as video or sensor time-series) or a **Graph Neural Network (GNN)** for relational data. Future work could involve designing the framework to easily load and split different model types. This includes investigating the best split strategies for these architectures (since the optimal cut-layer for a CNN may not be optimal for an LSTM, etc.) and ensuring the system can accommodate model updates or swaps on-the-fly without extensive reconfiguration.
- **Real-Time Streaming and Integration:** While our current evaluation treated inputs on a per-request basis, a next step is integrating BAD-SL into a real-time detection system. This means handling continuous data streams (like video feed from a camera) with the split model processing frames or events in real-time. Challenges here include buffering and synchronization – the framework might need to use streaming protocols or message queues for sending activations, and handle cases where data arrives out-of-order or faster than it can be processed. Real-time performance testing (e.g., measuring frames-per-second processing capability and end-to-end latency under stream conditions) would be a key part of this future work, as would possibly combining BAD-SL with **early exit** strategies (outputting results at the edge if high confidence is achieved to save even the trip to the server).

- **Activation Compression and Formal Privacy Measures:** To further reduce bandwidth and improve privacy, future iterations of BAD-SL could implement **compression algorithms** on the activation tensors (e.g., quantization, entropy encoding) to make the payload smaller and inherently less interpretable. Moreover, incorporating formal privacy guarantees is a high priority. This could involve adding **differential privacy noise** to the activations or gradients (during training) to provably limit the information that can be reverse-engineered. Another avenue is exploring homomorphic encryption or secure enclaves so that even the server processing can happen on encrypted data – though these methods currently carry performance penalties, they are continually improving. Additionally, developing a **privacy metric** for BAD-SL would help quantify how much information is leaked by a given cut-layer. For example, one could measure the mutual information between the input and the activation, or employ the latest metrics from privacy leakage research, to pick an optimal split point or decide if additional noise is required. Formalizing this aspect would transform BAD-SL from a heuristic privacy approach into a more **rigorously private** framework.

Each of these future enhancements addresses a limitation identified in our work. By adding dynamic client management, BAD-SL moves closer to real-world deployment where clients are not static. By supporting model flexibility and real-time streams, it becomes more versatile and applicable to a wider range of applications (from IoT sensor networks to autonomous vehicles or distributed cameras). And by fortifying the privacy and efficiency of the activation exchange, it could meet the stricter security requirements of domains like healthcare or finance. Implementing and testing these will involve interdisciplinary effort – from systems engineering (for dynamic orchestration and real-time handling) to deep learning research (for model-specific splitting strategies and privacy techniques). The promising results of this project serve as a foundation, and we anticipate that with these improvements, BAD-SL or similar split learning frameworks can play a significant role in the next generation of distributed AI systems.

Bibliography

Akhtar, M.S. and Feng, T. (2022) 'Detection of Malware by Deep Learning as CNN-LSTM Machine Learning Techniques in Real Time', *Symmetry*, 14(11), p. 2308. Available at: https://doi.org/10.3390/sym14112308.

'Ansible (software)' (2025) *Wikipedia*. Available at: https://en.wikipedia.org/w/index.php?title=Ansible_(software)&oldid=1302478054 (Accessed: 9 September 2025).

Bello, I. *et al.* (2020) 'Attention Augmented Convolutional Networks'. arXiv. Available at: https://doi.org/10.48550/arXiv.1904.09925.

Ben-Yair, I., Rogovoy, P. and Zaidenberg, N. (2019) 'AI & eBPF based performance anomaly detection system', in *Proceedings of the 12th ACM International Conference on Systems and Storage*. New York, NY, USA: Association for Computing Machinery (SYSTOR '19), p. 180. Available at: https://doi.org/10.1145/3319647.3325842.

Beutel, D.J. *et al.* (2022) 'Flower: A Friendly Federated Learning Research Framework'. arXiv. Available at: https://doi.org/10.48550/arXiv.2007.14390.

Brodzik, A. *et al.* (2024) 'Ransomware Detection Using Machine Learning in the Linux Kernel'. arXiv. Available at: https://doi.org/10.48550/arXiv.2409.06452.

Carlos, W.C. *et al.* (2024) 'Human activity recognition: an approach 2D CNN-LSTM to sequential image representation and processing of inertial sensor data', *AIMS Bioengineering*, 11(4), pp. 527–560. Available at: https://doi.org/10.3934/bioeng.2024024.

Creech, G. (2014) *Developing a high-accuracy cross platform Host-Based Intrusion Detection System capable of reliably detecting zero-day attacks*. UNSW Sydney. Available at: https://doi.org/10.26190/UNSWORKS/16615.

Data Protection Impact Assessment (DPIA) (2018) *GDPR.eu*. Available at: https://gdpr.eu/data-protection-impact-assessment-template/ (Accessed: 9 April 2025).

Du, M. *et al.* (2017) 'DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning', in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery (CCS '17), pp. 1285–1298. Available at: https://doi.org/10.1145/3133956.3134015.

