IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 January 2026, accepted 20 February 2026, date of publication 25 February 2026, date of current version 6 March 2026.

*Digital Object Identifier 10.1109/ACCESS.2026.3668033*

RESEARCH ARTICLE

# Lightweight WebAssembly-Based Intrusion Detection for Zero Trust Edge Networks

**JONATHAN WEBER[1], ILIR MURTURI[1,2], (Senior Member, IEEE), XHEVAHIR BAJRAMI[2], REZA FARAHANI[3], PRAVEEN KUMAR DONTA[4], (Senior Member, IEEE), AND SCHAHRAM DUSTDAR[1,5], (Fellow, IEEE)**

[1]Distributed Systems Group, TU Wien, Vienna 1040, Austria
[2]Department of Mechatronics, University of Prishtina, Prishtina 10000, Kosova
[3]Institute of Information Technology (ITEC), University of Klagenfurt, Klagenfurt 9020, Austria
[4]Department of Computer and Systems Sciences, Stockholm University, 16425 Stockholm, Sweden
[5]ICREA, 08002 Barcelona, Spain

Corresponding author: Ilir Murturi (i.murturi@dsg.tuwien.ac.at)

**ABSTRACT** IoT devices deployed across computing continuum infrastructures present significant security challenges due to resource constraints and decentralization. Traditional centralized intrusion detection systems struggle in such environments because of limited connectivity, high latency, and single points of failure. To address these challenges, this article extends a learning-driven Zero Trust framework tailored to resource-constrained edge environments and proposes an approach for evaluating lightweight intrusion detection models in such environments. Our extended approach enables systematic evaluation of lightweight machine learning models for localized intrusion detection, comprising three layers: *(i)* compilation, *(ii)* execution, and *(iii)* measurement. The proposed approach is implemented using Rust and WebAssembly to ensure portable, efficient, and isolated execution across heterogeneous devices. Using this framework, seven representative intrusion detection models (i.e., Decision Tree (DT), Random Forest (RF), k-Nearest Neighbor (KNN), Logistic Regression (LR), Artificial Neural Network (ANN), and Convolutional Neural Network (CNN) variants) were implemented and evaluated on the UNSW-NB15 dataset. Results show that RF achieved the best trade-off between detection accuracy and efficiency, while simpler models (DT and LR) offered near-instant inference with minimal resource usage, making them ideal for highly constrained devices. In contrast, more complex models such as deep neural networks and KNN introduced significant overhead for only modest accuracy gains. These findings underscore the need to balance accuracy and resource efficiency for effective Zero Trust edge security.

**INDEX TERMS** Zero trust, edge computing, computing continuum, intrusion detection models, WebAssembly.

## I. INTRODUCTION

The proliferation of Internet of Things (IoT) devices across computing continuum infrastructures, also known as Distributed Computing Continuum Systems (DCCS), has transformed modern cyber-physical infrastructures by enabling large-scale sensing, automation, and real-time analytics [1], [2]. However, this massive decentralization also leads to a proportional increase in the attack surface. Edge and fog tier devices frequently operate with limited computational resources, unstable connectivity, and heterogeneous architectures, making them attractive targets for cyber adversaries [3], [4]. In critical systems such as smart energy grids, the compromise of even small subsets of distributed devices can result in system-wide instability, service disruption, or cascading failures [5], [6]. To do that, the *Zero Trust Architecture* (ZTA), formalized by the National Institute of Standards and Technology (NIST), has emerged

The associate editor coordinating the review of this manuscript and approving it for publication was Tyson Brooks.

as a paradigm shift away from perimeter-based security models [7].

ZTA requires continuous authentication and authorization of every access request, independent of network location. However, applying ZTA directly to DCCS exposes several limitations. For example, intermittent connectivity prevents reliable round-trip verification to cloud-resident security components. In DCCS, many edge devices are either mobile or deployed in remote, hard-to-reach environments, which means their network links are often unstable or intermittently available. Traditional ZTA workflows rely on constant communication with cloud-hosted engines for tasks such as authentication, log retrieval, or trust evaluation. Disruptions in this communication make it impossible to ensure timely verification. For example, an edge device operating in a remote agricultural field may temporarily lose its uplink to the cloud. During that period, the device cannot request updated trust scores, send activity logs for analysis, or receive policy updates, preventing ZTA mechanisms from functioning as intended.

To mitigate these constraints, recent research proposes augmenting ZTA with learning-driven components that integrate lightweight machine learning (ML) models into Policy Enforcement Points (PEPs) deployed at the edge [8]. These models support localized risk estimation and adaptive trust scoring, enabling continuous ZT enforcement even during communication disruptions. Such an approach requires ML models that are simultaneously accurate, predictable, resource-efficient, and deployable across heterogeneous hardware platforms. However, a key challenge in this domain is the lack of a standardized, reproducible methodology for assessing lightweight ML intrusion-detection models under ZT constraints. Despite advancements in research and the development of lightweight intrusion detection models, deploying them in ZTA requires a suitable execution environment that ensures secure isolation with minimal latency and resource consumption. Traditional container runtimes, such as Docker, incur hundreds of milliseconds of cold-start delays and high memory usage [9], [10], [11], making them unsuitable for edge devices with limited resources. WebAssembly (Wasm), on the other hand, provides a lighter-weight alternative for sandboxed execution, with roughly 95–99% shorter cold-start times and over 5x lower memory usage compared with container-based serverless frameworks, while maintaining near-native execution speed [11], [12], [13]. Moreover, recent experiments have shown that ML workloads can execute efficiently within Wasm runtimes [14].

To address the existing gap, this work extends previous work [8] with an evaluation framework for lightweight intrusion detection models compiled to the Wasm standard in the ZT architecture. The proposed framework introduces three layers: compilation, execution, and measurement, to enable reproducible benchmarking across heterogeneous models and Wasm runtimes. Using this framework, we evaluate a representative set of traditional and deep learning approaches on the UNSW-NB15 dataset [15], analyzing trade-offs between detection effectiveness, computational overhead, inference latency, memory usage, and binary footprint.

