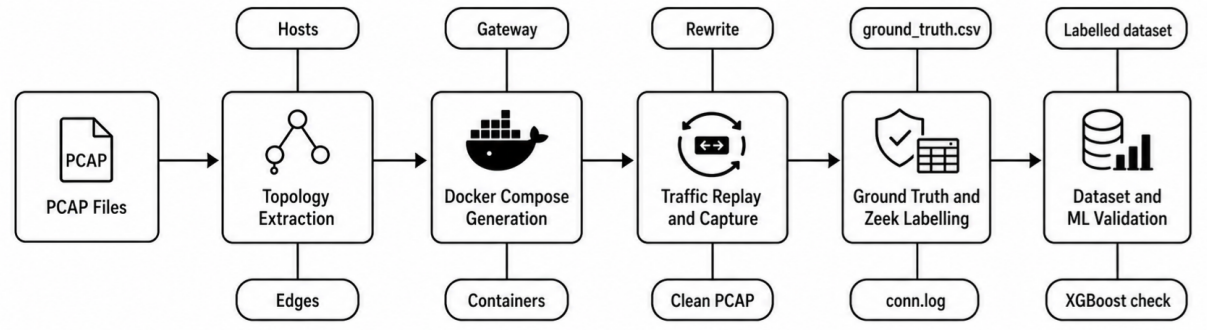
CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG



Malicious Traffic Generator for ML-Based Network Anomaly Detection

Master's thesis in Computer Science and Engineering

MOHAMMAD MOURAD

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

Malicious Traffic Generator for ML-Based Network Anomaly Detection

Mohammad Mourad

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

Malicious Traffic Generator for ML-Based Network Anomaly Detection
Mohammad Mourad

Supervisor: Muoi Tran and Ilias Chanis, Department of Computer Science and Engineering
Examiner: Yinan Yu, Department of Computer Science and Engineering

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Overview of the proposed Packet Capture (PCAP) to labelled flow traffic generation framework.

Typeset in LaTeX
Gothenburg, Sweden 2026

Malicious Traffic Generator for ML-Based Network Anomaly Detection

Mohammad Mourad
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Good-quality labelled network traffic remains a key bottleneck for research in machine learning-based intrusion detection. Canadian Institute for Cybersecurity Intrusion Detection System 2017 (CICIDS2017) and University of New South Wales Network-Based 2015 (UNSW-NB15) public benchmarks played an important role for evaluation; however, these datasets are mostly static and problematic to reproduce, adapt or modify to fit new requirements posed by containerization and service-oriented architectures. This thesis addresses this problem by proposing a reproducible framework to construct, replay, capture and label malicious traffic from Packet Capture (PCAP) traces inside a Docker-based testbed.

The framework identifies communicating hosts, protocol edges, DNS names, and Dynamic Host Configuration Protocol (DHCP) metadata from the input traces. These elements are then mapped into a synthetic multi-zone topology with automatic Docker Compose configuration generation. Traffic is then rewritten and replayed from simulated source containers via a Scapy-based replay engine. A routed gateway is used as an observation point, a delay-injection point, and a capture point. Metadata about the replay process is stored as ground truth, traffic is converted to Zeek connection logs, and flow labels are derived based on replay time windows and traffic class metadata. An additional packet-to-flow mapping step is performed to improve data traceability.

While a new intrusion detection model is not a key contribution of this thesis, it introduces a reproducible pipeline for constructing a malicious traffic dataset. After early live-execution trials, the project shifted to a replay-based design in order to improve reproducibility, containment, and experimental control.

Preliminary machine learning (ML) evaluation using Zeek connection-level features and an Extreme Gradient Boosting (XGBoost) classifier showed that replay-generated datasets achieved classification performance close to datasets generated directly from the original PCAP traces. The results suggest that the replay process preserved many of the flow-level statistical properties relevant for ML-based intrusion-detection tasks.

Keywords: malicious traffic generation, PCAP replay, Docker testbed, Zeek flow labelling, packet-to-flow traceability, intrusion detection, machine learning.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Ilias Chanis and Muoi Tran, for their continuous guidance, valuable feedback, and support throughout this thesis project. Their expertise, encouragement, and constructive discussions greatly contributed to the development of this work. I would also like to thank the examiner, Yinan Yu, for the time, evaluation, and academic feedback provided during the thesis process. Special appreciation is extended to Chalmers University of Technology and Secura Lab for providing the academic environment, technical resources, and research support necessary for conducting this project. Finally, I would like to thank my family for their support and encouragement throughout my studies. I am especially grateful to my father and mother for always encouraging me, and to my wife for her patience, understanding, and continuous support during the completion of this thesis.

Mohammad Mourad, Gothenburg, 2026-06-05

List of Acronyms

Acronym	Meaning
AI	Artificial Intelligence
ARP	Address Resolution Protocol
CICIDS2017	Canadian Institute for Cybersecurity Intrusion Detection System 2017
CPU	Central Processing Unit
CSV	Comma-Separated Values
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
FIN	Finish
GAN	Generative Adversarial Network
GiB	Gibibyte
GNS3	Graphical Network Simulator-3
HTB	Hierarchical Token Bucket
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
ID2T	Intrusion Detection Dataset Toolkit
IDS	Intrusion Detection System
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
JS	Jensen–Shannon
KS	Kolmogorov–Smirnov
LLMNR	Link-Local Multicast Name Resolution

Acronym	Meaning
LTS	Long-Term Support
MAC	Media Access Control
mDNS	Multicast DNS
ML	Machine Learning
NAT	Network Address Translation
NBNS	NetBIOS Name Service
PCAP	Packet Capture
RAM	Random Access Memory
RTT	Round-Trip Time
SSL	Secure Sockets Layer
SSDP	Simple Service Discovery Protocol
SYN	Synchronize
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
UNSW-NB15	University of New South Wales Network-Based 2015
UTC	Coordinated Universal Time
WSL2	Windows Subsystem for Linux 2
XGBoost	Extreme Gradient Boosting

Contents

List of Tables

List of Figures

1

Introduction

1.1 Problem Context

Intrusion Detection Systems (IDS) remain a central component of modern cyber-defence architectures because they monitor network traffic for malicious activity, misuse, and policy violations. Traditional network IDS approaches often depend on signatures of previously known attacks. Such methods can be effective when the attack pattern is already documented, but they are less effective against zero-day threats, evolving malware behaviour, and attack variants that deliberately avoid known signatures. These limitations are one of the reasons anomaly-based and machine-learning-based detection methods have received long-standing research attention [22].

Anomaly detection in networks, however, is not simply a matter of choosing a stronger classifier. Network behaviour varies over time, legitimate traffic is heterogeneous, and laboratory evaluations often do not represent real deployment conditions. Sommer and Paxson argue that intrusion detection does not fit a simple "closed world" machine-learning assumption, because the distinction between benign and malicious traffic is tied to an environment that evolves continuously [22]. As a result, the quality, realism, and provenance of the training and testing data become just as important as the learning model

itself.

1.2 Motivation

A major practical difficulty in network-anomaly-detection research is the lack of traffic data that is simultaneously realistic, controllable, reproducible, and well labelled. Public benchmark datasets such as CICIDS2017 and UNSW-NB15 have had major impact because they provide packet traces, derived features, and attack labels [20, 15]. Yet the literature has repeatedly pointed out that benchmark datasets age quickly, do not necessarily reflect current infrastructures, and are difficult to regenerate exactly under new conditions [21, 17, 10].

This problem is especially important for ML-based IDS research. If the data-generation process is opaque, if labels cannot be traced back to concrete traffic events, or if the environment cannot be repeated under controlled variations, then later classifier results become difficult to trust. For this reason, this thesis focuses on dataset engineering rather than starting from the machine-learning phase. The thesis investigates how malicious communication traces captured externally in sandbox environments can be reconstructed and replayed in a reproducible local environment whose topology, routing, timing, capture position, and label generation are all explicitly controlled.

1.3 Research Questions

This thesis investigates the following research questions:

1. How can malicious network traffic captured from sandbox-generated PCAP files be reproducibly reconstructed and replayed within a controlled containerized network environment?

2. How can relevant properties of the original traffic, such as topology, source placement, timing relations, and bidirectional communication context, be preserved or reconstructed inside that simulated environment?

3. How can packet-level and flow-level ground truth be generated in a transparent and traceable way so that downstream anomaly-detection experiments can rely on auditable labels?

1.4 Aim and Scope

The aim of this thesis is to design and implement a modular framework for generating labelled malicious network traffic through replay and reconstruction rather than through one-time dataset collection alone. The final system is intended to begin from PCAP traces captured from controlled malware sandboxes, extract their communication structure, map that structure into a generated Docker network, replay the traffic under controlled timing conditions, capture the resulting traffic, and export labelled flow-level data for later analysis.

The scope of the present thesis is primarily architectural and engineering-oriented. It includes topology extraction, automatic Docker Compose generation, replay orchestration, gateway-based capture, and delay injection. It also includes ground-truth generation, Zeek-based flow export, packet-to-flow traceability, and ML validation. The ML component is not presented as a novel detector contribution. Instead, it is used as an evaluation mechanism to test whether the simulated traffic preserves useful discriminative information for malicious-flow classification.

In this thesis, ML-based network anomaly detection is used as the broader application context for the generated dataset. The implemented ML evaluation itself is a supervised binary classification baseline using benign and malicious flow labels, rather than a full unsupervised anomaly-detection model.

1.5 Contributions

The main contributions of the thesis can be summarized as follows:

1. A replay-centred methodology for reconstructing malware-related network traffic from sandbox-generated PCAP traces under controlled experimental conditions.
2. A topology-extraction and environment-generation pipeline that maps original traffic into a Docker-based multi-zone routed network with deterministic addressing and explicit capture points.
3. A traceable labelling workflow that combines replay metadata, gateway captures, Zeek connection logs, and packet-to-flow mapping to produce auditable malicious/benign annotations.
4. An engineering validation of replay fidelity, timing control, capture quality, and automation, demonstrating that the framework can support later ML-based anomaly-detection studies.

1.6 Thesis Outline

The remainder of the thesis is structured as follows. Chapter 2 presents background and related work on intrusion detection, dataset generation, flow-based traffic analysis, and the technical tools that informed the implementation. Chapter 3 describes the research methodology, the iterative development process, and the final architecture and pipeline. Chapter 4 presents the implementation results and engineering validation achieved so far. Chapter 5 discusses the significance of the work in relation to prior dataset literature and analyses the remaining challenges. Chapter 6 summarizes the main limitations and identifies directions for future work. Chapter 7 addresses ethical and safety aspects. Chapter 8 concludes the thesis.

2

Background and Related Work

2.1 Intrusion Detection and Anomaly Detection

Network intrusion detection is commonly divided into misuse detection and anomaly detection. Misuse detection searches for previously defined patterns, rules, or signatures. Anomaly detection instead attempts to identify traffic that deviates from expected behaviour, which in principle makes it attractive for detecting previously unseen attacks. Machine learning has a natural place in anomaly detection because it can model traffic distributions, temporal patterns, and flow relationships that may be difficult to encode manually. At the same time, network intrusion detection remains a difficult application domain for ML because attack rarity, concept drift, contextual dependence, and operational cost of false positives all complicate evaluation and deployment [22].

These challenges imply that research claims about detection performance are only meaningful when the underlying dataset is suitable for the intended task. A detector trained on unrealistic or weakly labelled traffic may achieve strong numerical performance while

learning artefacts of the dataset rather than properties of malicious behaviour. This is one of the central motivations for the present thesis.

2.2 Requirements for Useful Network-Security Datasets

A recurring theme in the IDS dataset literature is that the usefulness of a dataset depends on more than its size. Shiravi et al. argue that the field should move away from static and one-time datasets toward dynamically generated datasets that are modifiable, extensible, and reproducible [21]. Ring et al. later systematize the problem further by identifying properties such as recording environment, traffic diversity, data volume, feature availability, and label quality as major axes for evaluating intrusion-detection datasets [17]. Guerra et al. make a related point from the labelling perspective: even when traffic is available, obtaining representative and validated labels remains difficult, costly, and methodologically fragile [10].

These observations are particularly relevant for anomaly-detection research. The challenge is not merely to gather traffic, but to know how the traffic was produced, what attack events occurred, where they occurred in the topology, which packets belong to them, and how those facts were translated into final labels. The present thesis therefore treats provenance and reproducibility as first-order design goals rather than as secondary implementation details.

2.3 Benchmark Datasets

2.3.1 CICIDS2017

CICIDS2017 is one of the most widely used benchmark datasets in intrusion-detection research. The dataset provides five days of traffic with benign activity and several attack scenarios, and it includes labelled flow records derived from timestamps, addresses, ports, protocols, and documented attack schedules [20, 3]. Its importance lies not only in the data itself, but also in the way the dataset combines raw traces with derived flow representations and explicit labelling criteria.

2.3.2 UNSW-NB15

UNSW-NB15 is another widely cited dataset that combines raw packet captures with Zeek/Bro and Argus-derived data, explicit feature extraction, and attack categories generated in a cyber range environment [15, 24]. The dataset demonstrates a structured

approach to combining raw traffic, derived features, and categorical labels, and it remains a common reference point for IDS evaluation.

2.3.3 Limitations of Static Benchmarks

Although such datasets are valuable, they do not remove the broader need for regenerable traffic-generation methods. Static datasets inevitably reflect a particular time, environment, workload, and labelling methodology. They are also difficult to adapt when a researcher wants to change topology, replay timing, protocol mix, capture position, or attack placement. The literature therefore continues to stress the importance of reproducibility, extensibility, and explicit label provenance [21, 17, 10]. The present thesis is positioned within that methodological gap.

2.4 Packet and Flow Representations

In network security research it is common to move between packet-level and flow-level representations. Packet traces preserve the most detailed network evidence: headers, payload fragments, timing, ordering, and low-level protocol interactions. Flow representations aggregate packets that belong to the same communication context and reduce the data into more manageable connection records. This aggregation is often preferable for ML because it reduces dimensionality while retaining key statistical and behavioural properties.

Zeek is an open-source network security monitoring framework that analyses packet captures and produces structured logs describing network activity. In this thesis, Zeek is used mainly to convert PCAP traffic into `conn.log` connection records. These records summarize communication flows using fields such as timestamps, endpoints, ports, protocols, packet counts, byte counts, and connection state [27]. This makes Zeek suitable for transforming packet-level replay output into flow-level data for labelling and later ML validation.

Zeek is particularly relevant in this context because its `conn.log` is designed as a core connection summary for both stateful and non-stateful transport activity, including Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) traffic [27]. For a thesis focused on malicious traffic generation, the important point is not to choose one representation over the other, but to preserve traceability between them. This is why the present work combines PCAP replay, Zeek flow extraction, and an explicit packet-to-flow mapping stage.

2.5 Traffic Generation and Replay

Traffic generation is an important part of intrusion-detection research because ML-based IDS evaluation depends strongly on the quality, representativeness, and labelling of the underlying traffic data. Existing approaches can be grouped broadly into synthetic traffic generation, attack-injection frameworks, testbed-based generation, public PCAP repositories, and replay-based methods. Each approach provides a different tradeoff between realism, controllability, scalability, and label traceability.

Synthetic traffic generation aims to create new traffic records that statistically resemble real network traffic. Ring et al. propose a generative adversarial network (GAN)-based method for generating flow-based network traffic and discuss the difficulty of modelling mixed numerical and categorical flow attributes such as ports, protocols, and addresses [16]. NetShare follows a related direction and uses generative models to create synthetic packet- and flow-header traces for networking tasks [26]. These methods are relevant because they address the need for scalable and privacy-preserving traffic generation. However, their main focus is statistical synthesis rather than replaying a specific PCAP trace with its original packet ordering, topology relationships, and packet-level evidence.

Another approach is attack injection into existing benign traffic. The Intrusion Detection Dataset Toolkit (ID2T) processes background PCAP files and injects synthetic attack traffic into them in order to create labelled IDS datasets [8]. This approach is useful because it combines real benign background traffic with controlled attack scenarios. It also supports repeatability because the injected attack configuration can be documented and reproduced. In contrast, the present thesis starts from already captured benign or malicious PCAP traces and focuses on reconstructing and replaying those traces inside a controlled containerized environment.

Testbed-based approaches generate traffic by running services, clients, attack tools, and sometimes malware-like behaviour inside a controlled network environment. Network emulation platforms such as Graphical Network Simulator-3 (GNS3) are commonly used to construct virtual network topologies for testing and education [9]. Security-focused testbeds such as Gotham provide reproducible environments for generating Internet of Things (IoT) security datasets through emulated devices, benign workloads, services, and attacks [18]. These systems are relevant because they demonstrate how controlled environments can support repeatable traffic generation. Their starting point is usually a designed scenario or emulated network, whereas this thesis derives the replay topology from observed communication relationships in an input PCAP trace.

Public PCAP repositories provide another important source of traffic for network-security research. The Stratosphere Laboratory publishes malware, normal, mixed, and IoT malware captures for research and education [23]. Malware-Traffic-Analysis.net also publishes malware-related PCAP traces and traffic-analysis exercises [14]. Such repositories are valuable because they provide realistic packet-level evidence of malicious behaviour. However, the PCAP files themselves are static artefacts. Additional processing is required if researchers want to replay them, reconstruct their topology, generate new labelled flow datasets, or evaluate replay fidelity.

Replay-based methods reuse previously captured traffic by transmitting or reconstructing packets in a controlled environment. Tcpreplay and tcprewrite are established tools for replaying PCAP traces and modifying packet headers before replay [1, 2]. Hussain et al. study malicious traffic replay in network testbeds and highlight the value of replaying real attack traces because replay can preserve fine-grained timing and packet-level behaviour that may be difficult to model synthetically [11]. These works are especially relevant to this thesis because the implemented framework also uses replay as the basis for reproducible traffic generation.