Duan, G. *et al.* (2023) 'DongTing: A large-scale dataset for anomaly detection of the Linux kernel', *Journal of Systems and Software*, 203, p. 111745. Available at: https://doi.org/10.1016/j.jss.2023.111745.

Farasat, T., Kim, J. and Posegga, J. (2024) 'SmartX Intelligent Sec: A Security Framework Based on Machine Learning and eBPF/XDP'. arXiv. Available at: https://doi.org/10.48550/arXiv.2410.20244.

Gao, J. *et al.* (2014) 'PLAID: a public dataset of high-resoultion electrical appliance measurements for load identification research: demo abstract', in *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*. *SenSys '14: The 12th ACM Conference on Embedded Network Sensor Systems*, Memphis Tennessee: ACM, pp. 198–199. Available at: https://doi.org/10.1145/2674061.2675032.

Grimmer, M. *et al.* (2023) 'Dataset Report: LID-DS 2021', in B. Hämmerli et al. (eds) *Critical Information Infrastructures Security*. Cham: Springer Nature Switzerland, pp. 63–73.

Gupta, O. and Raskar, R. (2018) 'Distributed learning of deep neural network over multiple agents'. arXiv. Available at: https://doi.org/10.48550/arXiv.1810.06060.

Karma IDS: An Intrusion Detection System using eBPF and LSTM (2024) *DEV Community*. Available at: https://dev.to/pree2111/karma-ids-497p (Accessed: 17 March 2025).

Kermenov, R. *et al.* (2023) 'Anomaly Detection and Concept Drift Adaptation for Dynamic Systems: A General Method with Practical Implementation Using an Industrial Collaborative Robot', *Sensors*, 23(6), p. 3260. Available at: https://doi.org/10.3390/s23063260.

Kim, R. *et al.* (2025) 'Detecting Cryptojacking Containers Using eBPF-Based Security Runtime and Machine Learning', *Electronics*, 14(6), p. 1208. Available at: https://doi.org/10.3390/electronics14061208.

Kim, S., Lee, S. and Kim, H.K. (2016) 'A LSTM-based system-call language model for host-based intrusion detection', *EURASIP Journal on Information Security*, 2016(1), pp. 1–14. Available at: https://doi.org/10.1186/s13635-016-0036-7.

Le, H. and Zhang, D. (2022) 'Deep learning approaches for log-based anomaly detection: A survey', *Journal of Systems and Software*, 192, p. 111361. Available at: https://doi.org/10.1016/j.jss.2022.111361.

Margaritelli, S. (2022) *Process Behaviour Anomaly Detection Using eBPF and Unsupervised-Learning Autoencoders*, *evilsocket*. Available at: https://www.evilsocket.net/2022/08/15/Process-behaviour-anomaly-detection-using-eBPF-and-unsupervised-learning-Autoencoders/index.html (Accessed: 14 June 2025).

Mollah, M.B., Azad, Md.A.K. and Vasilakos, A. (2017) 'Security and privacy challenges in mobile cloud computing: Survey and way ahead', *Journal of Network and Computer Applications*, 84, pp. 38–54. Available at: https://doi.org/10.1016/j.jnca.2017.02.001.

Paszke, A. *et al.* (2019) 'PyTorch: An Imperative Style, High-Performance Deep Learning Library'. arXiv. Available at: https://doi.org/10.48550/arXiv.1912.01703.

Pham, N.D. *et al.* (2022) 'Binarizing Split Learning for Data Privacy Enhancement and Computation Reduction'. arXiv. Available at: https://doi.org/10.48550/arXiv.2206.04864.

Priyadarshini, I. (2024) 'Anomaly Detection of IoT Cyberattacks in Smart Cities Using Federated Learning and Split Learning', *Big Data and Cognitive Computing*, 8(3), p. 21. Available at: https://doi.org/10.3390/bdcc8030021.

Redhu, S. *et al.* (2024) 'Deep learning-based anomaly detection in system calls using convolutional neural networks', *Future Generation Computer Systems*, 152, pp. 145–157. Available at: https://doi.org/10.1016/j.future.2023.11.017.

Ristea, N.-C. *et al.* (2022) 'Self-Supervised Predictive Convolutional Attentive Block for Anomaly Detection'. arXiv. Available at: https://doi.org/10.48550/arXiv.2111.09099.

Shi, W. *et al.* (2016) 'Edge Computing: Vision and Challenges', *IEEE Internet of Things Journal*, 3(5), pp. 637–646. Available at: https://doi.org/10.1109/JIOT.2016.2579198.

Thapa, C. *et al.* (2022) *SplitFed: When Federated Learning Meets Split Learning*. Available at: http://arxiv.org/abs/2004.12088 (Accessed: 2 February 2025).

Vepakomma, P. *et al.* (2018) 'Split learning for health: Distributed deep learning without sharing raw patient data'. arXiv. Available at: https://doi.org/10.48550/arXiv.1812.00564.