The contributions of this work are as follows:

- We design a Wasm-based testing framework to evaluate lightweight intrusion detection models (e.g., DT, RF, KNN, LR, ANN, and CNN variants) in ZT edge environments.
- We instantiate the framework using Rust, Wasmtime, and UNSW-NB15 to enable reproducible and technology-agnostic experiments.
- We systematically evaluate seven lightweight ML and DL models and provide a detailed analysis of the accuracy-efficiency trade-offs relevant to ZT enforcement.

The remainder of this article is organized as follows. Section II reviews the background and related work, including DCCS security, ZT principles, lightweight IDS techniques, and the UNSW-NB15 dataset. Section III details the architecture of the proposed learning-driven ZT framework and its implementation. Section IV presents the experimental setup and discusses the evaluation results. Finally, Section V concludes the article and outlines directions for future work.

## II. BACKGROUND AND RELATED WORK

In this section, we provide a brief overview of DCCS, the ZT security paradigm, and the role of IDS in this context. Additionally, we provide an overview of IDS datasets and the technical foundations of Wasm and Rust, enabling lightweight learning models on edge devices.

### A. DISTRIBUTED COMPUTING CONTINUUM SYSTEMS

Over the past decade, cloud infrastructures have become the dominant approach to application deployment, offering large-scale computing resources, elasticity, and scalability. Yet more and more use cases in distributed applications require processing closer to data sources. For instance, a smart city with a smart energy grid or interconnected personalized health systems. To meet this requirement, applications integrate intermediate fog nodes and peripheral edge devices into their infrastructure and connect them to the cloud, forming a DCCS [1], [16].

#### 1) ARCHITECTURAL VIEW

A DCCS represents a unified computing fabric spanning over three computing tiers, illustrated in Figure 1. At the top is the central cloud tier encompassing abundant resources available in cloud data centers with global network visibility. DCCS applications, therefore, perform heavy computation in this layer, such as conducting large-scale analyses of data aggregated by the fog tier. At the fog tier, devices such as gateways or regional microdata centers serve as an intermediate layer between the cloud and the edge tier. They

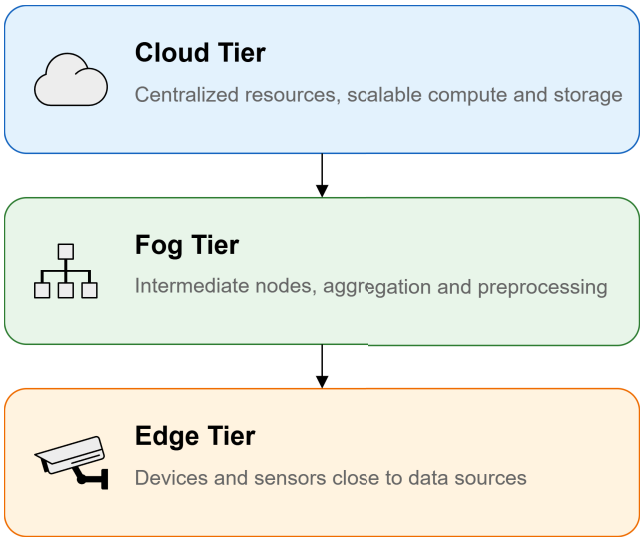



**FIGURE 1.** Conceptual view of the cloud, fog, and edge tiers within distributed computing continuum systems.

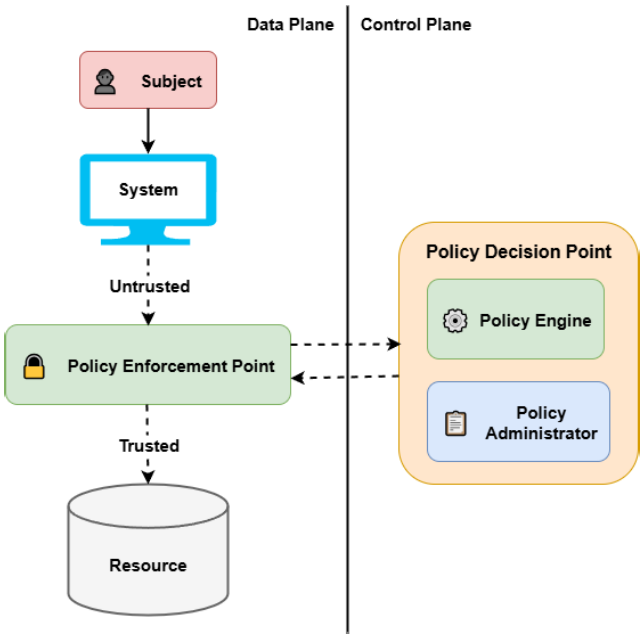



**FIGURE 2.** Core components of the ZTA introduced by NIST [7].

aggregate and process edge-tier data for the cloud tier. At the remote edge tier, diverse and often resource-constrained devices interact with the physical world, generating data from sensors, meters, cameras, and drones [17].

#### 2) CHARACTERISTICS AND CHALLENGES

The characteristics of DCCS differ significantly from those of more common cloud-centric infrastructures. At scale, a DCCS may encompass thousands or millions of devices of varying architectures, capacities, and energy availability. Additionally, devices may appear and disappear unexpectedly due to unstable or fluctuating connections. Furthermore, sudden shifts in workloads due to external factors, such as weather events or social factors, can lead to system congestion and affect the stability and performance of DCCS [18]. Classical cloud management approaches for handling these sudden workload changes, particularly threshold-based elasticity, are less effective in these complex adaptive systems [17].