The present thesis is positioned within this replay-based line of work. Rather than generating entirely synthetic traffic or injecting synthetic attacks into a background trace, it takes existing PCAP files as input, extracts their communication structure, maps observed hosts into a Docker-based topology, replays packets from simulated source containers, captures the resulting traffic at a routed gateway, and exports labelled Zeek flows. The focus is therefore on reproducible PCAP reconstruction, controlled replay, and traceable labelling for later ML-based IDS experiments.

3

Methodology and System Design

3.1 Research Methodology and Design Requirements

The thesis followed an iterative engineering methodology. Instead of assuming a final architecture from the beginning, the system was refined through repeated design–implementation–validation cycles documented in weekly development records. This iterative process was necessary because the initial goal—to execute malware directly inside the target environment and generate labels from the execution context—introduced several practical and environmental challenges.

The project initially explored direct malware execution inside controlled containerized environments. This direction was attractive because it could, in principle, preserve live runtime behaviour. In practice, however, it introduced significant reproducibility and networking challenges. Windows containers provided less flexible topology construction, routing control, and network configuration than the Linux-based Docker environment. In addition, many malware samples depended on external DNS resolution, command-and-

control infrastructure, or server-side responses that were not available inside the isolated testbed. Although selected DNS and responder behaviour could be emulated locally, reproducing stable live execution across repeated experiments proved unreliable.

For these reasons, the framework shifted from live malware execution toward replaying externally captured PCAP traces inside a controlled Linux-container testbed. This replay-centred approach preserved the network-level behaviour needed for IDS dataset generation while improving reproducibility, containment, topology control, capture visibility, and label transparency.

From the beginning, five design requirements guided the work:

1. **Reproducibility:** the same input trace and mapping should generate the same network structure, replay process, and labels.
2. **Traceability:** labels should be auditable back to explicit replay events rather than inferred from opaque preprocessing.
3. **Containment:** all malicious communication should remain within an isolated local environment.
4. **Configurability:** topology, timing, routing, and capture points should be changeable without rewriting the full environment manually.
5. **Flow-oriented export:** the output should support later ML-based IDS evaluation through reliable flow-level representations.

These requirements align closely with themes in the dataset-generation literature, especially reproducibility, extensibility, and label quality [21, 17, 10]. They are revisited in Chapter 5, where their implementation support, evaluation evidence, and remaining limitations are discussed.

3.2 Final System Architecture

The final implemented system is a replay-centred framework composed of topology extraction, environment generation, packet replay, capture, and labelling. Several pipeline stages store labelled flow outputs and merged datasets as Comma-Separated Values (CSV) files. Table 3.1 summarizes the main software components.

Table 3.1: Main software components of the implemented framework.

Component	**Main role**	**Main outputs or effects**
`extract_topology.py`	Parses the source PCAP, identifies communicating endpoints, DNS names, DHCP metadata, and inferred host roles, then creates the simulated topology specification.	`topology.json`, `simulated_topology.json`
`generate_compose.py`	Generates the Docker Compose environment from `simulated_topology.json`, including networks, services, static Internet Protocol (IP) addresses, gateway routing, and helper route containers.	Auto-generated `docker-compose.yml` and Docker network definitions
`replay_traffic.py`	Rewrites original packets into the simulated topology, replays them from the correct emulated source containers, captures gateway traffic, cleans captures, and records replay metadata.	Gateway capture files, cleaned replay PCAP, and replay ground-truth rows
`set_delay.sh`	Applies, removes, and lists directional gateway delay rules between simulated source and destination IP addresses using Linux `tc` and `netem`.	Gateway egress delay rules stored for replay timing integration
`ground_truth_base.py`	Adds ground-truth rows for original PCAPs without replay by reading the first and last packet timestamps from the source PCAP.	Base ground-truth rows with `_base` sample identifiers
`label_zeek.py`	Labels Zeek `conn.log` records using completed ground-truth time windows and the `traffic_label` field.	`labeled_conn_*.csv` files

Component	Main role	Main outputs or effects
`map_packets_to_flows.py`	Maps packets from a PCAP to labelled Zeek flows using 5-tuple matching and timestamp overlap.	`packet_flow_map.csv` for packet-to-flow traceability
`merge_datasets.py`	Merges labelled Zeek CSV files into either a training or testing dataset and stores the originating filename for debugging and traceability.	`train_dataset.csv` or `test_dataset.csv`
`train_xgboost.py`	Trains and evaluates a binary XGBoost baseline using numeric Zeek flow features from the merged dataset.	Accuracy, precision, recall, F1-score, and confusion-matrix counts for benign versus malicious flow detection
`run_pipeline.py`	Provides the interactive end-to-end workflow for base labelling, replay, optional random gateway-delay generation, Zeek labelling, dataset merging, and ML training.	Labelled base/replay CSV files, merged dataset, and XGBoost baseline output

Figure 3.1 shows the end-to-end workflow that connects the implemented components from the original PCAP input to labelled Zeek flows and ML validation.

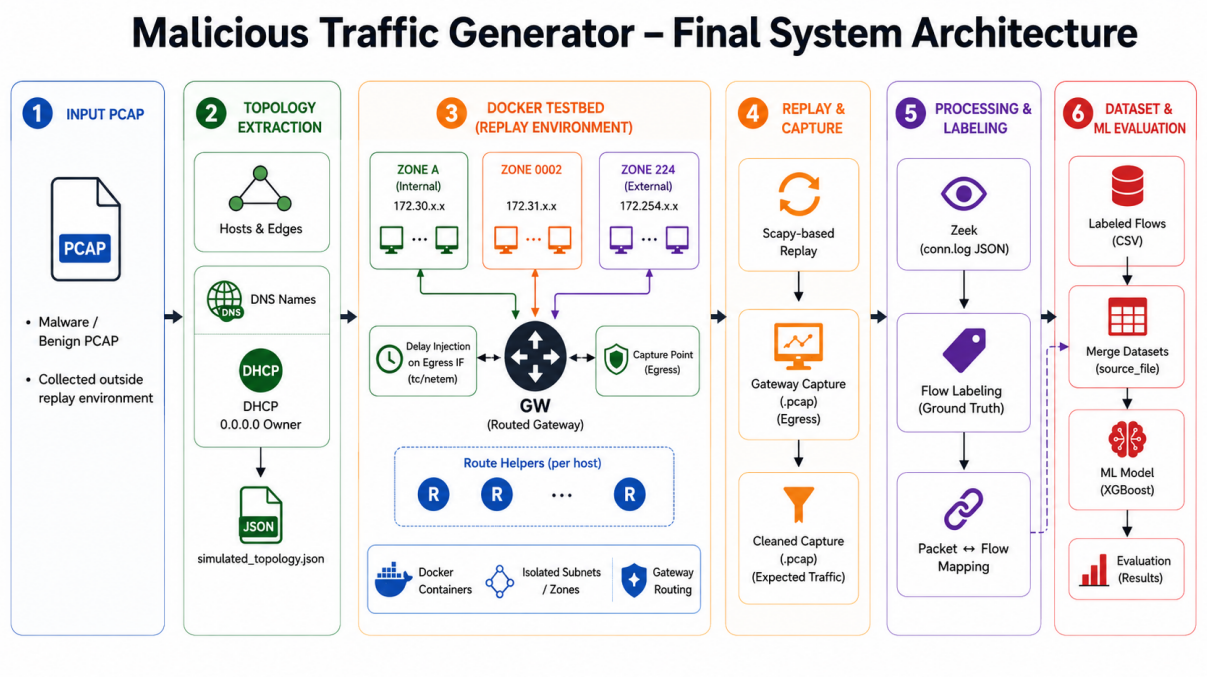


Figure 3.1: End-to-end dataset generation pipeline from original PCAP to labelled Zeek flows and ML baseline.

3.3 Development History and Design Decisions

The final architecture described in the previous section was the result of an iterative development process. The development history is included here only to explain the main design decisions that shaped the final replay framework, rather than to document every intermediate implementation step.

The earliest experiments used a small manually configured container topology with clients, a server, an attacker, and packet capture. These experiments showed that controlled traffic generation and labelling were possible, but they also revealed that host-side capture was not a reliable long-term observation method. Packet visibility depended too strongly on where capture was performed, and duplicate or unrelated packets complicated later analysis. This motivated the move toward a routed gateway architecture where traffic could be observed at a single controlled point.

The gateway architecture then became the central design element of the framework. By forcing communication through a routed gateway, the system could centralize packet capture, routing control, and later delay injection. At the same time, the topology model evolved from manually configured zones into automatically generated replay zones. This was necessary because larger PCAP traces could contain many external address groups, which made the earlier manually named zone structure too limited for replay-based topol-

ogy generation.

A later development stage explored service-aware reconstruction using local DNS and Hypertext Transfer Protocol (HTTP) responder services. This showed that selected application-layer behaviour could be emulated inside the isolated environment. However, it also showed that reproducing complete live malware behaviour was unreliable, especially when malware depended on external infrastructure, dynamic responses, or operating-system-specific runtime behaviour. This reinforced the decision described in Section 3.1 to focus on replaying externally captured PCAP traces instead of executing malware during dataset generation.

The replay mechanism also evolved during development. Early replay attempts used the tcpreplay toolchain, which was useful for establishing an initial baseline but did not provide enough control over source placement, per-packet rewriting, and topology-aware emission. The final implementation therefore moved to Scapy-based replay, where packets could be rewritten into the simulated address space, emitted from the correct source containers, routed through the gateway, and captured again after replay.

These development steps led directly to the final system design: automatic topology extraction, deterministic Docker Compose generation, source-aware packet rewriting, routed gateway capture, optional delay injection, ground-truth generation, Zeek labelling, and packet-to-flow provenance mapping. The following sections describe these final components in detail.

3.3.1 Topology Extraction

The topology-extraction stage begins from a source PCAP obtained from an external sandbox or prior controlled execution. TShark was used as the main extraction utility because it can read previously stored capture files and expose protocol-level fields for further processing [25]. The purpose of this stage is not to reproduce the original environment exactly, but to reconstruct its communication structure in a form suitable for deterministic local replay.

The implemented extractor reads source and destination Internet Protocol (IP) addresses together with protocol information from the PCAP and builds two main internal structures: a host inventory and a communication-edge list. The host inventory records all observed IP addresses and assigns each address an inferred role, such as internal host, external host, broadcast, multicast, or DHCP zero-source traffic. The edge list records observed source–destination communication pairs, packet counts, and protocols. DNS

query names are also extracted so that domain information observed in the original trace can be preserved in the simulated topology.

A specific case handled by the extractor is Dynamic Host Configuration Protocol (DHCP) traffic using the source address `0.0.0.0`. Such packets cannot be treated as normal hosts, because `0.0.0.0` is not a real replay container address. Instead, the extractor attempts to infer the real DHCP client by matching DHCP transaction identifiers and assigned addresses. The resulting metadata is stored so that the replay stage can emit DHCP discover/request packets from the correct simulated container while still preserving `0.0.0.0` as the packet source where appropriate.

The extractor writes two JavaScript Object Notation (JSON) artefacts. First, `topology.json` stores the observed PCAP structure, including hosts, communication edges, DNS names, and DHCP ownership metadata. Second, `simulated_topology.json` maps the observed addresses into the synthetic Docker replay environment. Private IP addresses are mapped to the internal Zone A, while external IP addresses are mapped to generated external zones such as `Z0002`, `Z0003`, and so on. External hosts are grouped according to the first two octets of their original IP addresses, and each external address group is assigned its own replay zone. This design was chosen because the simulated topology uses deterministic static IP assignment, where hosts belonging to the same replay zone share the same first two octets in their generated addresses. As a result, the grouping strategy preserves coarse external network separation from the original PCAP while keeping the replay topology deterministic and reproducible across repeated experiments. Internal hosts are assigned addresses in `172.30.x.x`, from `172.30.11.11` to `172.30.253.253`, supporting up to 243 internal hosts. External hosts are assigned from `172.31.11.11` up to `172.254.253.253`, supporting 224 external zones with 243 host positions each, for a maximum of 54,432 external hosts. For each active host, the corresponding gateway address uses the same subnet with host address `.254`, for example `172.30.11.254` or `172.31.11.254`. If the internal or external address range is exhausted, topology generation stops with an explicit error. This limitation is a consequence of the static IP-allocation strategy, which was chosen to make replay addressing deterministic and reproducible. Each active mapped host receives a simulated IP address, gateway IP address, zone identifier, and service name.

Broadcast and multicast addresses are not instantiated as Docker containers. They are classified as ignored topology entries because creating containers for such addresses would add noise and provide little value for later anomaly-detection experiments. However, they are still retained as metadata where needed for replay correctness. Similarly, DHCP zero-source traffic is kept as metadata rather than as a standalone container. The extraction step therefore functions as a translation layer between a raw PCAP trace and

a replayable experiment specification. Figure 3.2 illustrates how the topology extraction stage translates packet-level observations from the original PCAP into a replayable simulated topology specification.

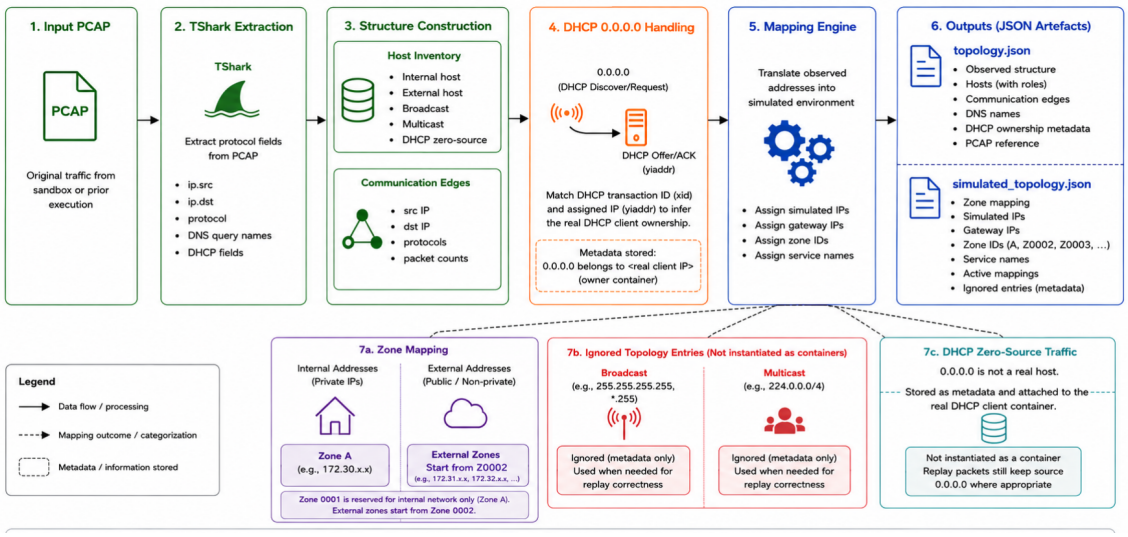


Figure 3.2: Topology extraction and translation pipeline from original PCAP traffic into a replayable simulated topology specification.

3.3.2 Network Synthesis and Zone-Based Routing

The environment-generation stage uses Docker Compose and user-defined bridge networks to instantiate the synthetic topology. Docker bridge networks are suitable here because they provide scoped communication within a single host and isolate containers that do not share a network [6]. Docker Compose complements this by letting services be attached explicitly to named networks with specific addressing plans and internal isolation properties [7]. Although the final workflow normally derives the environment from `simulated_topology.json`, the generator can also be used with a manually prepared topology file. This allows the user to define or adjust the synthetic network structure directly when automatic topology extraction is not desired or when a controlled custom scenario is needed.

The generated environment is derived directly from `simulated_topology.json`. Only active mappings are instantiated as Docker containers. Entries classified as `ignore`, `gateway`, or `dhcp_zero` are not created as standalone containers, because broadcast and multicast addresses are not real hosts and DHCP zero-source traffic is handled as replay metadata. This prevents unnecessary containers from being created while still preserving replay-relevant metadata.

For each active mapped host, the generator creates a dedicated Docker bridge network

and assigns a static simulated IP address. The corresponding gateway address is attached to the central gateway container. Internal hosts are placed in Zone A, while external hosts are placed in separate external zones named `Z0002`, `Z0003`, and so on. This design means that each replay host is isolated on its own network segment and must communicate through the gateway rather than directly at layer 2.

The generated Compose file contains a central gateway service named `master-thesis-gw`. This gateway is connected to all generated networks, has IP forwarding enabled, and is given the required network capabilities for routing and packet processing. Each host service is also accompanied by a small route-helper container that shares the host network namespace and replaces the default route with a route through the gateway. This ensures that all inter-zone traffic follows the controlled gateway path.

The gateway therefore became more than a router. It served as the central routing point, the observation point, the later delay-injection point, and the anchor that made the topology behaviour predictable across runs. The generator also includes an optional capture service under a Docker Compose profile. This capture service shares the gateway network namespace and can record traffic from all generated subnets into a PCAP file, although the final replay workflow usually starts its own gateway captures through `replay_traffic.py`. Figure 3.3 shows the generated zone-based Docker topology in which simulated hosts communicate through the central routed gateway.