Verdeyen, C. *et al.* (2024) 'DECML: Distributed Edge Consensus Machine Learning Framework', in *2024 IEEE Global Conference on Artificial Intelligence and Internet of Things (GCAIoT). 2024 IEEE Global Conference on Artificial Intelligence and Internet of Things (GCAIoT)*, pp. 1–7. Available at: https://doi.org/10.1109/GCAIoT63427.2024.10833588.

ZeroMQ (no date). Available at: https://zeromq.org/ (Accessed: 9 September 2025).

Zhou, J.T. *et al.* (2020) 'Attention-Driven Loss for Anomaly Detection in Video Surveillance', *IEEE Transactions on Circuits and Systems for Video Technology*, 30(12), pp. 4639–4647. Available at: https://doi.org/10.1109/TCSVT.2019.2962229.

References

Douch, S., Abid, M.R., Zine-dine, K., Bouzidi, D. and Benhaddou, D. (2024) 'Split Edge-Cloud Neural Networks for Better Adversarial Robustness', *IEEE Access*. **DOI:** 10.1109/ACCESS.2024.3487435.

Qiu, X., Leontiadis, I., Melis, L., Sablayrolles, A. and Stock, P. (2024) 'Evaluating Privacy Leakage in Split Learning', *arXiv preprint*, arXiv:2305.12997. Available at: https://arxiv.org/abs/2305.12997 (Accessed: 08 July 2025).

Zhu, X., Luo, X., Wu, Y., Jiang, Y., Xiao, X. and Ooi, B.C. (2025) 'Passive Inference Attacks on Split Learning via Adversarial Regularization', in *Proceedings of the Network and Distributed System Security Symposium (NDSS 2025)*, San Diego, CA, 24–28 Feb 2025. **DOI:** 10.14722/ndss.2025.230030.

Appendix A: Glossary of Terms

Activation (Smashed Data)
The intermediate tensor produced by the client-side sub-model in split learning. Transmitted to the server for further processing instead of raw input data.

Accuracy
Metric indicating the proportion of correctly classified samples over the total evaluated.

Anomaly Detection
Task of identifying unusual patterns (e.g. malicious activity) in system calls or network behavior that deviate from normal profiles.

Ansible
Automation tool used to configure, deploy, and manage distributed BAD-SL components across multiple virtual machines.

Attention Entropy
A metric that quantifies how concentrated or spread out the attention distribution is. Lower entropy indicates sharper focus.

Attention Mechanism
Neural network module (multi-head self-attention in SL+Attention) that enhances interpretability by weighting important input positions.

BAD-SL (Behaviour-Aware Distributed Split Learning)
The proposed framework in this dissertation: integrates eBPF telemetry, CNN-based anomaly detection, and split learning for privacy-preserving distributed inference.

BCC (BPF Compiler Collection)
A toolkit for creating eBPF programs, used in BAD-SL to capture kernel telemetry such as system calls.

Centralized CNN
Baseline configuration where the full CNN model is trained and executed on a central server with all raw data.

Composite Privacy Score
Aggregated score (0–1) combining privacy metrics such as leakage, reconstruction, and membership inference risks.

Convolutional Neural Network (CNN)
Deep learning architecture using convolutional filters to extract patterns from sequential data (e.g. system calls).

CurveZMQ (ZMQ Curve Security)
ZeroMQ's built-in security mechanism for TLS-like encrypted communication channels.

DongTing Dataset
Linux system call dataset used in this study, containing normal and anomalous traces. 85% used for training, 15% reserved for testing.

Embedding Layer
Neural network layer that maps integer system call IDs into dense vector representations of fixed dimension (64 in BAD-SL).

eBPF (Extended Berkeley Packet Filter)
Linux kernel technology for efficient in-kernel telemetry and tracing, used here to capture system call sequences.

Federated Learning (FL)
Distributed learning paradigm where clients train local full models and share model updates with a central server for aggregation, without sharing raw data.

F1-Score
Harmonic mean of precision and recall, balancing false positives and false negatives in classification.

Gradient Reconstruction Attack
Adversarial attack attempting to reconstruct input data from shared gradients in FL or activations in SL.

Hyperparameters
Configurable training parameters (e.g. learning rate, batch size, sequence length) used to control model training.

Information Leakage Score
Quantitative measure of how much sensitive information can be inferred from shared activations or gradients.

Interpretability Confidence Score
Aggregate measure of how reliably the attention mechanism provides meaningful explanations.

Latency
Time taken for a full inference round, including client computation, communication, and server processing.

Membership Inference Attack (MIA)
Attack that attempts to determine whether a specific data sample was included in the training dataset.

Overhead (Communication)
Network cost per inference (KB) for transmitting activations between client and server in SL.

Precision
Metric that quantifies the proportion of detected anomalies that are truly anomalous.

Privacy Metrics
Set of metrics used to quantify privacy protection: leakage, reconstruction success, and MIA accuracy.