### *B. LEARNING-DRIVEN ZT FRAMEWORK*

In traditional enterprise security models, only perimeter-based defenses are in place, namely, firewalls and virtual private networks. In these types of defenses, once a system or user is granted access to the network, they are implicitly trusted without further verification, making malicious activities, such as lateral movement within the network, often undetectable once access is granted. Even worse is the case of modern distributed applications, where not only are system perimeters secured, but also these organizational boundaries are often blurred and cannot be clearly defined [7].

The ZTA comprises three core components: the Policy Engine (PE), the Policy Administrator (PA), and the Policy Enforcement Point (PEP), and their interactions are best illustrated through the lifecycle of an access request [7]. To start, a subject, which denotes entities such as users, devices, or services, makes an access request to a resource. This request is then first passed through the PEP to the PA, which in turn consults the PE, which is ultimately responsible for making access decisions. For this, the PE draws on enterprise policies and contextual data (e.g., user role, device health, time of day, or recent activity), sourced from identity providers, activity logs, or threat intelligence feeds. After determining whether access should be granted to the subject, the PE notifies the PA of its decision, and the PA then translates the PE's ruling into action by directing the PEP accordingly. In case of an access denial by the PE, the PEP blocks or terminates the session between the subject and the protected resource. Upon granting access, the PA configures communication paths or issues tokens that permit the subject to access resources. The PEP then enables the connection and continues to monitor it, ready to terminate it again should conditions change.

The ZTA introduced by NIST formalizes a shift from perimeter-based defenses to continuous verification of all access requests, which is effective in centralized infrastructures (depicted in Figure 2). Nonetheless, the direct application of ZTA to heterogeneous and large-scale DCCS poses several issues, particularly, the edge tier of DCCS is both the most exposed to threats and the most constrained in terms of resources, factors that complicate the enforcement of ZT principles [7], [8]. Limitations, including limited resources, connectivity, and visibility, fundamentally restrict the effectiveness of centralized ZT enforcement in DCCS. To this end, we initially proposed a ZT learning-driven framework that introduces additional learning and resource

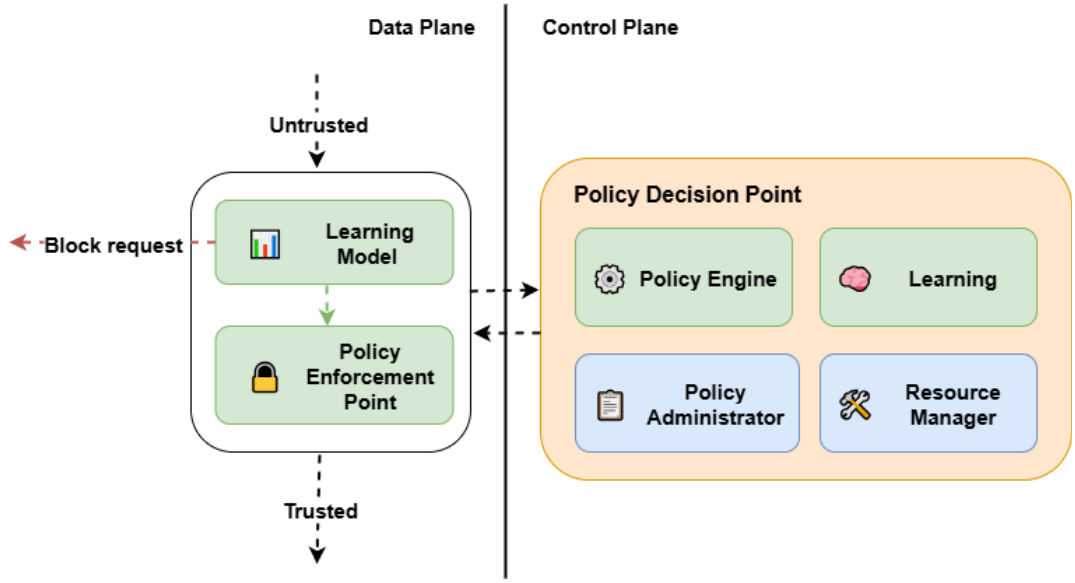



**FIGURE 3.** Learning-driven ZT framework for DCCS [8]. The baseline NIST architecture (Policy Engine, Policy Administrator, and Policy Enforcement Points) is expanded with a learning component and a resource management component.

management components into its conceptual architecture, as depicted in Figure 3.

A new learning component is introduced into the PDP, tasked with adapting its decision logic to evolving access patterns and threat behaviors, thereby improving the overall quality and consistency of trust decisions. Complementing this, lightweight learning models are embedded directly into PEPs, enabling local, autonomous access request classification even during intermittent connectivity with the PDP. They rely on available local data, such as historical logs, during inference to predict the trustworthiness of incoming requests. Our proposed approach emphasizes the importance of lightweight learning approaches in this context, as the resource constraints of edge DCCS environments necessitate models that optimize available resources and minimize inference latency without compromising accuracy.

The core capabilities of the second new component encompass the discovery and continuous monitoring of available resources, and the addressing of issues arising from the high heterogeneity of devices in DCCS. In addition, the resource management component orchestrates the placement of software components and configures and distributes the learning models. While most DCCS tasks run in the cloud to leverage abundant resources there, selected functionalities can be deployed across the computing continuum as edge functions to meet local demands. For example, compute-intensive model training may be performed in the cloud, whereas lightweight inference tasks are executed closer to the edge to reduce latency and communication overhead.

In the extended learning-driven ZTA, the PEP and the PDP continue to maintain a bidirectional decision flow, similar to the NIST ZTA. The additional embedded learning model at the PEP continuously evaluates incoming requests and estimates their trustworthiness. Once a connection to the PDP is established, these locally derived trust scores are forwarded to the PDP, which ultimately decides whether to grant access. However, when the PEP has limited or no connection to the PDP, as may be common in DCCS, the PEP relies solely on its local model to make autonomous access decisions.