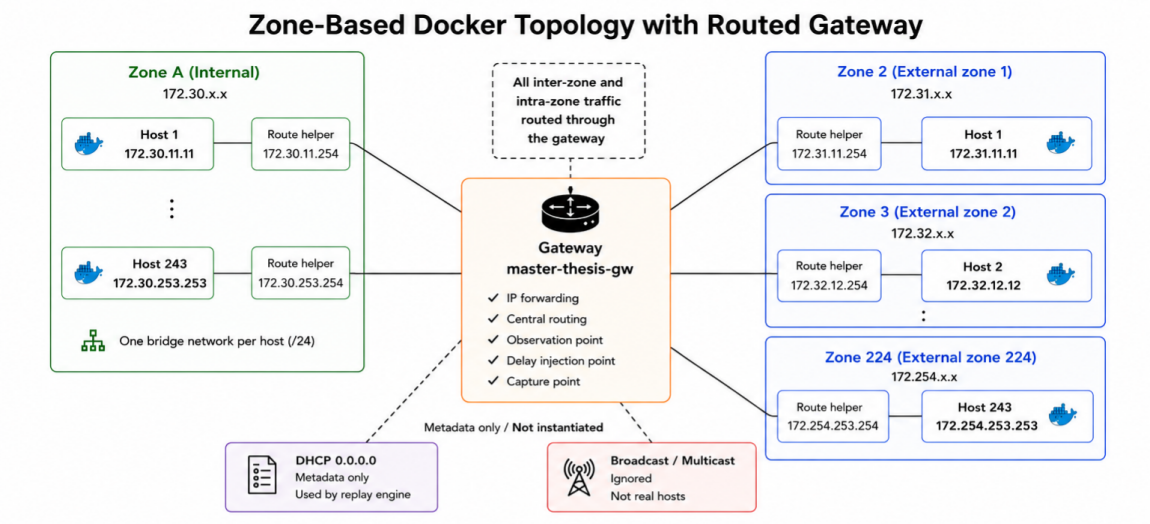


Figure 3.3: Zone-based Docker replay topology with routed gateway.

3.3.3 Delay Injection and Timing Control

Controlled delay was introduced to simulate varied network conditions and to support repeated replay experiments under different timing assumptions. Linux `tc` provides the

general traffic-control interface, while `netem` provides network-emulation functions such as delay and other impairments [13, 12]. In this framework, delay is applied at the routed gateway rather than inside individual hosts, because all inter-zone traffic is forced through the gateway. This makes the gateway a single controllable point where directional timing changes can be introduced consistently.

Delay rules are managed by `set_delay.sh`. The script supports three operations: adding a delay rule, deleting a rule, and listing currently installed rules. A rule is defined by a simulated source IP address, a simulated destination IP address, and a delay value in milliseconds. Before installing a rule, the script validates that Docker is available, that the gateway container is running, that both IP addresses are valid, that the source and destination are different, and that both addresses exist in the generated Docker topology.

When a delay rule is added, the script first determines the correct gateway egress interface for the destination address using the gateway routing table. It then creates or reuses a Hierarchical Token Bucket (HTB) traffic-control hierarchy on that interface and attaches a `netem` delay queue to a class representing the selected source–destination direction. A `tc` filter matches packets by source and destination IP address and sends only that direction of traffic into the delayed class. This makes the delay directional: a rule for `SRC -> DST` does not automatically delay traffic in the reverse direction. The script also stores each configured delay value inside the gateway container under `/tmp/replay_delay_rules`. This metadata is used by the replay logic so that the Scapy-based sender can account for configured directional delays.

3.3.4 Replay Engine Evolution

Replay Strategy

As discussed in Section 3.1, the framework shifted from live malware execution toward replay-based reconstruction in order to improve reproducibility, containment, and network control. The replay engine therefore focused on reconstructing previously captured PCAP traffic inside the generated topology rather than reproducing the full malware runtime environment.

This design decision shaped the remaining replay implementation: packets needed to be rewritten into the simulated address space, emitted from the correct source containers, routed through the gateway, captured again after replay, and exported in a form suitable for Zeek-based labelling and flow-level analysis.

Tcpreplay and Tcprewrite

The first packet-replay implementation used tcpreplay together with tcprewrite. This toolchain was attractive because tcpreplay is specifically designed for replaying previously captured traffic, and tcprewrite can modify packet headers and checksums before replay [1, 2]. In practice, this stage helped establish an initial replay baseline and supported experiments with packet editing, timing preservation, and interface-level replay.

However, the early replay model was still relatively centralized and replayed traffic from a limited sender perspective. As the framework evolved toward a topology-aware multi-host environment, the replay process required finer-grained control over where packets originated and how they were injected into the simulated network. The thesis therefore needed packets to be emitted from the correct emulated hosts and zones rather than replayed as a single monolithic stream through one interface. This requirement became increasingly important once replay traffic had to preserve distributed communication structure across multiple simulated containers and routed zones.

These limitations gradually motivated the transition toward a Scapy-based replay approach, where packets could be rewritten, scheduled, and transmitted independently from the appropriate emulated hosts inside the generated topology.

Scapy-Based Source-Aware Replay

For that reason, the replay path later shifted to Scapy. Scapy supports programmatic packet construction, modification, and transmission directly from Python, making it substantially better suited for topology-aware replay and per-packet decision logic than the earlier tcpreplay-based approach [19]. The earlier replay model treated traffic more as a single stream emitted from a replay interface, while the final framework required distributed replay where packets originated from the correct simulated hosts inside the generated Docker topology.

The replay engine therefore evolved into a source-aware system where each replayed packet is associated with a mapped sender container derived from `simulated_topology.json`. Before replay begins, the framework rewrites the original PCAP into a topology-consistent intermediate replay PCAP. Original source and destination IP addresses are translated into their simulated addresses according to the generated topology mappings. Address Resolution Protocol (ARP) addresses are rewritten as well so that local neighbour-discovery traffic remains consistent inside the synthetic environment.

Several problems emerged during this stage. One major challenge was that packets

copied directly from the original trace often became invalid after rewriting because protocol checksums, packet lengths, and Ethernet metadata no longer matched the modified topology. To solve this, the replay engine removes stale checksums and length fields and lets Scapy recalculate them automatically after rewriting. Ethernet padding is also added where necessary so that replayed Ethernet frames satisfy minimum frame-size requirements.

Another challenge concerned Ethernet-layer correctness. Docker containers dynamically generate their own Media Access Control (MAC) addresses, meaning the original Ethernet headers change. The framework therefore reconstructs Ethernet source and destination MAC addresses dynamically during rewriting. Sender MAC addresses are obtained directly from the corresponding replay containers, while destination MAC addresses are determined according to the replay context.

Ethernet-layer handling required additional replay logic because the original MAC addresses captured in the source PCAP did not belong to the generated Docker environment. During replay, normal routed unicast traffic is therefore directed toward the gateway MAC address, while broadcast traffic uses the Ethernet broadcast address `ff:ff:ff:ff:ff:ff`. Multicast traffic is translated into the corresponding multicast MAC ranges, and selected discovery protocols such as NetBIOS Name Service (NBNS), multicast DNS (mDNS), Link-Local Multicast Name Resolution (LLMNR), and Simple Service Discovery Protocol (SSDP) use explicit protocol-aware Ethernet destination mappings during replay.

Broadcast and multicast traffic also introduced replay problems after topology translation. Many packets in the original trace referenced broadcast domains and subnet addresses that no longer existed once the traffic had been mapped into the simulated Docker topology. To solve this, subnet broadcast destinations are rewritten into the corresponding simulated subnets while preserving global broadcast addresses such as `255.255.255.255`. This allows local discovery and broadcast behaviour to remain meaningful inside the replay environment even though the original network structure has been replaced by the synthetic zone-based topology.

Special handling was required for DHCP traffic using the source address `0.0.0.0`. Such packets cannot be mapped to a normal replay container directly because `0.0.0.0` is not a valid host address. Earlier replay attempts either dropped these packets or replayed them incorrectly. The final implementation instead infers the real DHCP client owner during topology extraction and associates the replay traffic with that simulated host while still preserving `0.0.0.0` as the visible packet source when appropriate.

TCP replay consistency became another major challenge after topology rewriting. Packet modification could introduce inconsistent TCP sequence numbers, acknowledgement values, and payload-length relationships. In some experiments this caused replayed streams to become partially invalid or inconsistent for downstream processing tools. To mitigate this, the framework performs sequence and acknowledgement normalization after packet rewriting. Payload lengths, Synchronize (SYN) flags, and Finish (FIN) flags are used to reconstruct a consistent TCP progression across rewritten packets before replay transmission begins.

Replay execution itself is distributed across multiple temporary sender processes. For each active emulated host, the framework launches a dedicated Scapy replay container attached to that host's Docker network namespace. During replay, packets are transmitted sequentially in original PCAP order, but each packet is emitted from the correct simulated sender container and interface. This preserves source placement and distributed communication structure across the replayed topology instead of replaying the entire trace from one generic interface.

Another practical problem was that the replay containers themselves could generate unwanted traffic during replay. Linux networking stacks automatically emit TCP reset packets, Internet Control Message Protocol (ICMP) destination-unreachable messages, and other local responses when replayed traffic does not correspond to active application sockets inside the container. These packets polluted early replay captures and interfered with later Zeek processing. The framework therefore installs temporary firewall rules inside the replay containers and their associated routing namespaces to suppress locally generated TCP reset and ICMP error traffic during replay execution. This prevents the container networking stacks from injecting unintended responses that would otherwise contaminate the replay traffic and later gateway captures.

Replay timing also evolved significantly during development. Early implementations simply preserved the original inter-packet gaps from the source PCAP. However, this became insufficient once directional gateway delay rules were introduced. In the final implementation, replay timing combines both the original PCAP timing and optional configured gateway delay rules while preserving the original packet ordering. During replay, packets remain globally ordered according to the original trace, and the sender preserves the original inter-packet gaps recorded in the PCAP. When packet direction changes, the replay engine consults the directional delay metadata generated by `set_delay.sh` and adds the configured gateway delay before transmission. Packet replay itself remains globally ordered according to the original trace, but the additional delay allows request–response timing to better reflect the configured gateway latency between simulated endpoints.

Because the delay is applied separately in each communication direction, the resulting behaviour approximates a configurable round-trip time (RTT) between the participating hosts. This prevents replayed communication from becoming unrealistically compressed in time, for example when later packets depend on earlier DNS responses or connection-establishment exchanges.

Gateway capture and replay validation also became important parts of the final replay pipeline. Traffic is captured again from the gateway egress interfaces during replay so that the final exported PCAP reflects the actual routed replay traffic rather than only the rewritten packets prepared before transmission. However, raw gateway captures contained significant noise, including unrelated container-management traffic, duplicated packets, and missing discovery packets. To solve this, the framework performs a replay-cleaning stage after capture.

During cleaning, replayed packets are matched against the expected rewritten packets using protocol-aware signatures derived from IP addresses, transport-layer fields, TCP state information, DNS metadata, and payload hashes. Only packets matching the expected replay traffic are retained in the final capture. Some expected broadcast and discovery packets that do not reliably appear in gateway captures are reinserted using timestamp estimation derived from neighbouring observed packets. Additional filtering removes gateway-management ARP traffic and unrelated container communication. Finally, packet timestamps are normalized to ensure monotonically increasing timing inside the exported replay PCAP.

The result is a replay pipeline that not only replays packets from the correct simulated hosts, but also reconstructs topology-aware addressing, timing behaviour, directional delay, Ethernet-layer correctness, and cleaned routed captures suitable for later Zeek-based labelling and IDS dataset generation. Figure 3.4 illustrates the final Scapy-based replay path, including packet rewriting, source-aware emission, optional gateway delay, and cleaned gateway capture.

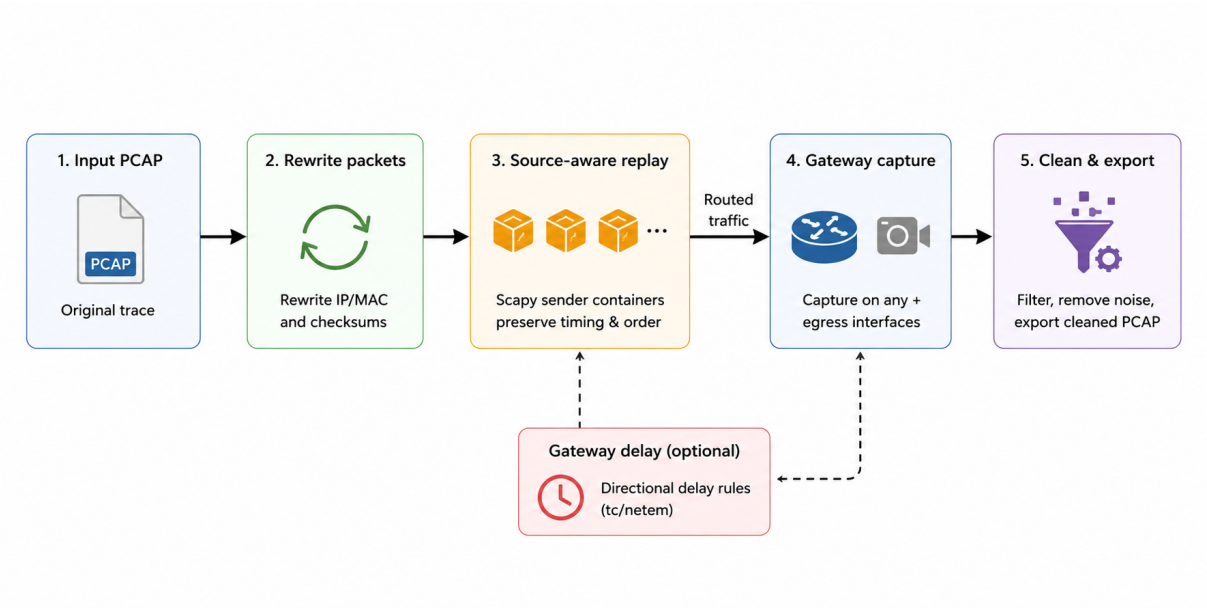


Figure 3.4: Packet rewriting, source-aware Scapy replay, optional gateway egress delay, and cleaned gateway capture.

3.3.5 Capture Strategy

Reliable capture was critical because all downstream labelling depends on packet visibility. Capture therefore evolved through several stages: host-mode capture, server-side capture, gateway capture, and finally refined multi-interface gateway capture.

The final implementation produces three related capture outputs:

1. a broad gateway capture on interface `any`, useful for observing the routed replay globally and debugging replay behaviour;
2. multiple gateway egress-interface captures using `tcpdump -Q out`, useful for observing outbound routed traffic separately on each gateway interface;
3. a cleaned gateway-egress replay capture created by filtering and combining the interface-specific captures so that the final PCAP more closely matches the intended replay traffic.

The need for multiple capture forms was discovered empirically. Capturing on broad interfaces such as `any` can introduce duplicate observations because packets may be visible multiple times while traversing Docker bridge networks and the routed gateway. In addition, captures on `any` may use Linux cooked-capture headers rather than normal Ethernet framing, which can affect packet-size comparisons and replay validation.

To address these issues, the final workflow uses interface-specific outbound captures as

the primary source for the cleaned replay PCAP. The replay pipeline filters duplicate observations, preserves Ethernet framing, corrects MAC-address handling, and removes unrelated gateway-management traffic before exporting the final replay capture. These refinements improved consistency between original and replayed traffic and made the resulting PCAPs more suitable for comparison and later Zeek processing.

These replay-time captures are distinct from the optional static capture service defined in the generated Docker Compose environment. The Docker Compose capture container provides a general-purpose background packet capture attached to the gateway namespace and is mainly intended for broad experiment observation or debugging. In contrast, the replay framework dynamically starts dedicated replay-time captures on the gateway `any` interface and on individual outbound gateway interfaces specifically for replay validation, filtering, duplicate reduction, and cleaned replay-PCAP generation.

The use of dedicated outbound gateway-interface captures was an intentional architectural decision. By capturing traffic separately on each gateway egress interface, the framework obtains deterministic visibility into routed replay traffic after rewriting, routing, and optional delay injection have been applied. However, this design also introduces a scalability tradeoff. The replay framework starts a separate `tcpdump` subprocess for each gateway egress interface, meaning that larger replay topologies generate many simultaneous capture processes. As the number of simulated hosts increases, Central Processing Unit (CPU), memory, and process-management overhead also increase, making large replay environments more difficult to scale reliably. Figure 3.5 shows the routed gateway capture design used to observe replayed traffic after rewriting, routing, and optional delay injection.

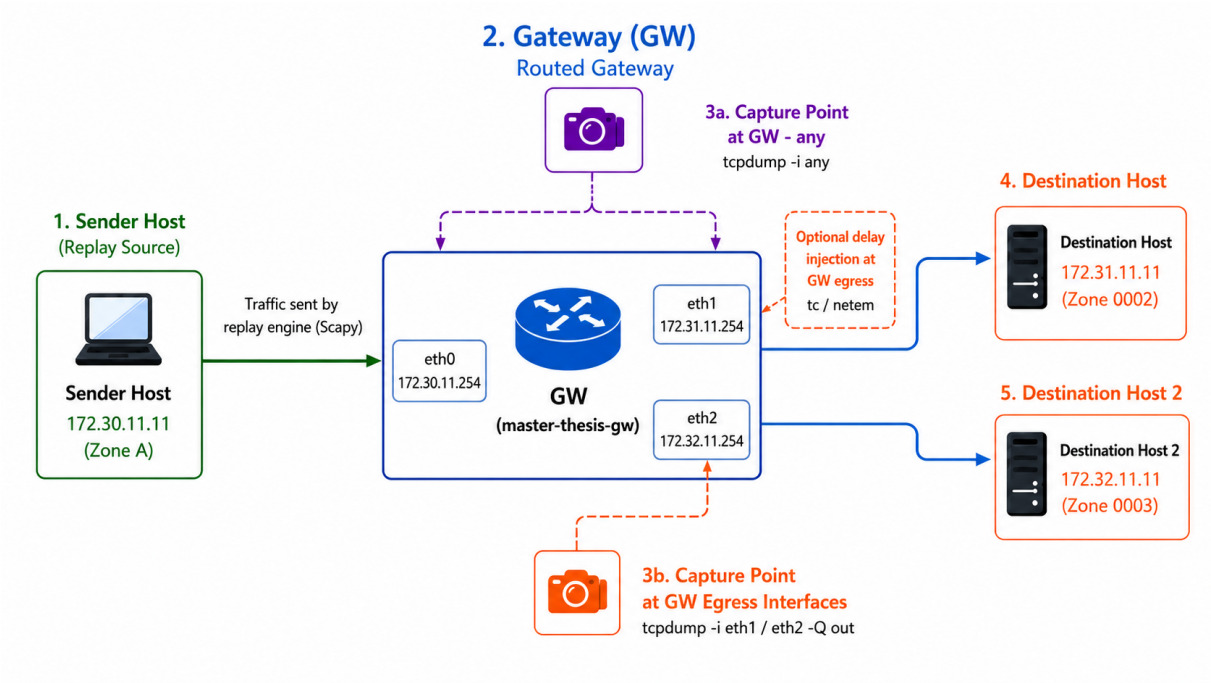


Figure 3.5: Routed gateway forwarding and multi-point gateway capture inside the simulated topology.