Recall
Metric that quantifies the proportion of actual anomalies that were correctly detected.

Resource Utilization
System resources consumed during inference, including CPU, memory, and network bandwidth.

Sequence Length
Maximum number of system calls per input sequence (1024 in this dissertation).

SL (Split Learning)
Distributed training/inference paradigm where the model is partitioned between client (early layers) and server (later layers). Raw data never leaves the client.

SL+Attention
Enhanced SL variant where the server includes an attention module for interpretability and improved accuracy.

Softmax Output
Final layer activation function used to produce class probabilities.

Syscall Localization Score
Metric quantifying how well attention focuses on anomalous system call positions.

System Call (Syscall)
Programmatic request from a user-space process to the Linux kernel. Basis of the telemetry used for anomaly detection in BAD-SL.

Tensor
Multidimensional numerical array used in deep learning computations.

Throughput
Number of sequences processed per second during inference.

TLS (Transport Layer Security)
Cryptographic protocol for secure communication. Optional in BAD-SL but not enabled in prototype experiments.

Virtual Machine (VM)
Software emulation of a physical computer, used to deploy BAD-SL clients and server in the evaluation setup.

ZeroMQ (ZMQ)
Lightweight messaging library used to implement client–server communication in BAD-SL, replacing heavier frameworks such as Flower.

Appendix B: CNN Model Architecture and BAD-SL Cut-Layer Design

The BAD-SL framework uses a one-dimensional convolutional neural network (CNN) to detect anomalies from sequences of system calls. This CNN begins with an embedding layer that maps each syscall identifier into a continuous feature vector, followed by several convolutional blocks that extract hierarchical patterns from the sequence. Convolutional filters scan local syscall motifs, and pooling layers progressively downsample the sequence while expanding the receptive field. At the end of the network, a global pooling layer aggregates information across the entire sequence and a final dense layer produces a probability of anomaly. All BAD-SL variants (centralized, split, federated, and split-with-attention) employ this same CNN structure. The choice of a 1D CNN reflects its proven ability to model sequential patterns efficiently and its suitability for split deployment: the convolutional feature extractors can be neatly partitioned between client and server. In particular, the client-side layers transform raw telemetry (system calls) into intermediate feature maps, and only these encoded activations are shared with the server, rather than raw data. This design inherently supports both effective anomaly modeling and stronger privacy guarantees.

Layer-by-Layer Architecture

The CNN processes each fixed-length syscall sequence through the following layers:

- **Embedding Layer:** Each input sequence of system calls (e.g. length 2048) is first passed through an embedding layer of dimension 64. This converts each discrete syscall ID into a 64-dimensional continuous vector, yielding an embedded sequence of shape 2048×64. Embedding allows the network to learn semantic relationships among different system calls, so that similar calls produce similar vectors.
- **Convolutional Block 1 (Conv1 + ReLU + MaxPool):** A 1D convolution with 32 filters (kernel size 5, stride 1, same padding) scans the embedded sequence. The convolution detects local patterns of up to 5 consecutive syscalls in the data. A ReLU non-linearity follows to introduce non-linear feature extraction. Then a 1D max-pooling layer (pool size 2, stride 2) reduces the sequence length by roughly half, retaining the most responsive features in each region. After pooling, the output feature map has shape 512×32 (i.e. length ~512 with 32 feature channels). This downsampling both increases the receptive field and reduces the amount of data sent to the server.
- **Convolutional Block 2 (Conv2 + ReLU + MaxPool):** On the server, the 512×32 activation is processed by a second 1D convolution with 64 filters (kernel size 5, stride 1). After another ReLU activation, a second max-pooling (pool size 2) halves the length again, yielding a feature map of shape 256×64. This block continues to capture longer-range patterns in the syscall sequence.
- **Convolutional Block 3 (Conv3 + ReLU):** A third 1D convolution (64 filters, kernel size 3) is applied with ReLU, producing an output still of shape 256×64 (no further pooling). At this point, the combined receptive field of the three convolutions

spans on the order of 44 syscalls. In practice, this depth was found sufficient to cover the typical short bursts of anomalous activity in the dataset.

- **Global Average Pooling:** To summarize the sequence features, a global average pooling layer computes the mean over the time dimension for each of the 64 feature channels. This produces a 64-dimensional feature vector representing the overall behavior of the entire 256-length sequence segment. Global pooling thus aggregates the learned patterns into a compact global representation.
- **Dense Output Layer:** The pooled 64-dimensional vector is fed into a final fully-connected layer with one neuron and a sigmoid activation. This output represents the probability (in [0,1]) that the input sequence is anomalous. During training, a binary cross-entropy loss is used to compare this score with the ground-truth normal/anomaly label.

Each of these layers plays a specific role: embeddings encode discrete calls into continuous space, convolutional filters detect local syscall motifs, pooling reduces dimensionality and enlarges context, and the global pooling plus dense layer synthesize the information to make a binary decision. The overall model is relatively lightweight (on the order of 50K parameters for the given syscall vocabulary) and optimized for edge deployment.