This fallback mechanism ensures uninterrupted ZT enforcement at the edge, but also introduces several risks, with the primary one being the possibility of incorrect or inconsistent access decisions. Prominent attacks in this context try to compromise the integrity and reliability of the learning process by poisoning the model during training with manipulated data to distort the decision boundary, coupled with crafted adversarial inputs to mislead the predictions during inference [19]. Since PEPs are often located on the remote edge, other potential attack vectors include physical tampering with the device or extracting the model to modify or reverse-engineer the inference logic. Another risk arises from privacy and confidentiality concerns, when learned representations could expose sensitive user or device information. Further risks include limited explainability and auditability of local model decisions, synchronization inconsistencies between distributed components that cause models to drift from the PDP, and cascading errors during automated model updates. To mitigate these risks and preserve the advantages of the learning-driven ZTA, high-quality, well-curated training data is required, as well as continuous monitoring for abnormal model behavior and clear operational boundaries that restrict the autonomy of edge-deployed models. While

these additional components to the NIST ZTA define the conceptual foundations of learning-driven ZT within the computing continuum, the question of how such concepts can be systematically evaluated in practice remains largely unexplored. In particular, the proposed learning component requires lightweight models that are not only accurate but also feasible to deploy on constrained devices. The resource management component emphasizes the need for a uniform and portable runtime environment.

### C. INTRUSION DETECTION SYSTEMS AND DATASETS

Intrusion Detection Systems (IDS) are a critical layer of defense against cyber threats [20]. IDS works by identifying malicious activity within computer systems and networks using various techniques before any damage can be caused, thereby protecting the confidentiality, integrity, and availability of these systems.

Multiple taxonomies are used to categorize IDS. IDS can be distinguished by their data sources, grouping them into Host-based IDS (HIDS) and Network-based IDS (NIDS) [21]. As their name suggests, HIDS operate within a host computer and analyze the available data there, often various audit records and log files of the system, firewall, applications, or databases. HIDS main advantage lies in detecting insider threats and attacks that do not generate any network traffic, at the cost of consuming valuable host resources and the need to install and maintain the IDS individually on each machine. NIDS, on the other hand, monitor traffic across entire networks using packet capture, NetFlow, or management protocols as input sources. Due to their location, they often cover multiple hosts simultaneously and can inspect a broad range of network protocols, reporting on a wide selection of network-based attacks. Nevertheless, NIDS have difficulties handling encrypted traffic or high-speed networks, and they cannot detect host-specific intrusions, which are among their disadvantages. In practice, both HIDS and NIDS are often deployed together to provide complementary coverage [22].

In recent years, numerous datasets have been used for training and evaluating IDS. Given that the effectiveness of an IDS depends heavily on the dataset, they hence play a key role in IDS research. In the following, we provide an overview of widely used datasets in the field, accompanied by a comparative table (Table 1) that highlights their key characteristics, including dataset size, number of features, attack diversity, data imbalance, and degree of real-world representation. One of the earliest benchmark datasets for IDS is the DARPA '98 (Defense Advanced Research Projects Agency) dataset, as well as its later adaptation, the KDD Cup 1999 (Knowledge Discovery and Data Mining) dataset. These datasets were collected in controlled environments and contain labeled connection records with both normal and malicious activity. Both of these datasets were criticized for the redundancy and unrealistic distributions of their records, and due to their age, are now widely regarded as outdated and have seen a decline in adoption [22], [24]. Furthermore, the NSL-KDD (National Security Laboratory) was created to address some of KDD's shortcomings. Researchers removed redundant records while maintaining the same feature set and attack categories, resulting in a more balanced overall dataset. Although the NSL-KDD is widely used in machine learning research, it inherited some outdated characteristics from the original KDD and DARPA datasets.

The UNSW-NB15 dataset was introduced at the Australian Centre for Cyber Security to overcome the shortcomings of the commonly used legacy IDS datasets, KDD and NSL-KDD [15]. By contrast to these outdated datasets, the UNSW-NB15 was designed to reflect contemporary network traffic and modern low-footprint threats. Using the IXIA PerfectStorm tool, the authors simulated regular traffic and nine different families of malicious activity: Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode, and Worms. Traffic was captured during two simulation runs, and later processed to produce labelled flows with 49 features per record, spanning four categories: basic connection properties, content information, temporal statistics, and connection-based patterns.

The Canadian Institute for Cybersecurity created the CIC-IDS dataset after concluding that most available IDS datasets were outdated and unreliable [25]. To provide a comprehensive, modern benchmark, they included up-to-date common attacks such as denial-of-service, brute-force, infiltration, and botnet scenarios. Furthermore, since IDS can be deployed across a wide range of security domains, datasets were created to support specific use cases. The Bot-IoT dataset, for example, concentrates on the IoT domain, encompassing large-scale IoT botnet activity and corresponding normal traffic [26]. On the other hand, the Kyoto 2006+ dataset belongs primarily to the web and enterprise network domain, created by capturing traffic traces from honeypots in an experiment to study HTTP-based intrusions with ongoing background network activity [27]. Lastly, AWID (Aegean Wi-Fi Intrusion Dataset) focuses on the wireless domain and contains IEEE 802.11 network traces to assess intrusion detection in Wi-Fi environments [28].