3.3.6 Ground Truth and Label Generation

A defining property of the framework is that labels are not inferred after the fact from packet contents alone. Instead, explicit experiment metadata is written to `ground_truth.csv` and later used as the authoritative source for labelling. The ground-truth file records the sample identifier, traffic label, replay or capture time window, replay multiplier, execution status, and optional notes.

The framework supports two ground-truth creation paths. For original PCAPs that are analysed without replay, `ground_truth_base.py` reads all packet timestamps from the PCAP using TShark's `frame.time_epoch` field. It then writes a completed base row using the first and last observed packet times as the labelling window. These rows use a `sample_id` ending in `_base`, a replay multiplier of `1.0`, and a user-selected `traffic_label` of either `benign` or `malicious`.

For replayed traffic, `replay_traffic.py` appends a ground-truth row after each replay attempt. The row records the replay start and end timestamps in Coordinated Universal Time (UTC), the replay multiplier, the selected traffic label, the execution status, and any notes or error information. If the same source PCAP is replayed multiple times, the script automatically versions the `sample_id` so that repeated executions remain distinguishable, for example when evaluating different optional gateway-delay configurations.

In both cases, `execution_id` values are regenerated sequentially when the ground-truth file is written, which keeps the metadata consistent as new rows are appended. Table 3.2 summarizes the current fields used by the uploaded scripts.

Table 3.2: Core fields used in replay ground truth.

Field	Meaning
`execution_id`	Sequential identifier assigned when the ground-truth file is written
`sample_id`	Identifier of the source PCAP; original-PCAP rows use the `<pcap_stem>_base` convention
`traffic_label`	Binary traffic label, currently `benign` or `malicious`
`replay_start_time_utc`	Start timestamp of the original PCAP window or replay execution
`replay_end_time_utc`	End timestamp of the original PCAP window or replay execution
`replay_multiplier`	Replay timing multiplier; base rows use `1.0`
`status`	Row status; only `completed` rows are used by `label_zeek.py`
`notes`	Optional notes, for example `original_pcap` for base rows

This ground truth supports two later stages. First, Zeek is run on the captured PCAP to generate `conn.log` and related artefacts. Zeek's connection logging is especially useful because it summarizes communication records in a form appropriate for later flow-based analysis [27]. Second, `label_zeek.py` assigns labels to these connections by comparing each Zeek flow start/end interval with completed ground-truth windows. If a Zeek connection overlaps a completed malicious or benign ground-truth interval, the corresponding `traffic_label` is assigned to that flow. Connections that do not overlap any completed interval are labelled benign by default. Figure 3.6 shows how ground-truth metadata are connected to Zeek connection logs in order to produce labelled flow-level outputs.

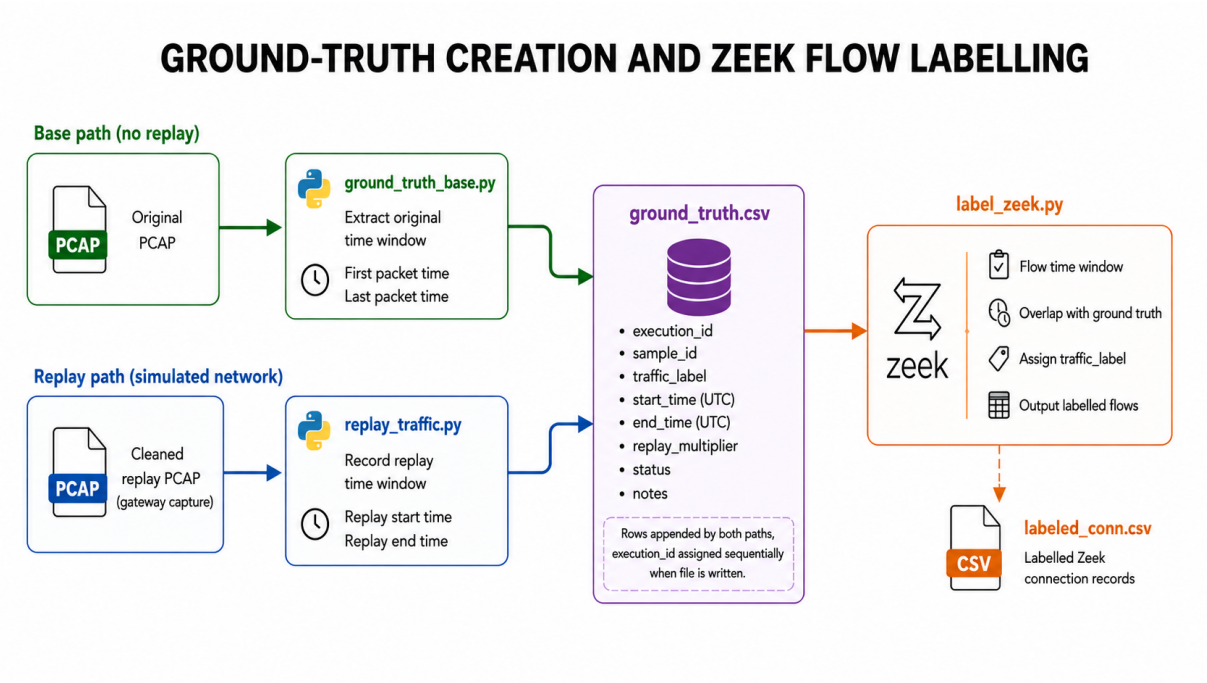


Figure 3.6: Ground-truth generation and Zeek-based flow labelling pipeline.

3.3.7 Role of `map_packets_to_flows.py`

In addition to direct Zeek labelling, the pipeline includes `map_packets_to_flows.py`. This script addresses a common but often under-documented problem in IDS dataset engineering: after packets have been aggregated into Zeek connection records, it can become difficult to trace a labelled flow back to the packets that produced it. The script therefore provides an audit layer between packet-level replay evidence and flow-level labels.

The script reads a PCAP file and a labelled Zeek connection CSV file. For each IP packet in the PCAP, it extracts the protocol, source and destination IP addresses, and transport ports when TCP or UDP is present. It then builds a normalized bidirectional flow key so that packets are matched to the same Zeek connection regardless of traffic direction. The same normalized key is built from Zeek fields such as `id.orig_h`, `id.orig_p`, `id.resp_h`, `id.resp_p`, and `proto`.

Because a single 5-tuple may appear in more than one Zeek row over time, the script also checks whether each packet timestamp falls within the Zeek flow time interval. The interval is computed from the Zeek `ts` and `duration` fields, with a configurable slack value to tolerate small timing differences. If multiple matching flows are possible, the default behaviour selects the candidate whose start time is closest to the packet timestamp. An optional `-all-candidates` mode can instead export all matching candidate flows.

The output is `packet_flow_map.csv`. Each row records the packet number, packet timestamp, packet source and destination, protocol, ports, matched Zeek row number, Zeek `uid`, sample identifier, flow start time, and duration. Packets that cannot be matched to any Zeek flow are still written to the output with the label `unmatched`. This makes the file useful for debugging missing flows, checking label propagation, and verifying whether replayed packets are represented correctly in the flow-level dataset.

The output is not intended to be the main ML feature table. Instead, it acts as a provenance and validation artefact that links cleaned replay captures to the labelled Zeek flows used downstream. This strengthens the defensibility of the labelling process because flow labels remain connected to observable packet-level evidence rather than being treated as disconnected derived records.

3.4 Validation Strategy and Deferred ML Evaluation

The framework was validated through two complementary perspectives. First, engineering validation was used to verify that the main pipeline stages operated correctly and reproducibly, including topology extraction, Docker environment generation, source-aware replay, gateway routing, optional delay handling, packet capture, Zeek processing, and flow labelling. Validation focused on whether replayed traffic preserved the expected communication structure, timing behaviour, and replay metadata required for later analysis.

Second, a lightweight ML-based validation stage was included to evaluate whether the generated replay data retained useful properties for downstream intrusion-detection tasks. The purpose of this stage was not to build a production-ready IDS model, but rather to test the practical utility of the generated dataset.

The ML workflow begins after Zeek processing and flow labelling. Labelled Zeek connection CSV files matching the `labeled_conn_*.csv` pattern are merged by `merge_datasets.py`. During execution, the user specifies whether the merged output should be generated as a training dataset or a testing dataset. The script then automatically exports the result as either `train_dataset.csv` or `test_dataset.csv`. During merging, the originating filename is also stored in a dedicated `source_file` column so that individual replay runs remain traceable during debugging and evaluation.

The resulting merged dataset is then processed by `train_xgboost.py`. The current implementation uses a binary XGBoost classifier trained on a small set of numeric Zeek flow features, including connection duration, packet counts, byte counts, IP byte counts, missed bytes, and transport protocol identifiers. Missing values are normalized to zero

before training. The binary target variable is derived directly from the Zeek flow label, where all non-benign labels are mapped to the malicious class. The script expects separate `train_dataset.csv` and `test_dataset.csv` files; therefore, the train–test split is created before training by the dataset-generation workflow rather than inside `train_xgboost.py`. XGBoost was selected because it is well suited for structured tabular flow data and provides a strong practical baseline without requiring a complex deep-learning pipeline [4].

The ML stage therefore functions primarily as a dataset-utility evaluation rather than as the central contribution of the thesis. If models trained on replay-generated data perform comparably to those trained on original captures, this suggests that the replay framework preserved important flow-level behavioural properties. Similarly, if combined datasets remain competitive or improve classification behaviour, this indicates that replay-generated traffic can contribute additional useful variation for later IDS experimentation. Figure 3.7 illustrates the ML-validation workflow used to merge labelled Zeek flows and evaluate them with the XGBoost baseline.

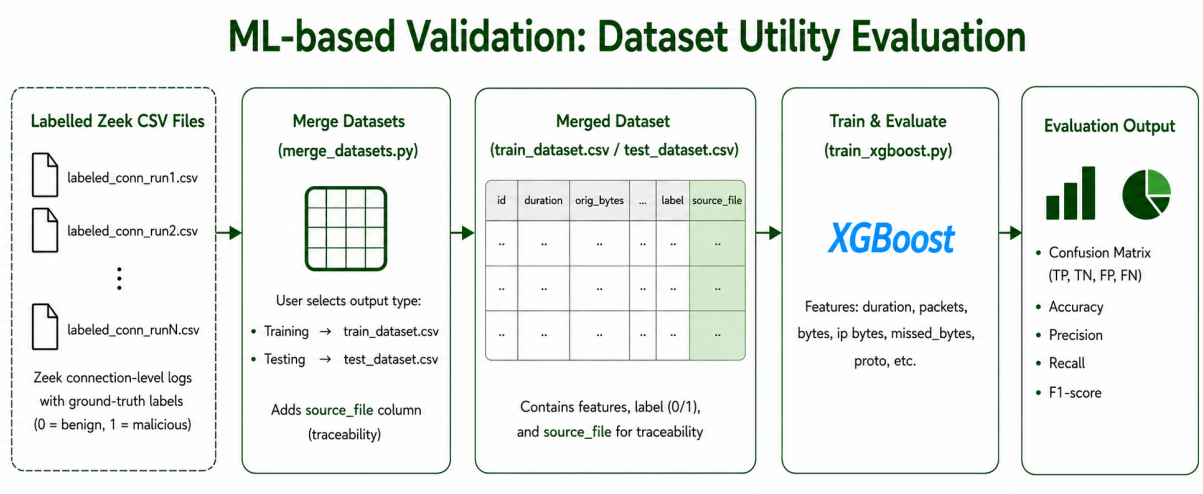


Figure 3.7: ML-based validation workflow for evaluating the utility of labelled Zeek flow datasets generated from replayed and original PCAP traffic.

3.5 Dataset Orchestration

The complete user-facing workflow is coordinated by `run_pipeline.py`. The script acts as an interactive orchestration layer that connects topology extraction, Docker environment generation, replay execution, capture processing, Zeek analysis, dataset merging, and ML training into one reproducible workflow. Rather than exposing every script individually, the pipeline provides a small command-oriented interface that guides the user through the required stages while still keeping each processing step explicit and inspectable.

In its current form, the orchestration layer provides short command aliases for the main actions. The `base(b)` mode labels an original PCAP directly without replay by generating a base ground-truth entry, running Zeek on the original capture, and applying flow labels. The `replay(r)` mode performs the full replay workflow: topology extraction, Docker Compose generation, container startup, optional gateway-delay configuration, Scapy-based replay, gateway capture generation, Zeek processing, and flow labelling. Additional modes support later dataset stages: `merge(mg)` merges multiple labelled Zeek CSV files into either a training or testing dataset, `train(t)` trains the XGBoost baseline model on the merged dataset, and `both(bt)` executes merging followed immediately by training.

The replay workflow also integrates several usability and validation mechanisms. Before replay, the script can load the generated `simulated_topology.json` file and display all active simulated hosts together with their simulated IP addresses, original IP addresses, service names, and topology roles. The user can then choose whether to apply random gateway-delay rules before replay. If enabled, the user specifies the number of delay rules and the minimum and maximum delay values in milliseconds. The orchestration layer then randomly selects valid simulated source–destination pairs and applies the generated rules through `set_delay.sh`. These delay rules are applied before replay begins and later influence the replay timing logic.

The orchestration layer also automates repeated operational tasks that were previously error-prone when executed manually. It generates consistent filenames for replay captures and labelled outputs, manages Docker lifecycle operations, launches Zeek processing automatically after replay completion, and optionally shuts down the generated topology when processing is finished. This interactive orchestration replaces the earlier dataset-runner workflow used during development and reduces manual command errors while still preserving transparency between the individual processing stages.

4

Results

4.1 Experimental Environment

All experiments were executed on a Lenovo laptop running Windows 11 Home directly on the physical machine, with the replay framework executed inside Ubuntu 24.04.3 Long-Term Support (LTS) on Windows Subsystem for Linux 2 (WSL2). Docker Desktop was used to run Linux containers from the WSL2 environment. The experiments therefore used a local single-machine setup rather than a distributed or cloud-hosted testbed.

Table 4.1: Experimental host and software environment.

Item	Configuration
Physical host	Lenovo 83CV laptop running Windows directly on bare metal
Host operating system	Microsoft Windows 11 Home, version 10.0.26200, build 26200
Host memory	15,805 MB physical Random Access Memory (RAM)
Experiment environment	Ubuntu 24.04.3 LTS (Noble Numbat) inside WSL2
Kernel	Linux 6.6.87.2-microsoft-standard-WSL2, x86_64
CPU visible to WSL2	Intel(R) Core(TM) Ultra 5 125H, 18 logical CPUs, 9 cores, 2 threads per core
Memory visible to WSL2	7.5 Gibibytes (GiB) RAM and 2.0 GiB swap
Docker Engine	Docker version 29.4.3, build 055a478
Docker server	Docker Desktop, Linux containers, x86_64 architecture
Docker Compose	Docker Compose version v5.1.4
Docker resources	18 CPUs and approximately 7.48 GiB memory available to Docker
Docker cgroup driver	`cgroupfs`
WSL2 network interface	`eth0` with address `172.23.181.180/20`
WSL2 default route	Default gateway `172.23.176.1` via `eth0`
Network configuration	WSL2 Network Address Translation (NAT)-style virtual networking; replay experiments used Docker bridge networks and a generated routed gateway container

The replay experiments were therefore performed in a reproducible local WSL2-based Docker environment. All replay topologies were generated as isolated Docker bridge networks, and inter-zone traffic was forced through the generated gateway container used for routing, capture, and optional delay injection. The PCAP traces used in the experimental evaluation are summarized in Appendix A.

4.2 Overview of the Current Outcome

Within the experimental environment described above, the final outcome of this thesis is a reproducible framework for reconstructing, replaying, capturing, and labelling malicious and benign network traffic from PCAP traces inside a controlled Docker-based environment. The implemented system integrates topology extraction, automated environment generation, packet rewriting, replay orchestration, and gateway-based capture. It also supports delay injection, Zeek flow generation, packet-to-flow provenance mapping, and dataset merging.

As discussed in Section 3.1, the framework evolved from early live-execution trials toward replay-based traffic reconstruction in order to improve reproducibility, containment, and experimental control. The final replay-centred architecture preserved the important communication behaviour of the original traffic while significantly improving reproducibility, automation, containment, and experimental control.

The implemented framework satisfies the primary engineering goals defined earlier in the thesis by providing reproducible malicious-traffic reconstruction, deterministic topology generation, centralized packet visibility, auditable ground-truth generation, configurable timing manipulation, and export of labelled flow-level datasets suitable for later ML-based IDS research. The complete workflow supports the following processing pipeline:

Original PCAP → Topology Extraction → Docker Environment Generation → Replay → Gateway Capture → Zeek Flow Extraction → Flow Labelling → Packet-to-Flow Mapping → Dataset Generation → ML Validation

4.3 Topology Reconstruction and Environment Generation

One of the primary results of the thesis is the successful implementation of automatic topology extraction directly from PCAP traces. The script `extract_topology.py` reconstructs communication structure using TShark field extraction and generates both `topology.json` and `simulated_topology.json`. The extracted topology includes communicating hosts, protocol edges, DNS query names, DHCP ownership metadata, broadcast and multicast handling, and host-role classification.