Cut-Layer Design for Split Learning

In BAD-SL's split learning variants, the CNN is partitioned between client and server at a designated **cut layer**. We chose to cut after the first convolutional block (i.e. after Conv1 + Pool1 on the client). Thus, the client computes the embedding and the first Conv1D+ReLU+MaxPool, yielding a 512×32 activation tensor. This intermediate activation is then transmitted to the server, and the server proceeds with the remaining Conv2, Conv3, pooling, and output layers. In effect, the client never sends raw syscall sequences—only an encoded feature map ("smashed data") is shared.

In practical terms, sending the 512×32 activation (approximately 16,384 floating-point values) corresponds to roughly 64 kilobytes per sequence. This is much smaller than sending the raw or fully-embedded sequence (e.g. the 2048×64 values would be ~130K floats). The choice of cut layer balances communication and privacy: by pooling once on the client, the data size is substantially reduced and meaningful feature abstraction is achieved, while the client still performs only light computation. Cutting even earlier (immediately after the embedding) would have produced a larger 2048×64 activation (roughly 520 KB of floats), whereas cutting later (after Conv2+Pool2) would further shrink the tensor but require more client-side work. Profiling showed that our chosen cut point provides a good compromise between data size and client load.

Because only encoded activations are sent, split learning preserves data privacy: sensitive raw syscall content remains local. The server sees only an abstract feature map, not the original call sequence. This design means that each client's raw telemetry and labels stay on-device, and only numeric activations (and their backpropagated gradients) cross the network. Such a partition ensures that the model still benefits from patterns learned across devices, while minimizing the risk of leaking private information from individual hosts.

Deployment in Centralized, Split, and Federated Modes

The same CNN architecture is deployed in three different configurations, each with distinct data flow and training roles (as summarized in Table 3.5).

- **Centralized:** In the centralized mode, all client data is sent to a single server, and the full CNN runs on that server. No part of the model is split. This yields the highest potential accuracy (since the model sees all data) but provides no privacy: raw system-call sequences must be transferred from each client to the server.
- **Split Learning (BAD-SL):** In the split-learning mode (the core BAD-SL approach), each client hosts the first portion of the CNN (embedding + Conv1 block) and performs local forward passes. After computing the intermediate activation, the client sends only this activation tensor to the server. The server holds the remaining layers and completes the forward/backward pass. During training, the server sends back gradients for the activation, allowing the client to update its local weights. Crucially, the client never uploads raw calls—only an encoded representation—which greatly enhances privacy. Communication overhead is limited to the size of the activation tensors (and occasional gradients), which is typically much smaller than raw telemetry. This partitioning means that sensitive details of host behavior are abstracted before leaving the device, at the cost of moderate network and synchronization overhead per batch.
- **Split Learning with Attention:** This variant is identical to BAD-SL except that the server's model includes an additional attention module (applied after the convolutional layers) for interpretability. The split remains the same: the client computes up through Conv1+Pool1, and the server receives the activation to feed into Conv2, Conv3, then attention and output layers. The attention mechanism adds little to the data flow (only slightly more computation and some additional weight parameters on the server), but it provides insight by highlighting which parts of the activation are most relevant.
- **Federated Learning:** In federated mode, each client holds a full copy of the CNN and trains locally on its own data. Periodically, clients send only model weight updates (gradients or parameters) to a central server, which aggregates them (e.g. via FedAvg). No part of the model inference is split in this case. Raw data never leaves the client, and no intermediate activations are transmitted. The privacy tradeoff is different: federated learning keeps raw data local, but all clients end up sharing the same model parameters, which can in principle leak some information. Communication involves periodic exchange of parameters rather than per-example activations. Federated learning can achieve comparable detection performance while preserving data locality, but it typically incurs more rounds of communication and offers weaker protection against model-inversion attacks than split learning.

In summary, the CNN itself remains the same in all four setups: we always use the same embedding + 3-conv-block design. What changes is the deployment: centralized has all layers on one server, split learning partitions the network between client and server (sharing activations), and federated learning replicates the entire model on each client (sharing weight updates). Each approach represents a different point in the privacy–performance trade-off. BAD-SL (split) was chosen to strike a balance: it achieves nearly

the same accuracy as centralized training (since the server ultimately trains on rich features from all clients) while avoiding raw data transfer. The split-with-attention variant adds interpretability by using attention at the server, and the federated variant explores how full-model training with secure aggregation compares in terms of accuracy and overhead.

Overall, the CNN architecture and cut-layer design were driven by the need to effectively learn from syscall sequences while respecting data privacy. Embeddings and convolutions provide powerful anomaly detection capability on sequential data, and the strategic model split ensures that clients expose only high-level features to the network. These design choices collectively enable BAD-SL to detect intrusions with high accuracy while keeping sensitive host behavior local.