### D. WEBASSEMBLY FOR EDGE AND IoT CONTEXTS

Many modern web applications require fast, consistent performance. JavaScript is not always suitable for this purpose due to its unpredictable performance. This is why Wasm, a portable and low-level bytecode format, was created as a compilation target for the web. Its compact representation, efficient validation, and safe execution semantics enable it to run low-level code securely across heterogeneous devices. In addition, Wasm is not bound to a single programming model; instead, it abstracts over modern hardware in a language- and platform-independent manner. Wasm has therefore expanded well beyond the web into domains such as edge and IoT computing [12]. One key aspect of Wasm in

**TABLE 1.** Overview of widely used IDS datasets (adapted from [23]).

| Dataset | Size | Attack Diversity | Data Imbalance | Real-World Rep. | Source / Origin |
|---|---|---|---|---|---|
| DARPA'98 / KDD'99 | Large | Various | Present | Limited | DARPA, USA |
| NSL-KDD | Moderate | Diverse | Present | Limited | UNB, Canada |
| Kyoto 2006+ | Moderate | Limited | Present | Limited | Kyoto Univ., Japan |
| AWID | Moderate | Limited | Present | Limited | Aarhus Univ., Denmark |
| UNSW-NB15 | Large | Comprehensive | Present (moderate) | Moderate | UNSW, Australia |
| CIC-IDS2017 | Large | Comprehensive | Present | Limited | CIC, Canada |

**TABLE 2.** Distribution of normal and attack records in UNSW-NB15 [15].

| Class | Records | Proportion |
|---|---|---|
| Normal | 2,218,761 | 87.7% |
| Generic | 215,481 | 8.5% |
| Exploits | 33,393 | 1.3% |
| DoS | 16,353 | 0.6% |
| Reconnaissance | 13,987 | 0.6% |
| Analysis | 2,677 | 0.1% |
| Fuzzers | 24,246 | 1.0% |
| Backdoors | 2,329 | 0.1% |
| Shellcode | 1,511 | <0.1% |
| Worms | 174 | <0.1% |

the IoT context is its strong and lightweight security model. Wasm programs execute within a sandbox that isolates linear memory from the code space and the host environment, separating the Wasm execution from the rest of the system and safeguarding against memory corruption and arbitrary code execution [29]. Another advantage of Wasm is its significantly smaller binary file size compared to equivalent JavaScript, which facilitates faster cold starts and reduces transmission overhead.

To execute a Wasm module, a Wasm runtime is required. These runtimes differ primarily in how they interpret the module. One such method is to interpret and execute Wasm instructions directly without compilation. This ensures maximum portability of the module, as every instruction is interpreted at runtime. However, this comes at the cost of slower execution due to the high overhead compared to native code. With Ahead-of-Time (AOT) compilation, the runtime generates native machine code in advance, resulting in accelerated startup times since no interpretation is required during execution, while still maintaining some portability. The third method of Just-in-Time (JIT) compilation translates Wasm into machine code on demand during execution, striking a balance between the portability of direct interpretation and the performance of AOT [14]. Furthermore, a range of standalone Wasm runtimes is currently available, namely Wavm, Wasmer, Wasmtime, WasmEdge, and Wamr, each designed with different trade-offs between performance and portability [29]. Regarding the increased interest in running AI applications in Wasm, a recent comparative study by Khelifa et al. [14] of common Wasm runtimes has revealed that JIT-based runtimes such as Wasmtime and Wasmer generally achieve execution speeds closest to native code and scale reasonably well to medium-sized AI workloads. In contrast, lightweight runtimes like Wamr are optimized for IoT devices with strict memory constraints.

#### 1) RUST FOR WebAssembly AND MACHINE LEARNING

Rust is a modern programming language designed to combine high performance with strong memory safety, while still allowing developers to control low-level memory details. One key characteristic of Rust's memory safety guarantees is the ownership and borrowing compile-time checks, which eliminate the need for a garbage collector and prevent common errors, such as dangling pointers and buffer overflows. Since its inception, Rust has become a popular choice for both systems programming and security-sensitive domains [30]. Furthermore, Rust natively supports compilation to Wasm and a wide range of targets, foremost the wasm32 family. Rust-generated Wasm binaries are kept small because only the functions that are used by the program are compiled, and no runtime or garbage collector is included by default. For integrating Wasm modules into modern JavaScript web applications, the Rust Wasm ecosystem provides additional bridging tools, such as wasm-bindgen and wasm-pack, that automate the generation of JavaScript method bindings and package management.

Stemming from its strengths as a systems language, Rust has an expanding machine learning ecosystem. The two notable frameworks in the context of this paper are SmartCore [31] and Burn [32]. While SmartCore focuses on traditional algorithms, such as decision trees, random forests, and logistic regression, Burn complements it by supporting deep learning with backends for CUDA and WebGPU for both training and inference. Both of these frameworks are compatible with Wasm, extending the possible application scenarios for Rust machine learning workloads.

### E. RELATED WORK

The development of particularly lightweight intrusion detection systems has been extensively studied in recent years.

In [33], Golestani and Makaroff explored the detection performance of one-class classifiers for lightweight intrusion detection across multiple IoT datasets and found that supervised methods provided better accuracy and consistency than unsupervised methods. The resource consumption of selected lightweight IDS was measured by De Vivo and Liguori [34]. In their study, the authors simulated resource-constrained environments using Docker. They demonstrated that detection accuracy degrades under tight CPU and memory constraints, underscoring the need for careful trade-offs to keep IDSs feasible. Well-established IDS datasets are used in practice to determine the effectiveness of proposed IDS, as with the work of Priya et al. [35], who evaluated seven machine learning classifiers across the KDD, NSL-KDD, and UNSW-NB15 datasets. Their comparison concluded that Decision Trees achieved the most favorable balance of accuracy and efficiency among the tested models. Recent work has also focused on newer datasets, such as Fatima et al. [36], who introduced Li-IDS on the TON-IoT dataset, demonstrating that tree-based models remain effective when combined with feature selection to lower CPU and memory usage. Similarly, in [37], Gadewar et al. integrated genetic optimization into the APA-DDoS dataset to reduce the false-positive rate and improve the efficiency of multiple classifiers, including Long Short-Term Memory, Recurrent Neural Networks, Deep Markov Random Fields, and Ridge Regression. Extending this direction of research, Reddy et al. [38] proposed a swarm-optimized approach for the Multi-Step Cyber-Attack dataset, demonstrating that metaheuristic tuning further enhances the trade-off between efficiency and accuracy in tree-based intrusion detection for smart city IoT environments.