The final implementation distinguishes automatically between internal and external hosts. Internal hosts are mapped into Zone A (`172.30.x.x`), while external hosts are distributed into isolated routed zones beginning from `172.31.x.x`. This mapping strategy preserves communication relationships from the original traffic while keeping all replay activity fully isolated inside the Docker environment.

The generated replay environment uses a routed gateway-centric design. Each simulated host receives a dedicated Docker network, a deterministic simulated IP address, explicit gateway-based routing, and a route-helper container. As a result, all communication traverses the gateway container, which becomes the centralized observation point for packet capture, routing control, and timing manipulation.

Compared to the earliest manually configured topologies used during the project, the final implementation provides significantly improved automation and scalability. Entire replay environments can now be generated automatically from a single PCAP trace without requiring manual network construction.

4.4 Replay Fidelity and Communication Reconstruction

The replay engine implemented in `replay_traffic.py` successfully reconstructs communication behaviour inside the generated environment while rewriting traffic into the synthetic topology.

The replay implementation evolved substantially during development. Initial experiments relied on Tcpreplay-style approaches, but the final implementation transitioned toward Scapy-based replay because it provided finer-grained control over packet rewriting, MAC-address reconstruction, checksum recalculation, replay timing, DHCP handling, broadcast forwarding, and container-specific replay placement.

The replay engine preserves several important communication properties from the original traces, including source and destination relationships, bidirectional communication structure, timing relationships, protocol behaviour, and replay ordering. Special protocol cases were also handled successfully. DHCP traffic using source address `0.0.0.0` is replayed through ownership reconstruction using DHCP transaction identifiers extracted during topology analysis, while broadcast and multicast traffic are preserved through protocol-aware Ethernet rewriting logic.

The replay framework additionally supports configurable replay-speed multipliers and gateway-based delay injection using Linux `tc` and `netem`. Delay rules can be applied selectively between specific simulated source and destination hosts, allowing controlled timing experiments for future IDS evaluation scenarios.

Another important result is that both benign and malicious traffic can be replayed through the same pipeline without modifying the replay logic itself. Maliciousness is instead represented through replay metadata and ground-truth labelling rather than through execution-specific behaviour inside the replay environment.

The packet-retention rate was computed as the number of packets in the final cleaned replay capture divided by the number of expected replayable packets. The replay-cleaning

script also reports intermediate cleaning counts. In particular, directly observed packets and fallback-added packets are counted before the final ARP/link-layer cleanup step, while the final cleaned replay packet count is measured after that cleanup. Therefore, the intermediate observed and fallback counts are not expected to sum directly to the final cleaned packet count. Packet-to-flow coverage was computed from `packet_flow_map.csv` as the fraction of replayed IP packet rows and replayed Zeek flow identifiers that were successfully matched. The TCP quality metric reports the fraction of replayed TCP Zeek flows where `missed_bytes = 0`.

Table 4.2: Replay-fidelity summary for the 5,000-packet subset of the `2025-06-13-traffic-analysis-exercise.pcap` trace.

Metric	Result
Expected replayable packets	5000
Matched observed packets before final cleanup	4933 / 5000 = 98.66%
Fallback-added packets before final cleanup	67 / 5000 = 1.34%
Packets removed by final ARP/link-layer cleanup	8 / 5000 = 0.16%
Final cleaned replay packets	4992 / 5000 = 99.84%
Packet-retention rate	99.84%
Original Zeek flows	144
Replayed Zeek flows	152
Zeek flow-count ratio	105.56%
Packet-to-flow matched IP packet rows	4978 / 4978 = 100%
Packet-to-flow matched Zeek flows	152 / 152 = 100%
TCP flows with zero missed bytes	75 / 75 = 100%

The difference between the intermediate count (4933 + 67) and the final cleaned packet count is caused by the final ARP/link-layer cleanup stage. The packet-to-flow result is reported separately because Zeek `conn.log` represents IP-level connections; therefore, non-IP/link-layer packets in the cleaned PCAP are not expected to appear as Zeek flow records.

The Kolmogorov–Smirnov (KS) statistic and Jensen–Shannon (JS) divergence were computed from histogram-based empirical feature distributions using 50 equal-width bins over the combined original and replay value range for each feature.

Table 4.3: Distribution-level replay fidelity for Zeek numeric flow features.

Feature	Original mean	Replay mean	KS statistic	JS divergence
`duration`	1.918	9.740	0.491	0.1328
`orig_bytes`	1557.813	1484.895	0.051	0.0005
`resp_bytes`	19108.764	18102.809	0.047	0.0000
`orig_pkts`	14.861	14.079	0.030	0.0002
`resp_pkts`	19.708	18.671	0.049	0.0002
`orig_ip_bytes`	2154.056	2039.697	0.052	0.0007
`resp_ip_bytes`	19891.847	18843.993	0.049	0.0000
`missed_bytes`	0.000	0.000	0.000	0.0000

Tables 4.2 and 4.3 provide quantitative replay-fidelity evidence for the 5,000-packet subset of the `2025-06-13-traffic-analysis-exercise.pcap` trace. The cleaned gateway capture retained 4992 of 5000 expected replayable packets, corresponding to a packet-retention rate of 99.84%. Of these packets, 4933 were directly observed in the cleaned gateway capture, while 67 were restored by the fallback step because selected broadcast or discovery packets did not appear reliably in the gateway egress capture. The packet-to-flow provenance step matched all 4978 IP packet rows to Zeek connection records and covered all 152 replayed Zeek flows. The remaining 14 packets in the cleaned replay capture are link-layer packets such as ARP, which are not represented as Zeek `conn.log` flows. The small difference between the expected replay input and the cleaned gateway capture is mainly explained by capture-position effects and link-layer behaviour. Some ARP, broadcast, multicast, and local discovery packets do not necessarily appear on the gateway egress interfaces after the traffic is mapped into the Docker topology. In addition, the cleaning stage intentionally removes duplicates and packets that do not match the expected replay signatures. Because all replayed IP packet rows were matched to Zeek flows and all TCP flows had zero missed bytes, this packet difference did not affect the reconstructed flow-level communication structure.

The distribution-level comparison shows strong preservation of byte-count, packet-count, IP-byte, and missed-byte features. All TCP flows in the replayed capture had `missed_bytes = 0`, indicating that Zeek did not observe TCP byte gaps in the replayed capture. The main divergence appears in the `duration` feature, where the replayed flows have a higher mean duration and a larger KS statistic. This suggests that the replay preserves packet order, communication structure, and traffic volume more accurately than exact flow timing. The timing difference is likely caused by source-aware container replay, gateway capture overhead, and scheduling effects during multi-container packet transmission.

4.5 Capture Quality and Packet Consistency

Reliable packet visibility became a major engineering challenge during the early development phases. Sender-side capture could observe the packets emitted by a replay container, but it did not observe the traffic after it had passed through the gateway. As a result, sender-side capture could not validate gateway-level effects such as routing behaviour, delay injection, or packet transformations introduced by the simulated topology. Gateway capture therefore became necessary because it provided visibility at the point where replayed traffic actually traversed the controlled network. However, capturing on the gateway with the Linux `any` interface introduced duplicated observations and interface-level ambiguity. The final implementation therefore moved toward more controlled gateway egress interfaces and post-processing in order to improve consistency between expected replay traffic and observed replay output. The final routed gateway architecture solved many of these limitations by centralizing traffic observation at a dedicated gateway node. Because all replay traffic traverses the gateway, packet captures become more deterministic and reproducible regardless of the original communication structure.

The replay framework automatically starts gateway captures before replay execution and stores replayed traffic for later processing. Captured packets are subsequently cleaned and validated in order to improve consistency between expected replay traffic and observed gateway traffic. The final implementation preserves visibility for TCP, UDP, ICMP, ARP, DHCP, multicast, and broadcast traffic, allowing Zeek to reconstruct meaningful connection summaries from replayed captures while maintaining centralized packet observation.

Another important result is that the replay framework preserves sufficient flow-level structure for later ML-based analysis. Zeek successfully reconstructs flow summaries from replayed captures, demonstrating that the replay process does not destroy important statistical communication properties required for flow-based IDS evaluation.

4.6 Ground Truth, Zeek Labelling, and Packet-to-Flow Provenance

One of the most important results of the thesis is the implementation of a transparent and auditable labelling workflow. Instead of assigning labels through opaque preprocessing or undocumented heuristics, the framework stores explicit replay metadata inside `ground_truth.csv`. Each replay execution generates metadata containing execution identifiers, sample identifiers, replay timing, traffic labels, replay multipliers, replay status, and execution notes.

After replay, Zeek processes the captured PCAP traffic and generates `conn.log` flow summaries. The script `label_zeek.py` then labels Zeek flows using overlap between flow timestamps and replay execution windows stored in the ground-truth metadata. This design provides deterministic labelling, explicit provenance, reproducible dataset generation, and auditable flow annotations. An additional packet-to-flow mapping stage implemented through `map_packets_to_flows.py` reconstructs relationships between replayed packets and labelled Zeek flows using normalized bidirectional flow keys and timing relationships.

As a result, the framework preserves traceability across the entire workflow, beginning from the original PCAP traces and continuing through rewritten replay traffic, gateway captures, Zeek flow summaries, and final labelled datasets. This provenance-oriented design directly addresses concerns identified in prior IDS dataset literature regarding reproducibility and label transparency.

4.7 Label-Audit Experiment

To estimate the precision of the automated labelling process, a stratified label audit was performed using one benign PCAP trace and one malicious PCAP trace. A random sample of 200 labelled Zeek flows was selected, consisting of 100 flows labelled as benign and 100 flows labelled as malicious. For each sampled flow, the corresponding PCAP was checked using the Zeek flow timestamp, duration, source and destination IP addresses, source and destination ports, and transport protocol.

A sampled flow was counted as correct when matching packet evidence was found in the corresponding PCAP and the automated label agreed with the known class of the source trace. This audit therefore estimates the precision of the implemented source-trace and time-window labelling scheme at the Zeek-flow level.

Table 4.4: Label-audit results for sampled Zeek flows.

Automated label	Audited flows	Correct	Incorrect	Precision
Benign	100	100	0	100.00%
Malicious	100	100	0	100.00%
Overall	200	200	0	100.00%

The audit found matching packet evidence for all 200 sampled flows and no incorrect labels within the audited sample. This result supports the correctness of the automated labelling process for the inspected benign and malicious traces. At the same time, the

audit should be interpreted as a sampled precision estimate rather than a proof that every generated flow label in every dataset is semantically perfect.

4.8 Machine-Learning Validation

The machine-learning validation stage evaluated whether the replay-generated traffic preserved useful flow-level characteristics for downstream intrusion-detection tasks. Labelled Zeek connection logs generated from original traffic, replay-generated traffic, and replay-generated traffic with gateway delay injection were used during evaluation. After dataset generation and merging through `merge_datasets.py`, an XGBoost classifier was trained using selected numeric Zeek flow features, including connection duration, packet counts, byte counts, IP byte counts, missed bytes, and protocol identifiers. All non-benign labels were mapped to the malicious class. Two datasets were used during evaluation:

- **Dataset 1**: a smaller initial dataset used during the first replay experiments.
- **Dataset 2**: a larger and more diverse dataset generated later in the project.

Dataset 1 contained 731 labelled Zeek flows in the original base dataset, consisting of 571 benign flows and 160 malicious flows. Dataset 1 additionally included replay-generated traffic without delay injection and replay-generated traffic with gateway delay injection enabled. Dataset 2 contained 1404 labelled Zeek flows in the original base dataset, consisting of 656 benign flows and 748 malicious flows. Dataset 2 included replay-generated traffic with gateway delay injection enabled. The following machine-learning results should be interpreted with caution, because the current evaluation does not fully enforce grouped train–test separation by PCAP source, malware family, or replay condition. This means that the results are best understood as proof-of-concept dataset-utility results rather than as strong evidence of generalization to unseen traffic sources.

4.8.1 Base and Replay Dataset Results

Tables 4.5 and 4.6 summarize the classification results for the original and replay-generated datasets.

Table 4.5: Dataset 1 classification results.

Traffic Type	Split	Accuracy	Precision	Recall	F1-score
Base traffic	50/50	97.54%	97.33%	91.25%	94.19%
Base traffic	70/30	98.64%	95.92%	97.92%	96.91%
Replay traffic	50/50	97.84%	95.51%	95.51%	95.51%
Replay traffic	70/30	98.21%	96.23%	96.23%	96.23%
Delayed replay traffic	50/50	98.38%	97.73%	95.56%	96.63%
Delayed replay traffic	70/30	98.65%	100.00%	94.44%	97.14%

Table 4.6: Dataset 2 classification results.

Traffic Type	Split	Accuracy	Precision	Recall	F1-score
Base traffic	50/50	93.00%	95.29%	90.76%	92.97%
Base traffic	70/30	92.14%	94.58%	89.72%	92.09%
Delayed replay traffic	50/50	91.74%	93.41%	90.91%	92.14%
Delayed replay traffic	70/30	91.47%	92.38%	91.56%	91.96%

4.8.2 Cross-Environment Experiments

Additional experiments evaluated classifiers trained on one traffic environment and tested on another traffic environment.

Table 4.7 summarizes the cross-environment experiments.

Table 4.7: Cross-environment classification experiments.

Training Dataset	Testing Dataset	Accuracy	Precision	Recall	F1-score
Base Dataset 1	Replay Dataset 1	95.82%	98.68%	83.71%	90.58%
Base Dataset 1	Delayed-Replay Dataset 1	95.96%	98.71%	84.53%	91.07%
Base Dataset 1	Replay and Delayed Replay Dataset 1	95.89%	98.69%	84.12%	90.83%
Base Dataset 2	Replay Dataset 2	88.53%	87.29%	91.84%	89.51%

4.8.3 Combined Dataset Experiments

The final experiments evaluated a merged dataset containing:

- Dataset 1 original traffic,
- Dataset 1 replay-generated traffic without delay,

- Dataset 1 replay-generated traffic with gateway delay injection,
- Dataset 2 original traffic,
- Dataset 2 replay-generated traffic with gateway delay injection.

The resulting merged dataset contained 5018 labelled Zeek flows, consisting of 3037 benign flows and 1981 malicious flows.

Table 4.8 summarizes the combined dataset classification results.

Table 4.8: Combined dataset classification results.

Split	**Accuracy**	**Precision**	**Recall**	**F1-score**
50/50	94.82%	94.98%	91.73%	93.33%
70/30	95.42%	95.66%	92.61%	94.11%

4.8.4 Confusion Matrix Results

To provide a simpler visualization of classifier behaviour, selected experiments were also evaluated using confusion matrices. The matrices summarize the number of correctly and incorrectly classified benign and malicious flows. The 50/50 train-test split was selected in order to provide a larger evaluation set and therefore a more representative view of classification errors.

Table 4.9: Dataset 1 base traffic confusion matrix (50/50 split).

	Predicted Benign	Predicted Malicious
Actual Benign	284	2
Actual Malicious	7	73

Table 4.10: Dataset 1 replay traffic confusion matrix (50/50 split).

	Predicted Benign	Predicted Malicious
Actual Benign	278	4
Actual Malicious	4	85

Table 4.11: Dataset 1 delayed replay traffic confusion matrix (50/50 split).

	Predicted Benign	Predicted Malicious
Actual Benign	279	2
Actual Malicious	4	86

Table 4.12: Dataset 2 base traffic confusion matrix (50/50 split).

	Predicted Benign	Predicted Malicious
Actual Benign	327	16
Actual Malicious	33	324

Table 4.13: Dataset 2 delayed replay traffic confusion matrix (50/50 split).

	Predicted Benign	Predicted Malicious
Actual Benign	304	24
Actual Malicious	34	340

Table 4.14: Combined dataset confusion matrix (50/50 split).

	Predicted Benign	Predicted Malicious
Actual Benign	1470	48
Actual Malicious	82	909

4.9 Automation and Reproducibility

A final major result concerns workflow automation and reproducibility. Earlier phases of the project required substantial manual configuration for topology construction, replay orchestration, packet capture, and labelling. The final implementation significantly reduced this complexity through script-based automation.

The script `run_pipeline.py` integrates the main semi-automated workflow, including topology extraction, Docker Compose generation, environment deployment, optional delay configuration, replay execution, gateway capture, Zeek processing, flow labelling, dataset merging, and ML validation. Packet-to-flow mapping is provided by the separate `map_packets_to_flows.py` script and can be run as an additional provenance-validation step when needed.

This automation substantially improves reproducibility because it separates malware execution from replay generation. The framework does not require malware execution during replay itself. Instead, malware may be executed independently inside isolated sandbox environments while only the resulting PCAP traces are introduced into the replay pipeline.

This separation improves containment, repeatability, experimental control, and safety during dataset generation. Selected console outputs from the topology extraction, Compose generation, Docker startup, replay, and ML-validation stages are provided in Appendix B. Overall, the implemented system demonstrates that reproducible malicious-traffic reconstruction can be achieved through a modular replay-centred workflow integrating topology extraction, controlled replay, centralized capture, explicit provenance, and automated labelled dataset generation.

5

Discussion

5.1 Overview

The results of this thesis demonstrate that reproducible malicious-traffic reconstruction can be achieved through a replay-centred methodology. This methodology combines topology extraction, controlled packet rewriting, centralized capture, explicit provenance tracking, and automated dataset generation. Rather than focusing solely on producing another static IDS dataset, the thesis approached traffic generation as a reproducible experimental process. This distinction is important because many existing datasets provide only final exported flows or CSV files without preserving the conditions under which the traffic was originally generated.

The implemented framework therefore contributes not only a dataset-generation pipeline, but also a methodology for reconstructable experiments. The workflow preserves relationships between original traces, replayed packets, captured traffic, Zeek flow summaries, and final labels. This improves traceability and allows later experiments to be repeated under controlled conditions.