Appendix C: System Configuration and Security Hardening

This appendix describes the system-level configurations and security measures applied in the BAD-SL deployment. Ensuring a secure and stable operation was important, given that the framework deals with potentially sensitive data and is distributed across multiple machines.

Operating System and Kernel: Both the edge and server were run on Linux (Ubuntu 22.04 LTS). We standardized on kernel version 5.15 for both, to minimize any discrepancies in system calls or container behaviour. The kernel choice was also influenced by maturity and support; 5.15 is a long-term support (LTS) kernel, meaning it has received extensive testing and will continue to get security updates. We explicitly disabled automatic updates to the kernel during our experiment window to maintain a consistent environment (as noted, this is one reason we had a fixed kernel version requirement). Security patches were applied up to a certain point, after which the system was considered "frozen" for the duration of testing.

Containerization and Isolation: Each component of BAD-SL runs in a Docker container managed by K3s. Containers provide a level of isolation – for instance, the client container cannot directly see the server's filesystem, and vice versa. We used Kubernetes namespace isolation so that only the necessary services can communicate. The client and server pods ran under a dedicated namespace (e.g., `badsl-namespace`) and network policies were applied to restrict traffic: the client pod was only allowed to initiate connections to the server pod's service on the specific port, and no external inbound access to the client was permitted (the edge device does not run any server service). This means even if an attacker got onto the same network, the architecture doesn't expose an open port on the client side – the server is the only one listening for connections, and it is hardened as described next.

Network Security: On the server (cloud) node, we enabled UFW (Uncomplicated Firewall) and only opened the port needed for the BAD-SL server API (for example, port 3000/TCP was used for the HTTP endpoint to receive activations). All other ports were closed to outside. Communication between the client and server uses HTTP over TLS (HTTPS). We set up a self-signed certificate for the server service (or in a real deployment, one would use a proper CA-signed cert) to ensure that the activation data in transit is encrypted. This protects against eavesdropping on the network; even though the activations are not easily interpretable, it was still important to encrypt traffic as a defense-in-depth measure. Additionally, we enforced client authentication at the application layer – the client includes an API token in its requests which the server verifies, preventing unauthorized devices from sending data to the server. In Kubernetes, this was facilitated by mounting a secret (token) into the client and server pods so that they share a secret key.

System Hardening: We applied several Linux sysctl settings and best practices for hardening both nodes:

- Disabled password logins over SSH (using key-based login only) to prevent brute-force remote access.
- Configured `sysctl` parameters such as `kernel.unprivileged_bpf_disabled=1` (to disable unprivileged eBPF which can be a security risk), `kernel.yama.ptrace_scope=2` (to restrict ptrace system call for better process isolation), and enabled address space layout randomization (ASLR) at the highest level (`kernel.randomize_va_space=2`).
- Limited the number of processes and open file handles for the BAD-SL user account to mitigate any fork-bomb or resource exhaustion scenarios (this also helps if the code had a bug that could spawn processes uncontrollably).
- Ensured that the Docker daemon was configured to use user namespaces (so that containers do not run as the root UID on the host, adding a layer of protection if a container were compromised).

Inside the containers, we also took some measures:

- The container images were built on minimal base images (Alpine Linux for some components, to reduce attack surface).
- We dropped unnecessary Linux capabilities from the containers. By default, Docker gives containers a set of capabilities; using Kubernetes securityContext, we removed things like `SYS_ADMIN`, `NET_RAW` etc., which were not needed for our application. The containers basically only needed network and process capabilities to run the model code.
- We set the user in the container to a non-root user (so even inside the container, the code does not run as root).

Resource Limits and Monitoring: As seen in Appendix B, we set resource limits in the Kubernetes manifests to ensure the client and server pods don't exceed expected CPU or memory usage (this prevents a runaway process from crashing the edge device by exhausting memory, for example). We also used Kubernetes liveness probes to automatically restart the pods if they became unresponsive (for instance, if the model code hung, the liveness probe would fail and K3s would restart that container). This provided a self-healing aspect to the deployment.

We logged system metrics using `sar` and Kubernetes metrics server to verify that throughout operation, the system stayed within safe bounds (CPU temperatures on the Raspberry Pi, for example, were monitored to ensure our workload didn't overheat the device – which it did not, given the moderate CPU usage).

In summary, Appendix C outlines that our BAD-SL deployment was not only functional but also **reasonably secure and robust** for a prototype. By hardening the OS, using container isolation, encrypting data in transit, and limiting resources, we attempted to reduce the risk of both external attacks and accidental failures. These measures are

important if such a system were to be deployed in the field, especially in applications where data sensitivity is high. Future iterations could incorporate even more advanced security (like running the server component inside a Trusted Execution Environment to shield it from the cloud OS, etc.), but the steps we took were aligned with standard DevOps security practices for distributed applications.

Appendix D: Experiment Scripts and Logs

This appendix describes the structure of the experiment scripts and provides samples of logs generated during the BAD-SL evaluation. The purpose is to give insight into how we orchestrated the experiments and to enable reproducibility.