In contrast to the mentioned works, our specific contribution is to provide a lightweight IDS evaluation explicitly within a learning-driven ZT context and to offer a reusable testing framework that operationalizes this perspective. By compiling models to Wasm and systematically measuring accuracy, latency, memory, binary size, and instruction-level complexity, our framework establishes a standardized basis for comparison, directly linking evaluation results to ZT enforcement requirements in DCCS.

## III. DESIGN AND IMPLEMENTATION OF A WebAssembly-BASED LIGHTWEIGHT IDS TESTING FRAMEWORK

In this section, we describe our proposed extension approach (i.e., we will refer to it as a testing framework), which enables the systematic evaluation of lightweight ML models for localized intrusion detection.

### A. TESTING FRAMEWORK DESIGN

We operationalize an aspect of the learning component by proposing a testing framework for the lightweight learning models embedded in the PEPs. In Figure 4, we present our testing framework, which provides a structured and standardized environment that ensures comparability across

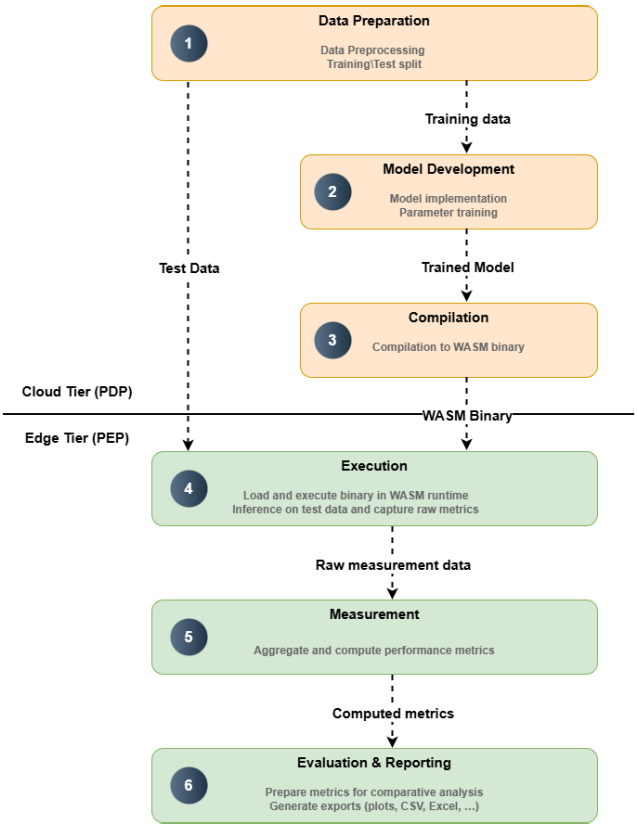



**FIGURE 4. Design and workflow of the testing framework. Steps (1)–(3) correspond to the compilation layer, step (4) to the execution layer, and steps (5)–(6) to the measurement layer.**

heterogeneous models and devices, making it reusable in future studies of learning-driven ZT enforcement. The proposed testing framework is designed in two tiers: (i) cloud tier (i.e., which includes three main steps, such as data preparation, model development, and compilation), and (ii) edge tier (i.e., which includes three layers, such as compilation, execution, and measurement).

In the compilation layer, trained learning models and their inference logic are compiled into portable, self-contained binaries. Wasm provides the technological foundation for this layer, making it particularly suitable for ZT enforcement scenarios at the edge. Alternatives like containerized or virtualized solutions often impose non-negligible runtime overheads, thereby unnecessarily increasing the execution resource requirements. As mentioned previously, various high-performance production-use programming languages can be compiled to Wasm. Prominently among them is Rust, which supports compilation to the wasm32-unknown-unknown target, as well as other widely used systems programming languages, including C, C++, and Go, which provide similar toolchains to bring existing codebases into the Wasm ecosystem.

The execution layer represents the on-device Wasm runtime execution environment and should therefore closely mirror the production deployments of the PEPs across

**TABLE 3.** Mapping of evaluation metrics to ZT requirements.

| Metric | ZT Requirement | Justification |
|---|---|---|
| Model accuracy | Reliability of enforcement | Models must make correct and reliable decisions at the PEP; recall avoids missed classifications, precision avoids unnecessary blocking, and F1 provides a balanced view. |
| Inference latency | Continuous verification | Decisions must be made in real-time to prevent unauthorized access or lateral movement. |
| Computational overhead | Efficiency of enforcement | Models must operate without exhausting CPU resources on constrained devices. |
| Memory usage | Deployability on constrained hardware | Enforcement mechanisms must coexist with application logic on devices with limited RAM. |
| Binary size | Scalability and updateability | Compact binaries enable frequent updates and large-scale distribution across heterogeneous devices. |

the computing continuum to assess model performance under conditions representative of practical operation. This process involves selecting a Wasm runtime tailored to the resource profile of the target tier (e.g., cloud, fog, or edge), as each runtime has performance and portability trade-offs.

Both the compilation and execution layers are designed to accurately mirror a real-world deployment scenario. In this context, the primary purpose of the measurement layer is to establish a systematic and comprehensive approach for evaluating the processed models, ensuring they perform effectively and reliably within this environment. To this end, the measurement layer captures standardized performance indicators that go beyond raw accuracy, providing a comparative view to determine whether a model is a viable candidate for ZT enforcement at the edge.

#### 1) METRICS FOR ZT AT THE EDGE

We carefully selected the following metrics for the measurement layer, each of which is critical for the operational viability of lightweight learning models in distributed PEP (as shown in Table 3).