The framework also demonstrates that replay-generated traffic can preserve important

flow-level characteristics required for ML-based IDS evaluation. Although the machine-learning validation performed in this thesis remained intentionally scoped, the results showed that replay-generated datasets achieved classification performance close to datasets generated directly from the original PCAP traces. This suggests that the replay process preserved many of the statistical properties required for downstream flow-based anomaly-detection tasks.

5.2 Methodological Contribution

The most significant contribution of the framework is methodological rather than algorithmic. Prior IDS dataset literature repeatedly identifies reproducibility, provenance, scalability, and transparency as major limitations in existing datasets [21, 17, 10]. Many publicly available datasets provide limited documentation regarding capture placement, timing manipulation, topology structure, traffic generation logic, or label derivation. As a result, reproducing the original dataset-generation conditions is often difficult or impossible.

The framework implemented in this thesis addresses several of these problems directly. Instead of treating malicious traffic generation as a single fixed output, the thesis treats it as a repeatable experimental workflow. Original PCAP traces are transformed into replayable scenarios through automatic topology extraction and deterministic address mapping. Replay metadata is stored explicitly through structured ground-truth records, while packet-to-flow mapping preserves provenance between replayed packets and labelled Zeek flows. A particularly important part of this workflow is gateway-based delay injection. By applying controlled delay between selected simulated source and destination hosts, the same original PCAP can be replayed under several different timing conditions. This makes it possible to generate dataset variation without changing the original malicious trace itself. For example, an attacker placed in a different network region would naturally introduce higher round-trip time, and the framework can emulate this effect through controlled gateway delay. Such timing variation is important because many flow-level ML features are sensitive to duration, packet spacing, and response timing. Therefore, delay injection strengthens the framework not only as a replay tool, but also as a dataset-variation mechanism for future IDS training and evaluation. This approach provides several advantages compared to static dataset-only generation. Timing manipulation becomes controllable, replay scenarios can be regenerated repeatedly, and new PCAP traces can be integrated into the same workflow without redesigning the environment manually. The framework therefore supports experimentation rather than merely dataset publication.

Another methodological contribution is the explicit separation between malware execution and replay generation. The final replay environment does not require active malware execution during the replay stage itself. Instead, malware may be executed independently inside isolated sandboxes, while only the resulting traffic traces are introduced into the replay pipeline. This separation improves reproducibility, reduces operational risk, and avoids many environment-specific compatibility issues encountered during the earlier development stages.

5.3 Assessment of Design Requirements

Section 3.1 introduced five design requirements for the framework: reproducibility, traceability, containment, configurability, and flow-oriented export. The implemented system addresses all five requirements, although the evidence is not equally quantitative for every requirement.

Reproducibility is addressed through deterministic topology extraction, static simulated IP assignment, generated Docker Compose files, scripted replay, Zeek processing, labelling, and pipeline orchestration. The results show that the main workflow can be repeated from the same PCAP input through topology generation, replay, capture, labelling, dataset merging, and ML validation. However, reproducibility was evaluated mainly through repeated successful pipeline execution, not through a formal repeatability study across many different hardware or operating environments.

Traceability is addressed through explicit ground-truth metadata, Zeek flow labels, packet-to-flow mapping, and provenance fields such as `sample_id`, and `execution_id`. The label-audit experiment and packet-to-flow mapping show that labelled flows can be traced back to replay metadata and packet-level evidence. At the same time, the labels still depend on time-window overlap and Zeek flow reconstruction, so they should be understood as transparent and auditable labels rather than perfect semantic descriptions of every packet.

Containment is addressed by replay-based traffic generation inside isolated Docker networks. Replayed traffic is routed through a controlled gateway, and the generated topology separates simulated hosts into controlled network segments. This limits communication to the local experiment environment and supports safer handling of malicious traffic traces. However, the thesis does not provide a formal security audit of Docker isolation, host-level exposure, or container escape risks.

Configurability is addressed through the use of JSON topology files, generated Compose files, replay parameters, gateway routing, configurable capture points, and optional

directional delay through `set_delay.sh`. The experiments with original, replayed, and delayed replay datasets show that replay conditions can be changed while still producing labelled flow-level data. However, the thesis does not exhaustively evaluate all possible topologies, protocols, replay speeds, or traffic volumes.

Flow-oriented export is addressed by converting PCAP traffic to Zeek `conn.log` records, assigning flow-level labels, merging labelled CSV files, and evaluating the resulting datasets with an XGBoost baseline. This requirement is supported by quantitative metrics such as accuracy, precision, recall, F1-score, and confusion matrices. These results validate the practical utility of the generated flow data for ML-based IDS experiments, but they do not prove general IDS performance across all models, features, malware families, or deployment environments.

Overall, the framework closes the loop between the stated design requirements, the implemented pipeline, and the supporting evidence. Reproducibility, traceability, containment, and configurability are mainly supported by engineering evidence, while flow-oriented export is also supported by quantitative ML-validation results. The evaluation therefore shows that the framework satisfies its main design goals for the experiments conducted in this thesis, while broader testing would be required to measure all requirements more comprehensively.

5.4 Interpretation of the Replay-Centred Design

As discussed in Section 3.1, the shift from live malware execution to replay-based reconstruction was a central design decision in the thesis. Replay-based reconstruction provided a more stable experimental basis because the input trace, topology mapping, capture position, replay timing, and labelling process could be controlled explicitly.

In this thesis, replay fidelity should not be understood as producing a replayed PCAP that is byte-identical to the original PCAP. Byte-level identity is neither expected nor required, because the framework intentionally rewrites addresses, reconstructs Ethernet headers, recalculates checksums, routes traffic through a generated Docker topology, and may apply controlled gateway delay. Instead, fidelity refers to the preservation of selected communication and flow-level properties that are relevant for labelled IDS dataset generation.

The essential properties to preserve are the communication relationships between hosts, the mapped source and destination placement, transport protocols and ports, packet ordering, approximate timing relationships, bidirectional flow context, and the flow-level

statistical properties exported through Zeek. Properties that are expected to change include simulated IP addresses, MAC addresses, checksums, link-layer details, exact capture position, and small timing differences caused by Docker routing, virtualization, capture processing, or optional delay injection.

The fidelity results should therefore be interpreted as a selected-property evaluation. Packet-level comparisons, timing comparisons, Zeek-flow feature distributions, and ML-utility results indicate whether the replayed traffic remains sufficiently similar for downstream flow-based IDS experiments. However, the selected fidelity properties are heuristic and task-driven rather than a complete formal definition of network replay fidelity.

The routed gateway is an intentional experimental abstraction. It approximates real network choke points such as routers, firewalls, gateways, or monitoring sensors where inter-zone traffic can be observed and controlled. This improves capture visibility, timing control, and reproducibility, but it also simplifies real networks that may contain multiple paths, switches, routers, and policy devices.

This design may introduce artefacts such as changed timing, different link-layer metadata, and capture-position effects caused by Docker routing or container scheduling. For this reason, the replay is evaluated in terms of preserved communication and flow-level properties rather than byte-identical packets. The artificial delay should also be understood as a controlled approximation of latency variation, not as a complete replacement for real RTT behaviour.

The trade-off is that replay does not fully reproduce host-side behaviour, dynamic malware adaptation, or complex command-and-control ecosystems. However, for flow-based IDS dataset generation, preserving communication structure and flow-level statistical behaviour is often more important than reproducing every host-level artefact. The replay-fidelity and machine-learning results indicate that many of these flow-level properties were preserved.

As shown in Table 4.2, the replay-fidelity analysis further showed that the framework preserved packet and flow structure with high accuracy. The cleaned gateway capture retained 99.84% of expected replayable packets, and the packet-to-flow provenance step matched all replayed IP packet rows and Zeek flows. The distribution-level comparison in Table 4.3 showed close preservation of packet-count and byte-count features, while the largest divergence appeared in Zeek flow durations. This timing difference is expected because replay timing reflects both the original inter-packet gaps and additional processing overhead from Scapy replay, container scheduling, routing, and gateway capture.

5.5 Interpretation of the Machine-Learning Validation

The machine-learning experiments should primarily be interpreted as a validation of dataset quality and replay fidelity rather than as a detector-performance contribution. The objective was not to develop a state-of-the-art IDS classifier, but instead to evaluate whether replay-generated traffic remained useful for downstream flow-based intrusion-detection experiments. The classification results discussed in this section are reported in Tables 4.5, 4.6, 4.7, and 4.8, with the corresponding confusion matrices shown in Tables 4.9–4.14.

XGBoost was selected for this validation because it is a strong and widely used baseline for structured tabular data, including flow-level features extracted from Zeek connection logs [4]. In this thesis, it is not used to claim a novel IDS model, but to check whether the generated labelled flows contain enough discriminative information for a supervised downstream experiment. The reported accuracy, precision, recall, F1-score, and confusion matrices should therefore be interpreted as dataset-utility indicators. They show how well the baseline classifier separates benign and malicious labelled flows under the tested conditions, but they should not be interpreted as evidence of production-ready IDS performance.

Although the thesis is motivated by ML-based network anomaly detection, the implemented ML validation should be understood as supervised binary classification. In strict terms, anomaly detection usually means learning a model of normal behaviour and identifying deviations from that normal baseline. The evaluation in this thesis instead trains and tests a classifier using labelled benign and malicious Zeek flows. The ML results therefore demonstrate that the generated labelled flow data can support supervised malicious-flow classification, but they do not demonstrate a complete unsupervised or semi-supervised anomaly-detection system.

This distinction is important for interpreting the results. The replay and labelling framework is intended to support future anomaly-detection research by producing traceable labelled flow data, while the XGBoost experiment is used only as a downstream dataset-utility check. Future work could evaluate stricter anomaly-detection settings by training only on benign replayed traffic and testing whether malicious replayed flows are detected as deviations from the learned normal behaviour.

This ML evaluation relates to the research questions indirectly. It supports the labelling

and flow-export part of the thesis by showing that the generated datasets can be consumed by a standard ML workflow. It also provides supporting evidence for replay fidelity when replayed and original datasets produce similar classification behaviour. However, the main contribution remains the replay, reconstruction, labelling, and dataset-generation pipeline rather than the classifier itself.

5.5.1 Interpretation of the Base and Replay Dataset Results

The baseline experiments using only the original PCAP-derived traffic achieved strong classification performance for both datasets. Dataset 1 produced accuracies between 97–98%, while Dataset 2 achieved accuracies around 92–93%. The lower baseline performance of Dataset 2 suggests that it contained more diverse or less separable traffic characteristics compared to Dataset 1.

The replay-only experiments showed that replay-generated traffic preserved many of the original flow-level characteristics used by the classifier. Replay-generated Dataset 1 traffic achieved classification performance very close to the original Dataset 1 baseline, both with and without gateway delay injection. Similarly, replay-generated Dataset 2 traffic achieved results close to the original Dataset 2 baseline despite the additional replay processing, simulated routing, and delay injection.

The results also indicate that gateway delay injection did not significantly degrade flow-level separability. In several Dataset 1 experiments, delayed replay traffic achieved results similar to or slightly stronger than replay traffic without delay injection. This suggests that the introduced timing modifications did not substantially disrupt the Zeek flow features used during classification.

5.5.2 Interpretation of the Cross-Environment and Combined Dataset Experiments

The cross-environment experiments produced more varied results. When training and testing remained within the same dataset distribution, replay-generated traffic generally preserved classification behaviour similar to the original datasets. For example, training on original Dataset 1 traffic and testing on replay-generated Dataset 1 traffic still achieved accuracies above 95%.

The combined dataset experiments further demonstrated that original traffic, replay-generated traffic, and delayed replay traffic could be merged into a single training and

evaluation dataset while still maintaining strong classification performance. The merged dataset contained 5018 labelled Zeek flows and achieved accuracies between 94–95% across both train-test split configurations.

5.5.3 Interpretation of the Confusion Matrix Results

The confusion matrices provide additional insight into the distribution of classification errors across the evaluated datasets. Overall, the results show that both replay-generated traffic and delayed replay-generated traffic preserved many of the flow-level characteristics required for binary malicious flow classification.

For Dataset 1, both the replay-generated and delayed replay traffic achieved results very close to the original base traffic. The number of false positives and false negatives remained low across all three Dataset 1 configurations, indicating that replay processing and gateway delay injection did not significantly disrupt the Zeek flow statistics used by the classifier. In particular, the delayed replay dataset slightly reduced false-positive classifications compared to the replay-only dataset, suggesting that the introduced delay mechanisms did not negatively affect flow separability.

Dataset 2 produced a larger number of classification errors than Dataset 1 for both the original and replay-generated traffic. Both false positives and false negatives increased noticeably, which is consistent with the lower overall accuracy and F1-score observed previously. This behaviour suggests that Dataset 2 contained more heterogeneous or less separable traffic patterns compared to Dataset 1.

Nevertheless, the replay-generated Dataset 2 traffic still achieved classification behaviour relatively close to the original Dataset 2 baseline, indicating that replay preserved many of the statistical properties present in the original flows.

The combined dataset confusion matrix further demonstrates that original traffic, replay-generated traffic, and delayed replay traffic could be merged into a single dataset while still maintaining strong classification performance. Although the combined dataset produced more false negatives than the smaller individual datasets, the overall number of incorrect classifications remained relatively low compared to the total number of evaluated flows.

The confusion matrices also highlight an important limitation of provenance-based labelling. Since malicious PCAP traces may contain benign background traffic generated by infrastructure services, DNS resolution, or unrelated operating-system activity, some benign flows may still receive malicious labels during dataset generation. This may par-

tially contribute to remaining false-positive and false-negative classifications within the evaluated datasets.

It is important to interpret these experiments within the scope of the thesis. The classifier architecture and feature set were intentionally kept relatively simple and relied only on a limited subset of Zeek flow statistics. The primary contribution of the work lies in the reproducible traffic-generation methodology, the replay framework, and the preservation of useful flow-level characteristics after replay inside a simulated routed environment.

5.6 Significance for Future IDS Research

The framework developed in this thesis may support future IDS research in several important ways. First, it preserves explicit relationships between original traces and replayed traffic. Many existing datasets lose this connection after preprocessing or feature extraction, making later validation difficult. By maintaining provenance across replay, capture, labelling, and flow generation, the framework improves traceability and experimental transparency.

Second, the framework supports both packet-level and flow-level analysis within the same workflow. Researchers can move between raw packets, gateway captures, Zeek logs, and labelled datasets without losing the relationship between these representations. This is useful for debugging, feature engineering, explainability experiments, and later IDS evaluation.

Third, the modular design simplifies extension with additional traffic sources, malware samples, replay strategies, or topology configurations. New PCAP traces can be integrated into the environment without redesigning the replay architecture manually. The gateway-centric design also enables future experiments involving routing policies, packet manipulation, or timing variation.

Fourth, the framework treats timing itself as a controllable experimental variable. Timing manipulation is often under-documented in IDS datasets, despite the fact that many flow-based features depend heavily on packet timing and communication duration. The implemented delay-injection support therefore provides opportunities for future experiments involving latency, traffic shaping, or timing-aware anomaly detection.

5.7 Threats to Validity

Although the framework addresses several important reproducibility and labelling challenges, a number of limitations and validity threats remain.

5.7.1 Construct Validity

Traffic labels are generated using replay timing windows and Zeek flow summaries rather than through perfect semantic understanding of every packet or host event. While this approach is substantially more transparent than weakly documented manual labelling, it still involves assumptions regarding replay timing and flow reconstruction. If replay timing diverges significantly from the original trace, if bidirectional flow reconstruction is imperfect, or if a flow spans replay boundaries unexpectedly, then the final label may still represent an engineered interpretation rather than absolute ground truth. The sampled label-audit results in Table 4.4 provide supporting evidence for the implemented labelling procedure, but they do not remove the broader construct-validity limitation.

This limitation is not unique to the present thesis and reflects a broader problem in IDS dataset generation. Prior work repeatedly identifies labelling accuracy and provenance as among the most difficult challenges in network-security datasets [10].

5.7.2 Internal Validity

Replay fidelity depends on correct namespace placement, address rewriting, routing configuration, interface selection, and packet reconstruction. Although the implementation explicitly addresses many of these issues, the framework remains technically complex and subtle configuration errors may still influence individual experiments. Similarly, some low-level network behaviour cannot be reproduced perfectly inside the replay environment. External infrastructure, dynamic internet services, adaptive malware behaviour, and host-side execution artefacts are simplified during replay. The framework therefore reconstructs communication behaviour rather than recreating a complete operational ecosystem.

A further internal-validity concern is the possibility of train–test leakage in the machine-learning validation. The current workflow uses separate `train_dataset.csv` and `test_dataset.csv` files, and the XGBoost script uses only selected numeric Zeek flow features for training. However, the thesis does not fully demonstrate that the split is grouped by original PCAP source, malware family, or replay condition. If flows derived from the same original PCAP, the same malware family, or the same replay configuration appear in both training and testing data, the classifier may learn dataset-specific artefacts

rather than behaviour that generalizes to unseen traces.

The feature set used for the XGBoost validation was restricted in order to reduce direct metadata leakage. In `train_xgboost.py`, the model input is constructed only from selected numeric Zeek connection-level features: `duration`, `orig_bytes`, `resp_bytes`, `orig_pkts`, `orig_ip_bytes`, `resp_pkts`, `resp_ip_bytes`, `missed_bytes`, and `ip_proto`. Fields such as `source_file`, `sample_id`, `execution_id`, original or mapped IP addresses, PCAP names, replay identifiers, and delay-configuration metadata are not included in this feature list. Although `merge_datasets.py` stores `source_file` in the merged dataset for debugging and provenance, this field is not selected by the classifier. This reduces the risk that the model learns explicit filenames, sample identifiers, or replay metadata. However, it does not fully remove the broader train–test leakage risk, because flows from the same source trace or replay condition may still share statistical patterns even when metadata fields are excluded.