Experiment Scripts Structure: We organized our experiment code in a dedicated repository (or directory) with the following structure:

```
badsl
├── deploy.yml
├── env_validate.yml
├── hosts.ini
├── inference
│   ├── attention_split_client.py
│   ├── attention_split_server.py
│   ├── centralized_inference.py
│   ├── data_source.py
│   ├── federated_client.py
│   ├── federated_server.py
│   ├── metrics.py
│   ├── split_client.py
│   └── split_server.py
├── lab_runner.py
├── models
│   ├── best_attention_model.pth
```

```
│   ├── best_centralized_model.pth
│   ├── best_federated_model.pth
│   ├── best_split_learning_model.pth
│   └── unified_badsl_results.json
├── runs
│   └── 2025-08-05
│       ├── centralized
│       │   ├── eval_summary.json
│       │   ├── inference_log.jsonl
│       │   ├── manifest.yaml
│       │   └── system_metrics.csv
│       ├── federated
│       │   ├── comm_stats.csv
│       │   ├── eval_summary.json
│       │   ├── inference_log.jsonl
│       │   ├── manifest.yaml
│       │   └── system_metrics.csv
│       ├── sl_attention
│       │   ├── attention_maps
│       │   │   ├── attn_map_seq_00001.bin
│       │   │   ├── attn_map_seq_00002.bin
│       │   │   ├── attn_map_seq_00003.bin
│       │   │   └── attn_map_seq_00050.bin
│       │   ├── comm_stats.csv
│       │   └── system_metrics.csv
│       └── split_learning
│           ├── comm_stats.csv
```

```
│           ├── eval_summary.json
│           ├── inference_log.jsonl
│           ├── manifest.yaml
│           └── system_metrics.csv
├── setup_ssh_keys.sh
├── syscall_vocabulary.json
├── syscall_vocabulary_discovered.json
├── training
│   ├── data
│   │   └── clean_splits
│   │       ├── dongting_test.pkl
│   │       ├── dongting_train.pkl
│   │       ├── dongting_val.pkl
│   │       ├── program_families.json
│   │       └── validate_split.py
│   ├── dongting_full_cache.pkl
│   ├── dongting_test_15percent.pkl
│   ├── program_aware_splitter.py
│   ├── split.py
│   ├── splits
│   │   ├── test.pkl
│   │   ├── train.pkl
│   │   └── val.pkl
│   ├── syscall_vocabulary.json
│   └── unified_badsl_trainer.py
```

- deploy.yml — Ansible playbook to deploy/start the BAD-SL inference stack on server/clients with predefined roles, variables, and tasks.

- env_validate.yml — Ansible playbook that checks machine prerequisites (Python/venv, kernel headers, ports, packages) before deployment.
- hosts.ini — Ansible inventory listing the server and edge clients (IP/host groups) used by deploy.yml.

inference/

- attention_split_client.py — Client-side process for Split Learning with attention: embeds + first CNN block, streams smashed activations to the attention server.
- attention_split_server.py — Server-side counterpart hosting attention + classifier; receives activations and returns predictions/metrics.
- centralized_inference.py — Baseline centralized pipeline: loads the full CNN on the server, consumes sequences from the data source, and outputs anomaly scores.
- data_source.py — Sequence producer for live eBPF traces or replayed test sets; performs vocab encoding/segmentation and publishes batches via ZeroMQ.
- federated_client.py — Federated variant client running the full model locally and reporting predictions/updates to the federated server.
- federated_server.py — Federated coordinator/aggregator that collects client results (and comm stats) and writes run artifacts.
- metrics.py — Shared utility to record latency/throughput/accuracy and system stats; writes JSON/CSV for Chapter 5 tables.
- split_client.py — Split Learning client (CNN encoder on edge); sends intermediate activations to the split server and logs client-side metrics.
- split_server.py — Split Learning server (decoder/classifier on core); completes forward pass, emits predictions, and aggregates run metrics.
- lab_runner.py — Orchestrator script to launch selected pipelines (centralized/federated/split/attention) with consistent arguments and output paths.

models/

- best_attention_model.pth — Checkpoint of the best CNN+Attention split model used for inference runs.
- best_centralized_model.pth — Trained centralized CNN model.
- best_federated_model.pth — Trained model selected from federated runs.
- best_split_learning_model.pth — Trained split-learning CNN (client/server compatible weights).
- unified_badsl_results.json — Aggregated summary of key metrics across modes (used to populate dissertation tables/figures).

runs/2025-08-05/centralized/

- eval_summary.json — Accuracy/AUROC and confusion-matrix style summary for the centralized run.