First is the `Model accuracy`, captured through the standard metrics precision, recall, and the F1-score. This metric is essential for ZT since it quantifies how reliably a learning model makes correct decisions. Recall is particularly relevant for this security-sensitive context as false negatives allow unauthorized requests to access the protected resource behind the PEP. Thereby, low recall directly weakens the core ZT principle of *"never trust, always verify"* by undermining the verification process itself, as attacks that are not detected effectively bypass ZT enforcement. Precision, on the other hand, correlates with system availability, as incorrectly blocking legitimate traffic due to many false positives can cause operational disruptions. The F1-score combines both aspects and provides a balanced measure of overall decision quality, thus reflecting not only statistical performance but also the security posture of the enforcing system.

Next is `Inference latency`, the wall-clock time required for a model to classify a request or interaction with a resource. Within ZT, the inference latency is closely tied to the principle of continuous verification, as decisions must be made in real-time or near real-time. Particularly if traffic volumes are high and enforcement must occur locally, low latency ensures that verification remains effective without creating bottlenecks or usability issues. Models with high accuracy but long inference times, therefore, have limited value for edge-based ZT enforcement.

Another important aspect of edge ZT viability is the computational load during inference (i.e., `Computational overhead`), as a low processing burden suggests that models can operate efficiently in the PEP without monopolizing the often limited computational resources available. On the other hand, models with excessive resource loads per inference could be targeted by adversaries by flooding the PEP with requests, potentially exhausting available CPU time and linking computational overhead to denial-of-service resilience.

Similarly, `Memory usage` is a critical factor, as many devices in the edge tier of the computing continuum have only a few hundred megabytes of RAM, and in some cases significantly less, such as smart meters or embedded controllers. Learning models, therefore, must remain lightweight enough to coexist with the PEP's primary functions, as excessive memory consumption risks destabilizing the PEP and may even cause outages, ultimately undermining its reliability.

Finally, the `Binary size` of the compiled models determines how easily they can be distributed and updated across a large number of devices. From a ZT perspective, learning models need to be continually retrained to adapt to emerging attack techniques, and consequently, models must be frequently updated in production. Small compact binaries directly support scalability and rapid updates in environments with limited bandwidth, ensuring that PEP devices remain synchronized with the latest enforcement logic. Furthermore, Table 3 shows that relying solely on accuracy metrics is not enough to evaluate edge ZT enforcement. By expanding the evaluation criteria to include latency, computational

cost, memory usage, and binary size, the testing framework ensures that models are effective at detection while also practical for deployment in constrained, distributed PEP across the computing continuum.

### B. WORKFLOW

The overall workflow of the framework, as illustrated in Figure 4, proceeds through the three defined layers as a repeatable process that produces consistent, comparable results while remaining flexible with respect to model, dataset, and runtime. The process begins with:

1) **Data Preparation:** The first step is to prepare and preprocess the input data for the model training and inference. Data sources include, but are not limited to, established labeled datasets, synthetic samples, and raw production requests, depending on the specific learning model being tested.
2) **Model Development:** The learning model itself is first specified and implemented in a WebAssembly-compatible programming language. Then the model parameters are trained on prepared data that reflects real-world deployment scenarios, where training is typically performed in resource-rich environments.
3) **Compilation:** The next step is to compile the trained model parameters together with the inference logic into a compact Wasm binary, as described in the compilation layer.
4) **Execution:** The module is then executed within a selected Wasm runtime that corresponds to the target tier of the computing continuum as described in the execution layer. Inference is then run on the test data from the first step to record the raw metrics introduced in the measurement layer.
5) **Measurement:** The captured raw measurement data from the previous step is then aggregated to compute the final performance metrics.
6) **Evaluation & Reporting:** The final step in the framework collects all performance metrics and prepares them for comparative evaluation across different models. Optionally, this step could also create structured reports in CSV format and visualize trade-offs via plots.

While the presented testing framework design and workflow are defined in abstract terms of compilation, execution, and measurement, they must ultimately be grounded in concrete technological choices and adapted to the requirements of different deployment domains to become operational.

### C. TESTING FRAMEWORK IMPLEMENTATION

The proposed testing framework design was realized in practice by mapping each layer to specific technologies and configurations.[1] Rust was selected as the implementation language for the compilation layer due to its native support for Wasm, resulting in lightweight, dependency-free binaries. All model and inference code was therefore written in Rust and compiled to the wasm32-unknown-unknown target, which provides maximum compatibility across Wasm runtimes, with the caveat that no assumptions can be made about the host hardware, e.g., no GPU or file system support.

**LISTING 1. Rust host setup using Wasmtime.**

```
use wasmtime::{Engine, Store, Module, Instance,
    Linker};

fn host_model(lib_path: &str) ->
    anyhow::Result<Instance> {
   let engine = Engine::default();
   let store = Store::new(&engine);
   let module = Module::from_file(&engine,
       lib_path)?;
   let mut linker = Linker::new(&engine);
   let instance = linker.instantiate(&store,
       &module)?;
   Ok(instance)
}
```

The following compiler settings were chosen to optimize the resulting binaries for constrained devices:

- `opt-level = 3` for maximum speed optimizations
- `lto = true` for link-time optimizations
- `debug = false` to omit debug symbols and reduce binary size
- `panic = "abort"` to avoid stack unwinding and minimize overhead

Wasmtime was chosen as the Wasm runtime for the execution layer because it provides a production-ready host environment with fast Just-in-Time compilation, sandboxing, and support for instrumentation features such as fuel metering. Furthermore, a shared Rust-based application was created to host the Wasmtime runtime and load the learning models. A simplified excerpt of the host setup is shown in Listing 1. It should be noted that in production deployments of DCCS, the choice of hosting approach often depends on the target device's operating system and resource profile. On Linux, for example, the Wasm runtime can often be launched directly as a standalone process from a terminal. On constrained devices, the runtime is typically embedded in a C or C++ application for tight integration with the local hardware. In contrast, in enterprise deployments, it might be integrated into higher-level services written in Java or C#.