For this reason, the reported ML results should be interpreted as proof-of-concept dataset-utility results rather than as a strong generalization claim. A more rigorous evaluation should construct train and test sets using grouped splitting, for example by assigning all flows from the same PCAP source, sample identifier, malware family, or replay condition to only one side of the split. This would provide a cleaner estimate of whether the generated labelled flows support classification on unseen traffic sources.

5.7.3 External Validity

The current framework is designed primarily for controlled malicious-traffic reconstruction on a single machine using Docker-based virtualization. The environment therefore does not yet represent large-scale distributed enterprise infrastructures, cloud-native deployments, or geographically distributed networks. In addition, the machine-learning validation was performed using a limited number of PCAP traces and a relatively small set of Zeek flow features. Although the evaluation produced encouraging results, broader validation using additional malware families, benign workloads, larger traffic captures, and more diverse Zeek log types would strengthen the generalizability of the findings. The thesis should therefore be interpreted as establishing a reproducible replay and labelling methodology for controlled network-security experimentation rather than as a complete simulation of large-scale real-world enterprise environments.

5.8 Answers to the Research Questions

This section answers the research questions stated in Section 1.3 by linking each question to the corresponding implementation and evaluation results.

RQ1: How can malicious network traffic captured from sandbox-generated PCAP files be reproducibly reconstructed and replayed within a controlled containerized network environment? The thesis answers this question by implementing a PCAP-driven replay pipeline that extracts topology, generates a Docker Compose environment, replays traffic through a routed gateway, captures the result, and exports labelled Zeek flows, as shown in Figure 3.1 and described in Sections 3.2 and 3.5. The results in Sections 4.2 and 4.9 show that this workflow can be automated and repeated from the same input PCAP and topology mapping. The remaining limitation is that the framework reconstructs network-level behaviour from captured traces rather than recreating the full live malware execution environment or its external infrastructure.

RQ2: How can relevant properties of the original traffic, such as topology, source placement, timing relations, and bidirectional communication context, be preserved or reconstructed inside that simulated environment? The thesis answers this question through topology extraction, deterministic address mapping, source-aware Scapy replay, protocol-aware packet rewriting, gateway capture, and packet-to-flow mapping, as described in Sections 3.3.1–3.3.7 and illustrated in Figures 3.2, 3.4, and 3.5. The replay-fidelity results in Tables 4.2 and 4.3 show that the cleaned replay capture retained 99.84% of expected replayable packets, matched all replayed IP packet rows to Zeek flows, and preserved packet-count and byte-count features closely. The main remaining limitation is timing fidelity: flow duration diverged more than packet and byte volumes, so the framework preserves communication structure more strongly than exact timing behaviour.

RQ3: How can packet-level and flow-level ground truth be generated in a transparent and traceable way so that downstream anomaly-detection experiments can rely on auditable labels? The thesis answers this question by storing explicit replay metadata in `ground_truth.csv`, labelling Zeek `conn.log` records through time-window overlap, and linking packets back to labelled flows with `map_packets_to_flows.py`, as described in Sections 3.3.6, 3.3.7, and 4.6 and illustrated in Figure 3.6. The label-audit experiment in Section 4.7 and Table 4.4 found matching packet evidence for all 200 sampled flows and no incorrect labels in the audited sample. The remaining limitation is that labels are still based on source-trace and time-window provenance, so benign background flows inside malicious PCAP windows and link-layer packets not represented in Zeek `conn.log` require careful interpretation.

5.9 Final Interpretation

Overall, the thesis demonstrates that replay-centred malicious-traffic generation can provide a practical balance between reproducibility, containment, controllability, and usefulness for ML-based IDS research. The implemented framework preserves important flow-level characteristics, maintains explicit provenance, and supports reproducible reconstruction of traffic scenarios through a modular workflow. The framework remains suitable for later Zeek-based anomaly-detection experiments, while avoiding many of the operational and reproducibility challenges associated with direct malware execution. Although further evaluation and expansion remain possible, the framework already establishes a strong methodological foundation for future reproducible IDS dataset-generation research.

6

Limitations and Future Work

6.1 Overview

Although the framework achieved the primary goals of reproducible malicious traffic reconstruction, centralized capture, explicit provenance tracking, and replay-based dataset generation, several limitations remain. Some of these limitations originate from practical constraints associated with a master's-thesis project, while others reflect broader challenges that remain open within IDS dataset research itself. The current implementation should therefore be viewed as a strong methodological foundation rather than as a complete production-scale traffic-generation platform. The framework demonstrates that replay-centred dataset generation is feasible and useful for ML-based IDS experiments, but additional work is still required to improve realism, scalability, protocol coverage, and experimental diversity.

6.2 Current Limitations

One important limitation concerns scalability. The framework currently operates on a single host and relies primarily on Docker bridge networks for isolation and routing. This design was sufficient for controlled experiments and simplified reproducibility during development, but it limits the realism and scalability of larger distributed infrastructures. The current environment does not yet support multi-host deployments, geographically distributed routing domains, cloud-native orchestration, or large-scale enterprise networks with complex switching and routing behaviour.

All experiments in this thesis were executed on the single-machine environment described in Table 4.1. On this hardware, the framework was practical for small and medium replay topologies, but larger generated environments exposed resource limitations. The main observed bottleneck was not the address-mapping scheme itself, but the gateway-centric capture design. The current implementation starts a separate `tcpdump` subprocess for each gateway egress interface, and the generated topology creates one gateway-facing network for each active simulated host. As a result, increasing the number of simulated hosts also increases the number of Docker networks, gateway interfaces, capture processes, packet buffers, and scheduling overhead.

In the experiments, replay environments with more than approximately 30 gateway egress interfaces caused a large increase in CPU, memory, and process-management overhead. On the evaluated WSL2/Docker laptop environment, this sometimes made the replay unstable and caused active replay or capture processes to be terminated. This limit should therefore be interpreted as a hardware-dependent practical limit of the tested setup, not as a fixed architectural maximum of the framework. Systems with more RAM, more CPU resources, faster storage, or a native Linux environment would likely support larger replay topologies. Nevertheless, the current implementation should be understood as practical for small and medium controlled experiments, while larger-scale traffic generation would require more powerful hardware and more systematic scalability testing.

Another limitation is the absence of an external replay-fidelity baseline. The thesis evaluates whether the implemented Scapy-based replay preserves selected packet-level, timing, and flow-level properties of the original PCAP traces, but it does not compare the results against alternative replay approaches such as a standard `tcpreplay/tcprewrite` workflow or another network-emulation testbed. Therefore, the reported fidelity results should be interpreted as evidence that the implemented framework preserves the selected properties under the tested conditions, rather than as evidence that it outperforms other replay methods.

Another limitation concerns replay-noise suppression. The replay framework installs temporary firewall rules in the replay containers and their route namespaces before packet transmission. These rules suppress common locally generated packets, specifically TCP reset packets and ICMP destination-unreachable messages, which were observed during early replay experiments when replayed traffic did not correspond to active sockets inside the containers. However, these rules should not be interpreted as a guarantee that all possible replay noise is removed for every network setup. Different container images, enabled services, protocols, or topology configurations could generate additional background traffic that would require further filtering or validation.

Another limitation concerns behavioural scope. The final implementation replays malicious network traces rather than executing the complete malware lifecycle inside the replay environment itself. This design choice improved reproducibility, containment, and experimental control, but it also means that certain host-side artefacts are not reproduced. Persistence mechanisms, dynamic command-and-control adaptation, privilege escalation behaviour, filesystem activity, process-level telemetry, and runtime interaction with external infrastructure remain outside the scope of the replay framework. The current implementation therefore reconstructs observed communication behaviour rather than recreating a fully operational malware ecosystem.

Protocol coverage also remains limited. The strongest parts of the framework currently concern IP-based communication, routed replay, DNS behaviour, connection-oriented traffic reconstruction, and Zeek flow generation. Although the replay engine preserves packet structure and timing relationships, deeper application-layer state emulation is still limited. Protocols that depend heavily on long-lived dialogue state, encrypted session negotiation, or adaptive responder logic may therefore not be reconstructed perfectly during replay.

Another limitation concerns benign workload diversity. The framework is capable of replaying both malicious and benign traffic traces through the same replay pipeline, including Scapy-based packet reconstruction and gateway-controlled replay. However, the current evaluation focused primarily on malicious-trace reconstruction, replay fidelity, labelling, and provenance preservation. As a result, the present experiments include only a limited set of benign background scenarios. Future IDS evaluation would benefit from a substantially broader collection of benign traffic traces in order to better represent realistic enterprise environments. Real-world networks typically contain heterogeneous benign communication patterns such as web browsing, streaming, software updates, authentication traffic, cloud services, developer workflows, and user-driven application activity. Expanding the diversity of replayed benign workloads would improve dataset realism and

reduce the risk of datasets being dominated by a relatively small number of malicious replay scenarios.

The current machine-learning evaluation also remains intentionally scoped. The thesis does not aim to achieve state-of-the-art IDS performance or provide comprehensive model benchmarking. Instead, the XGBoost experiments were designed primarily to evaluate whether replay-generated traffic preserved useful flow-level characteristics for downstream ML-based anomaly-detection tasks. Consequently, the evaluation used a relatively small set of Zeek connection-level features and a limited number of traffic captures. Additional experiments using larger datasets, alternative classifiers, and more diverse Zeek log sources could provide a broader evaluation of replay fidelity in future work.

Another limitation concerns the interpretation of malicious and benign traffic labels within replayed PCAP traces. The label-audit experiment in Section 4.7 found no incorrect labels among 200 sampled Zeek flows, indicating strong precision for the audited benign and malicious traces. However, the audit remains sample-based and does not prove perfect semantic correctness for every flow in every generated dataset. Malware-related PCAP captures may still contain benign background traffic generated by operating-system services, DNS resolution, or unrelated network activity. More precise flow-level attribution would require additional host-based telemetry, sandbox instrumentation, or process-level network tracing.

Replay fidelity itself also remains a limitation. Although packet timing, address relationships, and overall flow structure were preserved to a substantial degree during the experiments, the replay environment cannot perfectly reproduce every aspect of the original network behaviour. Internet-facing infrastructure may change between the original capture and later replay, while some multicast, broadcast, and timing-sensitive behaviours may depend on environmental conditions not fully reproduced inside the simulated topology. Similarly, replay timing may diverge slightly from the original traces because of virtualization overhead, routing delays, or container scheduling behaviour within the replay environment.

Finally, the framework currently depends heavily on Zeek `conn.log`-oriented analysis. While Zeek provides strong flow-level visibility, richer analysis involving DNS logs, HTTP logs, Transport Layer Security (TLS) logs, certificate metadata, TLS fingerprinting features, and application-layer protocol features has not yet been explored fully within the current implementation.

6.3 Future Work

Several natural extensions follow directly from the present work. One of the most important future directions concerns expansion of benign background traffic. A broader library of replayable benign traces and service-oriented workloads would significantly improve the realism of generated datasets. Future environments could include simulated enterprise activity such as web browsing, email traffic, software updates, cloud applications, internal services, authentication systems, development workflows, and user-driven interaction patterns. Increasing benign traffic diversity would also improve the statistical balance of future anomaly-detection experiments.

Another important extension involves scaling the framework beyond a single host. A distributed replay architecture using multiple machines or container-orchestration platforms could improve realism for larger routing domains and more demanding traffic scenarios. Integration with technologies such as Kubernetes, distributed virtual switches, software-defined networking, or cloud-based orchestration would allow experiments involving larger infrastructures and more complex routing behaviour.

Future work should include a replay-fidelity baseline comparison. For example, the same PCAP traces could be replayed using the implemented Scapy-based framework and a simpler `tcpreplay`/`tcprewrite`-based workflow, then compared using the same packet-level, timing, Zeek-flow, and ML-utility metrics. This would make it possible to separate the benefits of the proposed topology-aware replay design from the effects of replaying the traffic in any controlled environment.

Future work should also extend the protocol-awareness of the replay environment. Although packet replay preserves communication structure, richer responder logic would improve realism for protocols that depend heavily on dialogue state or adaptive interaction. More advanced protocol emulation could include stateful HTTP responders, DNS behaviour modelling, TLS negotiation reconstruction, or lightweight command-and-control simulation.

The packet-to-flow provenance layer could also be extended into a formal dataset-export interface. Such an interface would allow every generated dataset to include explicit mappings between original traces, replayed packets, Zeek flows, labels, and replay metadata. This would improve auditability, dataset transparency, and long-term reproducibility for future IDS research.

The machine-learning component also provides several opportunities for future expan-

sion. Future evaluation should include larger datasets, more malware families, additional benign workloads, repeated experiments using multiple delays, and stronger separation between training and testing sources. Additional models such as Random Forest, Logistic Regression, LightGBM, neural-network-based approaches, or sequence-oriented models could also be evaluated. Feature engineering should likewise be expanded beyond basic `conn.log` statistics. Future experiments could incorporate richer Zeek logs including DNS, HTTP, Secure Sockets Layer (SSL)/TLS, certificate, file-transfer, and application-layer metadata. This would help evaluate whether replayed traffic preserves useful behavioural properties beyond basic connection-level statistics.

Finally, future work could investigate automatic scenario generation and continuous dataset production. Instead of replaying only manually selected PCAP traces, future systems could automatically ingest new captures, generate replay environments dynamically, validate replay fidelity, and export labelled datasets continuously. Such automation could support long-term IDS evaluation pipelines and continuously updated research datasets.

6.4 Final Remarks

Despite its current scope limitations, the framework establishes a strong foundation for reproducible replay-centred malicious-traffic generation and dataset creation for network-security research. The thesis demonstrated that controlled topology reconstruction, deterministic packet rewriting, centralized gateway-based capture, explicit provenance tracking, automated labelling, and replay-based traffic generation can be integrated into a unified and reproducible workflow suitable for ML-based IDS experimentation. The evaluation results further showed that replay-generated traffic preserved many of the flow-level statistical properties required for downstream classification tasks, while also supporting controlled experimentation across different replay, routing, and delay configurations. The framework should therefore be viewed not only as a prototype implementation, but also as a methodological contribution toward more reproducible, auditable, and extensible traffic-generation practices within network-security research.

7

Ethical Considerations

7.1 Research Ethics and Safe Experimentation

The ethical perspective of this thesis is informed by the Menlo Report, which adapts the Belmont principles to information and communication technology research and emphasizes the principles of respect for persons, beneficence, justice, and respect for law and public interest [5]. This framework is particularly relevant because the thesis concerns cybersecurity experimentation, malicious network traffic, and potentially dual-use technical artefacts.

The project was designed to minimize ethical and operational risk throughout the development process. No real end-user traffic was collected, and the malicious traces used by the framework originated from controlled sandbox environments rather than from live victim systems or production infrastructure. An important ethical characteristic of the final framework is that dataset generation relies on replayed PCAP traces rather than active malware execution. Previously captured traces are reconstructed and replayed inside an isolated environment, reducing the risk of unintended or uncontrolled malicious behaviour during dataset generation.

7.2 Containment and Dual-Use Considerations

Containment was treated as a core engineering requirement rather than as an optional operational safeguard. The framework uses generated local topologies, deterministic address mapping, centralized gateway-based routing, controlled replay placement, and isolated Docker networks to reduce the risk of uncontrolled external communication. The replay-centred architecture also improves transparency regarding experimental boundaries. Because replay traffic traverses a centralized gateway, capture placement and network visibility remain explicitly controlled throughout the experiments.

The work additionally acknowledges the dual-use nature of malicious-traffic research. Techniques developed for replaying or reconstructing malicious communication could potentially be misused if deployed irresponsibly. However, the purpose of the framework is defensive and research-oriented. The objective is not to facilitate offensive deployment or malware development, but rather to support reproducible IDS dataset generation, traffic analysis, and anomaly-detection evaluation.

For this reason, the thesis emphasizes replay traceability, dataset provenance, labelling transparency, and controlled experimentation rather than operational attack automation. If the framework or generated datasets are distributed in the future, care should be taken to share replay artefacts, labels, documentation, and derived metadata without unnecessarily distributing active malware binaries or dangerous execution components.

7.3 Data Handling and Publication Considerations

The thesis focuses primarily on network traces and derived flow labels rather than on personal user data, which substantially reduces many common privacy risks associated with enterprise traffic collection. Nevertheless, future publication of replay-generated datasets should still include clear documentation regarding sample origin, replay methodology, topology mapping assumptions, packet transformations, and labelling procedures. Such transparency is important not only for scientific reproducibility, but also for accountability and responsible cybersecurity research. Documenting how datasets are generated, transformed, and labelled helps other researchers evaluate methodological limitations and reduces the risk of misleading or weakly documented benchmark construction. These considerations are consistent with the Menlo Report's emphasis on respect for law, public interest, and responsible disclosure practices within information-security research [5].

Regarding public release of artefacts, the implementation repository has been published on GitHub to make the framework code and documentation available. The intended

public artefacts are limited to the scripts, Docker-related configuration logic, helper files, and usage documentation needed to reproduce the framework. The repository is configured so that generated or sensitive experiment artefacts, such as PCAP files, Zeek output directories, labelled CSV datasets, ground-truth files, and temporary replay outputs, are excluded from version control through `.gitignore`. Raw malicious PCAP files, third-party PCAP traces, derived Zeek logs, and labelled flow datasets are therefore not planned for unrestricted public release by default, because they may contain sensitive network indicators, third-party data, or information that could support misuse if distributed without review. If datasets are released in the future, they should be limited to carefully reviewed, anonymized, or synthetic examples together with clear documentation of their origin, transformations, and labelling procedure.

In the current evaluation, the risk of private-information leakage was reduced by using controlled or publicly available research PCAP traces rather than traffic collected from private users or production enterprise networks. However, PCAP traces and derived Zeek logs may still contain sensitive network identifiers, such as IP addresses, domain names, hostnames, URLs, certificates, or protocol metadata. Therefore, any future dataset release should include an additional review and sanitization step, such as removing or anonymizing sensitive identifiers and documenting the transformation process before publication.