- inference_log.jsonl — Per-batch/sequence prediction log (JSON Lines) for traceability.
- manifest.yaml — Run metadata (model hash, args, seeds, data paths, timestamps).
- system_metrics.csv — Time-series CPU/RAM/I/O metrics collected during the run.

runs/2025-08-05/federated/

- comm_stats.csv — Message counts/bytes and round timings for the federated session.
- eval_summary.json — Federated evaluation summary (same schema as centralized).
- inference_log.jsonl — Per-sequence prediction records from federated execution.
- manifest.yaml — Provenance/config snapshot for the federated run.
- system_metrics.csv — System resource usage timeline during federated inference.

runs/2025-08-05/sl_attention/

- attention_maps/attn_map_seq_*.bin — Serialized attention weights per sequence for interpretability analysis.
- comm_stats.csv — Activation-transfer statistics (count/bytes/latency) for attention-split mode.
- system_metrics.csv — Resource footprints while running attention-split inference.

runs/2025-08-05/split_learning/

- comm_stats.csv — Activation payload metrics (size, rate) for standard split learning.
- eval_summary.json — Split-learning evaluation summary.
- inference_log.jsonl — Sequence-level outputs for the split run.
- manifest.yaml — Configuration and environment snapshot for reproducibility.
- system_metrics.csv — System-level metrics during split inference.
- setup_ssh_keys.sh — Helper to provision SSH keys/known_hosts for Ansible and scripted remote execution.
- syscall_vocabulary.json — Canonical mapping of syscall names → IDs used by the encoder.
- syscall_vocabulary_discovered.json — Extended/observed mapping produced during data exploration (captures OOV handling decisions).
- training/
- data/clean_splits/ — Cleaned, finalized splits for experiments.
- dongting_train.pkl, dongting_val.pkl, dongting_test.pkl — Pickled tensors/sequences and labels for each split.
- program_families.json — Mapping of binaries/program families to help program-aware splitting and stratification.

- validate_split.py — Sanity checks ensuring no leakage and expected class/program distributions.
- dongting_full_cache.pkl — Cached raw/parsed DongTing dataset prior to final cleaning.
- dongting_test_15percent.pkl — Test subset (≈15%) used for the reported evaluations.
- program_aware_splitter.py — Splitting utility that groups by program family to avoid train-test contamination.
- split.py — General dataset splitter/packer (ratio-based, seed-controlled).
- splits/train.pkl, splits/val.pkl, splits/test.pkl — Earlier or alternative split artifacts retained for reproducibility.
- syscall_vocabulary.json — Local training copy of the syscall ID mapping (kept with data to freeze preprocessing).
- unified_badsl_trainer.py — End-to-end trainer supporting centralized, split, federated, and attention variants with consistent I/O and checkpoints.

Sample Log Snippet (Client and Server):

From `client.log`:

```
[INFO] 2025-06-01 10:23:45 - Loaded 1000 test images for inference.
[INFO] 2025-06-01 10:23:45 - Starting inference on test set...
[DEBUG] 2025-06-01 10:23:45 - Processing image 1 locally.
[DEBUG] 2025-06-01 10:23:45 - Local inference time: 8 ms. Activation shape: (16, 16, 32).
[DEBUG] 2025-06-01 10:23:45 - Sending activation to server (payload 32768 bytes).
[DEBUG] 2025-06-01 10:23:45 - Received prediction from server: Class=2, Confidence=0.85.
[DEBUG] 2025-06-01 10:23:45 - Image 1 done. Total time: 20 ms.
...
[INFO] 2025-06-01 10:23:50 - Inference complete. Processed 1000 images in 5.2 s (avg 5.2 ms/image edge + network + 7.8 ms/image server).
```

From `server.log`:

```
[INFO] 2025-06-01 10:23:45 - BAD-SL Server started, waiting for activations...
[DEBUG] 2025-06-01 10:23:45 - Received activation from client. Size: 32768 bytes.
[DEBUG] 2025-06-01 10:23:45 - Running server-side inference...
[DEBUG] 2025-06-01 10:23:45 - Inference done in 12 ms. Predicted Class=2.
[DEBUG] 2025-06-01 10:23:45 - Sent prediction back to client.
...
[INFO] 2025-06-01 10:23:50 - Served 1000 inferences. Average server compute time: 12.1 ms.
```

These log snippets illustrate the sequence: the client processes an image, sends the activation, server processes it, sends back result. The timestamps help to calculate where

time is spent (as shown, ~8 ms on client, ~12 ms on server in this example, plus network transit which can be inferred by the gaps).

Reproducibility: We fixed random seeds in the training scripts (for weight initialization and batch shuffling) to ensure that results were consistent across runs for comparison. The code environment (library versions) is documented in a `requirements.txt` (for Python packages) and the container image specifications (Appendix C mentions the base OS and security settings). Together, the scripts and logs serve as a record of what was run and what was observed, which is crucial for both debugging and any future work building on this project.

We include this level of detail in Appendix D to aid anyone who wants to replicate our experiments or understand the execution flow of BAD-SL. The logs in particular provide insight into the runtime behavior, while the scripts outline how to set up similar tests. Minor differences might be present if running on different hardware (e.g., absolute timings), but the relative comparisons and methodology should hold.