7.4 Use of AI Tools

Generative AI tools, including ChatGPT, were used during the thesis process for language refinement, grammar correction, code assistance, and improvement of text readability. AI-assisted tools were also used to support the preparation and refinement of some explanatory figures. Console-output figures in Appendix B were captured from the implemented experimental workflow.

All implementation work, experiments, system design decisions, analysis, and final verification of the thesis content were performed by the author. The author reviewed and validated all generated text, code, and figures before inclusion in the final thesis.

8

Conclusion

This thesis presented a reproducible framework for reconstructing, replaying, capturing, and labelling malicious network traffic inside a controlled containerized environment for future machine-learning-based intrusion-detection research. Starting from sandbox-generated PCAP traces, the framework automatically extracts communication structure, generates a synthetic replay topology, replays the traffic inside isolated Docker networks, captures the resulting communication at centralized observation points, and produces labelled Zeek flow data through explicit replay metadata.

A central outcome of the thesis is the replay-centred dataset-engineering approach, which improved reproducibility, containment, experimental control, and label transparency. The thesis therefore contributes primarily to methodology and reproducible infrastructure, rather than to the development of a novel IDS model.

The implemented framework addresses several limitations identified in the IDS dataset literature by improving replay traceability, provenance, and dataset reproducibility. Relationships between original traces, replayed traffic, gateway captures, Zeek flow summaries, and final labels are preserved throughout the workflow, allowing experiments to be regenerated under controlled conditions.

Machine-learning validation using Zeek flow features and an XGBoost classifier showed that replay-generated datasets achieved classification performance close to datasets generated directly from the original PCAP traces. The experiments also demonstrated that replay-generated traffic remained useful for mixed-dataset evaluation scenarios involving both original and replay-derived traffic. The results suggest that replay-centred reconstruction preserved many of the flow-level statistical properties required for downstream flow-based IDS experiments, even after topology reconstruction, packet rewriting, simulated routing, and gateway delay injection. Several limitations remain, including restricted benign workload diversity, limited protocol-state emulation, and evaluation using a relatively small number of traffic traces and Zeek flow features. Nevertheless, the thesis demonstrates that reproducible malicious-traffic reconstruction is both feasible and useful for network-security research. The resulting framework establishes a strong foundation for future replay-centred IDS dataset generation, replay fidelity evaluation, and controlled ML-based network-security experimentation.

A

Experimental Dataset

Table A.1: PCAP traces used in the experimental evaluation.

Field	**Value**
Identifier	`2025-06-13-traffic-analysis-exercise.pcap`
Source	`https://www.malware-traffic-analysis.net/2025/06/13/index.html`
Attack type/family	Malware / intrusion traffic; exact malware family not stated by the public source
Capture date	2025-06-13
Traffic label	Malicious
SHA-256	`33e274d7246f4eca8fbe7337c465c7ded077cd650d5d4d42481cad822c962741`
Publication status	Publicly available source PCAP; thesis uses a derived 5000-packet subset
Used in evaluation	Replay-fidelity analysis, Zeek labelling, packet-to-flow mapping, Dataset 1 generation, and ML validation

Field	Value
Identifier	`2026-04-23-traffic-from-SmartApeSG-activity.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/04/23/index.html`
Attack type/family	SmartApeSG malware activity
Capture date	2026-04-23
Traffic label	Malicious
SHA-256	`bc4b6f4f5913523a13cc079f00a7bb30a8822e58603fad6d82c28f013809dd80`
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 1 generation (5 000-packet subset), Dataset 2 generation (10 000-packet subset), and ML validation
Identifier	`2026-02-03-GuLoader-for-AgentTesla-style-infection-with-FTP-data-exfil.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/02/03/index.html`
Attack type/family	GuLoader / AgentTesla-style infection with FTP data exfiltration
Capture date	2026-02-03
Traffic label	Malicious
SHA-256	`9fbd2e59fb9981cb6d9d5cbf7e0f12fb6f7cb56d1a944f7eb34e5fddf8e54c31`
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 1 generation, Dataset 2 generation, and ML validation
Identifier	`2026-05-08-macOS-Shub-Stealer-infection.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/05/08/index.html`
Attack type/family	macOS Shub Stealer infection
Capture date	2026-05-08
Traffic label	Malicious
SHA-256	`e1e2d5c5a4e71e7c6f8a1d7dcaad53fd0f3f3ecdd34a9325b53d04e4d1cf0e65`

Field	Value
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation (10 000-packet subset) and ML validation
Identifier	`2026-05-11-macOS-malware-infection-traffic.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/05/11/index.html`
Attack type/family	macOS malware infection traffic
Capture date	2026-05-11
Traffic label	Malicious
SHA-256	`3f4c418774b6860247c9053db6062081eaf98f514b02401850aa7e49ad1ff1ea`
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation (10 000-packet subset) and ML validation
Identifier	`2026-01-09-VIP-Recovery-traffic.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/01/09/index.html`
Attack type/family	VIP Recovery malware traffic
Capture date	2026-01-09
Traffic label	Malicious
SHA-256	`42a49060889d44dee993f6b1c6493e5a1ec2c708befdd28c8d57c039f406af2d`
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation and ML validation
Identifier	`2026-01-20-Xworm-infection-traffic.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/01/20/index.html`
Attack type/family	Xworm infection traffic
Capture date	2026-01-20

Field	Value
Traffic label	Malicious
SHA-256	271fd88240a28c2c5a2fdbec769e0746a6b2384199d94455c3dc0de8917ea270
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation and ML validation
Identifier	2026-02-02-KongTuke-activity-for-MintsLoader-and-GhostWeaver-RAT.pcap
Source	https://www.malware-traffic-analysis.net/2026/02/02/index.html
Attack type/family	KongTuke activity for MintsLoader and GhostWeaver RAT
Capture date	2026-02-02
Traffic label	Malicious
SHA-256	b7b770b626c7e0ed46284b11c066594f6f77a77a562b0514cf5921d339745a8b
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation (10 000-packet subset) and ML validation
Identifier	2026-04-06-SmartApeSG-traffic.pcap
Source	https://www.malware-traffic-analysis.net/2026/04/06/index.html
Attack type/family	SmartApeSG malware traffic
Capture date	2026-04-06
Traffic label	Malicious
SHA-256	a46c76bbf8cf91e7a7249d1ec1fc4d91cfba9db66f7640bdb1c3c2f0f3f6b2d6
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation (10 000-packet subset) and ML validation
Identifier	2026-04-13-XLoader-infection-traffic.pcap
Source	https://www.malware-traffic-analysis.net/2026/04/13/index.html

Field	Value
Attack type/family	XLoader infection traffic
Capture date	2026-04-13
Traffic label	Malicious
SHA-256	`482a9c90793565dd1be62e4212bbc3af07bdaa6577296e3220c87e5c2aa2d070`
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation and ML validation
Identifier	`2026-04-16-Lumma-Stealer-infection-with-Sectop-RAT.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/04/16/index.html`
Attack type/family	Lumma Stealer infection with Sectop RAT activity
Capture date	2026-04-16
Traffic label	Malicious
SHA-256	`a430dc87508451dad8e70a7ed9b5d912054482d16745b4eadd93411e44dd41fc`
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation (10 000-packet subset) and ML validation
Identifier	`2026-04-22-macOS-malware-infection-from-ClickFix-script.pcap`
Source	`https://www.malware-traffic-analysis.net/2026/04/22/index.html`
Attack type/family	macOS malware infection from ClickFix script activity
Capture date	2026-04-22
Traffic label	Malicious
SHA-256	`f06e6c3a388dbd534eb54d72a5ea7f998f2c48c3ebd7b68f247df3d017eb5e7d`
Publication status	Publicly available source PCAP
Used in evaluation	Dataset 2 generation (10 000-packet subset) and ML validation

B

Experimental Console Outputs

This appendix contains selected console outputs.

```
(venv) moham@LAPTOP-RLVT1O97:~/master-thesis$ python3 extract_topology.py Malware_5k_packets.pcap
Wrote topology.json
Detected DHCP 0.0.0.0 owner: none
dhcp_zero=0.0.0.0 ignored because DHCP owner could not be detected
Wrote simulated_topology.json
zones=23
zone_labels=A,Z0002,Z0003,Z0004,Z0005,Z0006,Z0007,Z0008,Z0009,Z0010,Z0011,Z0012,Z0013,Z0014,Z0015,Z0016,Z0017,Z0018,Z001
9,Z0020,Z0021,Z0022,Z0023
hosts_per_zone=2,1,1,2,8,1,2,2,2,1,1,1,1,1,1,1,2,1,1,1,1,1,1
mapping_entries=36
ignored_hosts=6
```

Figure B.1: Extract topology.

```
(venv) moham@LAPTOP-RLVT1O97:~/master-thesis$ python3 generate_compose.py --topology simulated_topology.json
Wrote docker-compose.yml
services=74
networks=36
active_hosts=36
skipped_dhcp_zero=0
```

Figure B.2: Generate compose based on simulated topology.

```
(venv) moham@LAPTOP-RLVT1097:~/master-thesis$ docker compose up -d
[+] up 109/109
 ✔Network master-thesis_Z0005_external_host_06_net Created
 ✔Network master-thesis_Z0017_external_host_02_net Created
 ✔Network master-thesis_Z0005_external_host_03_net Created
 ✔Network master-thesis_Z0005_external_host_01_net Created
 ✔Network master-thesis_Z0007_external_host_02_net Created
 ✔Network master-thesis_Z0011_external_host_01_net Created
 ✔Network master-thesis_Z0019_external_host_01_net Created
 ✔Network master-thesis_Z0015_external_host_01_net Created
 ✔Network master-thesis_Z0005_external_host_04_net Created
 ✔Network master-thesis_Z0007_external_host_01_net Created
 ✔Network master-thesis_Z0012_external_host_01_net Created
 ✔Network master-thesis_Z0005_external_host_02_net Created
 ✔Network master-thesis_Z0006_external_host_01_net Created
 ✔Network master-thesis_Z0005_external_host_07_net Created
 ✔Network master-thesis_Z0005_external_host_05_net Created
 ✔Network master-thesis_Z0003_external_host_01_net Created
 ✔Network master-thesis_Z0023_external_host_01_net Created
 ✔Network master-thesis_Z0010_external_host_01_net Created
 ✔Network master-thesis_Z0013_external_host_01_net Created
 ✔Network master-thesis_Z0009_external_host_02_net Created
 ✔Network master-thesis_Z0018_external_host_01_net Created
 ✔Network master-thesis_Z0004_external_host_02_net Created
 ✔Network master-thesis_Z0020_external_host_01_net Created
 ✔Network master-thesis_Z0014_external_host_01_net Created
 ✔Network master-thesis_Z0008_external_host_01_net Created
 ✔Network master-thesis_Z0008_external_host_02_net Created
 ✔Network master-thesis_Z0004_external_host_01_net Created
 ... 82 more
```

Figure B.3: Starting docker compose.

```
(venv) moham@LAPTOP-RLVT1097:~/master-thesis$ python3 replay_traffic.py --label malicious
[*] Sample ID            : Malware_5k_packets
[*] Multiplier           : 1.0
[*] Capture point        : GW outbound interfaces, tcpdump -Q out
[*] Rewriting progress   : 100%
[*] Rewrite completed    : 5000 replayable packets
[*] Skipped packets      : 0
[*] GW egress capture    : master-thesis-gw:eth0-eth35 (-Q out)
[*] Starting live replay : ARP/IP all protocols
[*] Replay progress      : 100%
[*] Live sent packets    : 5000
[*] Replay skipped       : 0
[*] Replay elapsed       : 317.872452s
```

Figure B.4: Rewriting process in replay stage.

```
[*] Clean gateway egress :    /home/moham/master-thesis/gateway_egress.pcap
[*] Observed kept        : 4933
[*] Fallback added       : 67
[*] Final packets        : 4992
[*] Expected packets     : 5000
[*] Skipped other        : 0
[*] Ground truth updated : ground_truth.csv
[+] Gateway replay finished.
```

Figure B.5: Replay process.

```
(venv) moham@LAPTOP-RLVT1O97:~/master-thesis$ python3 train_xgboost.py

Experiment results 50/50 split
------------------------------
TP: 340
TN: 304
FP: 24
FN: 34

Accuracy : 0.9174
Precision: 0.9341
Recall   : 0.9091
F1-score : 0.9214
```

Figure B.6: ML validation 50/50 using Dataset 2 replayed traffic.

Bibliography

[1] AppNeta. *Tcpreplay: Pcap Editing and Replaying Utilities*. URL: https://tcpreplay.appneta.com/ (visited on 05/18/2026).

[2] AppNeta. *tcprewrite*. URL: https://tcpreplay.appneta.com/wiki/tcprewrite (visited on 05/18/2026).

[3] Canadian Institute for Cybersecurity. *Intrusion Detection Evaluation Dataset (CIC-IDS2017)*. URL: https://www.unb.ca/cic/datasets/ids-2017.html (visited on 05/18/2026).

[4] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785.

[5] David Dittrich and Erin Kenneally. *The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research*. U.S. Department of Homeland Security, 2012. URL: https://www.dhs.gov/archive/science-and-technology/publication/st-menlo-report (visited on 05/18/2026).

[6] Docker Inc. *Bridge Network Driver*. URL: https://docs.docker.com/engine/network/drivers/bridge/ (visited on 05/18/2026).

[7] Docker Inc. *Define and Manage Networks in Docker Compose*. URL: https://docs.docker.com/reference/compose-file/networks/ (visited on 05/18/2026).

[8] Carlos Garcia Cordero et al. "On Generating Network Traffic Datasets with Synthetic Attacks for Intrusion Detection". In: *ACM Transactions on Privacy and Security* 24.2 (2021), pp. 1–39. DOI: 10.1145/3424155.

[9] GNS3. *Getting Started with GNS3*. URL: https://docs.gns3.com/docs/ (visited on 05/27/2026).

[10] Jorge Luis Guerra, Carlos Adrián Catania, and Eduardo Enrique Veas. "Datasets Are Not Enough: Challenges in Labeling Network Traffic". In: *Computers & Security* 120 (2022), p. 102810. DOI: 10.1016/j.cose.2022.102810.

[11] Alefiya Hussain, Yuri Pradkin, and John Heidemann. "Replay of Malicious Traffic in Network Testbeds". In: *Proceedings of the 13th IEEE Conference on Technologies for Homeland Security*. Waltham, Massachusetts, USA: IEEE, 2013. URL: https://ant.isi.edu/~johnh/PAPERS/Hussain13a.html (visited on 05/27/2026).

[12] Michael Kerrisk. *tc-netem(8) — Linux Manual Page*. URL: https://man7.org/linux/man-pages/man8/tc-netem.8.html (visited on 05/18/2026).

[13] Michael Kerrisk. *tc(8) — Linux Manual Page*. URL: https://man7.org/linux/man-pages/man8/tc.8.html (visited on 05/18/2026).

[14] Malware-Traffic-Analysis.net. *Malware-Traffic-Analysis.net*. URL: https://www.malware-traffic-analysis.net/ (visited on 05/27/2026).

[15] Nour Moustafa and Jill Slay. "UNSW-NB15: A Comprehensive Data Set for Network Intrusion Detection Systems". In: *2015 Military Communications and Information Systems Conference*. IEEE, 2015, pp. 1–6. DOI: 10.1109/MilCIS.2015.7348942.

[16] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. "Flow-Based Network Traffic Generation Using Generative Adversarial Networks". In: *Computers & Security* 82 (2019), pp. 156–172. DOI: 10.1016/j.cose.2018.12.012.

[17] Markus Ring et al. "A Survey of Network-Based Intrusion Detection Data Sets". In: *Computers & Security* 86 (2019), pp. 147–167. DOI: 10.1016/j.cose.2019.06.005.

[18] Xabier Sáez-de-Cámara et al. "Gotham Testbed: A Reproducible IoT Testbed for Security Experiments and Dataset Generation". In: *IEEE Transactions on Dependable and Secure Computing* (2024). DOI: 10.1109/TDSC.2023.3247166.

[19] Scapy Project. *Usage — Scapy Documentation*. URL: https://scapy.readthedocs.io/en/stable/usage.html (visited on 05/18/2026).

[20] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization". In: *Proceedings of the 4th International Conference on Information Systems Security and Privacy*. 2018, pp. 108–116. DOI: 10.5220/0006639801080116.

[21] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. "Toward Developing a Systematic Approach to Generate Benchmark Datasets for Intrusion Detection". In: *Computers & Security* 31.3 (2012), pp. 357–374. DOI: 10.1016/j.cose.2011.12.012.

[22] Robin Sommer and Vern Paxson. “Outside the Closed World: On Using Machine Learning for Network Intrusion Detection”. In: *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 305–316. DOI: 10.1109/SP.2010.25.

[23] Stratosphere Laboratory. *Stratosphere Laboratory Datasets Overview*. URL: https://www.stratosphereips.org/datasets-overview (visited on 05/27/2026).

[24] UNSW Research. *The UNSW-NB15 Dataset*. URL: https://research.unsw.edu.au/projects/unsw-nb15-dataset (visited on 05/18/2026).

[25] Wireshark Foundation. *TShark(1) Manual Page*. URL: https://www.wireshark.org/docs/man-pages/tshark.html (visited on 05/18/2026).

[26] Yucheng Yin et al. “Practical GAN-Based Synthetic IP Header Trace Generation Using NetShare”. In: *Proceedings of the ACM SIGCOMM 2022 Conference*. Association for Computing Machinery, 2022, pp. 458–472. DOI: 10.1145/3544216.3544251.

[27] Zeek Project. *conn.log — Book of Zeek*. URL: https://docs.zeek.org/en/current/reference/logs/conn.html (visited on 05/18/2026).