The measurement layer was implemented through a combination of host-side instrumentation and runtime-provided monitoring features. Notably, depending on the domain and available infrastructure of DCCS deployments, alternative measurement approaches, such as different profiling tools or runtime-specific instrumentation of other runtimes, for example, WasmEdge or WAMR, which provide their own profiling interfaces, can be similarly integrated. In our Rust-based host with a Wasmtime runtime, the metrics were realized as follows:

- **Model accuracy**, such as precision, recall, and F1-score, was calculated using Rust-based evaluation functions,

[1]Full implementation available at: https://gitlab.com/jonnygw1/learning-driven-zero-trust-testing-framework

implemented alongside the inference pipeline, to compare model predictions to the ground-truth labels of each dataset record.
- **Inference latency** was measured using Rust's Instant API to capture the wall-clock duration of each inference execution.
- **Computational overhead** was tracked via Wasmtime's built-in fuel metering capability to quantify instruction execution. A warm-up inference execution was performed before any measurement to exclude model initialization overhead.
- **Memory usage** was observed by monitoring the Wasm linear memory allocation during execution and using the runtime API to query memory size before and after inference.
- **Binary size** was recorded from the size of the compiled Wasm module at deployment time.

Having demonstrated how the framework design can be instantiated with concrete technologies while retaining flexibility for alternative choices, we now proceed to its application in the evaluation of a set of lightweight learning models.

## IV. RESULTS AND DISCUSSIONS

In this section, we apply the proposed testing framework to a set of state-of-the-art lightweight intrusion detection models. The evaluation follows the framework workflow described in Section III-A, which begins with data preprocessing and feature selection, and then implements traditional and deep learning models. We then collected the framework's measurement metrics and carried out a performance analysis. Finally, we interpreted the results in terms of trade-offs relevant to ZT enforcement at the edge.

### A. DATA PREPROCESSING

For training and testing, we used the pre-split, training, and test sets provided by the UNSW-NB15 dataset.[2] We adopted the feature subset established by [39] to reduce the input complexity. The authors obtained this subset using the chi-square criterion, which computes the statistical dependence between each feature and the class label, to identify the most impactful features for detection. The feature correlation matrix in Figure 5 further supports this method, showing several strongly correlated numeric features. The resulting subset comprises 20 discriminative features, including network flow characteristics, connection table features, and various aspects of network traffic, such as packet timing, protocol information, and connection statistics.

Min–max normalization was applied to scale all features to $[0, 1]$, preventing any single feature from dominating training and improving numerical stability. The resulting dataset comprised 175,341 training samples and 82,332 test samples, respectively, each with 20 normalized numerical

[2]https://research.unsw.edu.au/projects/unsw-nb15-dataset

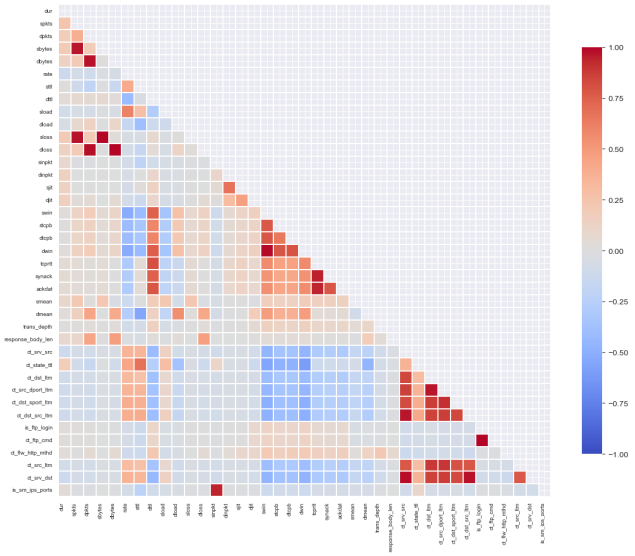


**FIGURE 5.** Feature correlation matrix (UNSW-NB15, Numeric Features).

features plus the binary target label. To preserve the integrity of the pre-split test set, all hyperparameter tuning was performed exclusively on the training portion. Furthermore, we employed stratified five-fold cross-validation (`StratifiedKFold` from `Scikit-learn`), using one fold for validation and four folds for training, thereby enabling a fair comparison of models across the subsequent layers.

### B. TRADITIONAL MACHINE LEARNING MODELS

As part of the framework evaluation, we included the following traditional machine learning models identified by Gnanasivam et al. [40]. These were chosen for their low computational complexity, fast reported inference speed, and ease of implementation, making them well-suited for resource-constrained edge scenarios. However, since the original configurations were not optimized for lightweight deployment, we performed a limited parameter search using the options available in the `smartcore` library, tuned on the five-fold split described in Section IV-A. Model parameters were selected based on the resulting precision and recall, with a slight preference for recall to reflect the security-sensitive nature of intrusion detection. All traditional learning model implementations used the `smartcore = { version = "0.4.1" }` Rust crate, a pure Rust machine learning library for classical models. Table 4 lists the chosen lightweight configurations for each architecture and the respective parameter search space we explored.

### C. DEEP LEARNING MODELS

As part of the framework evaluation, lightweight deep learning models were also implemented, using the `burn = { version = "0.17.1"}` Rust crate, a Rust-based deep learning library, which provides native support for

**TABLE 4.** Traditional ML models and hyperparameters.

| Model | Parameters and Search Space |
|---|---|
| DT | max_depth: 3 [3, 5, 8, 10, 15]<br>min_samples_split: 5 [5, 10, 25, 50]<br>min_samples_leaf: 1 [1, 5, 10, 20]<br>criterion: Entropy |
| RF | n_trees: 5 [5, 10, 25]<br>min_samples_split: 2 [2, 10]<br>min_samples_leaf: 1 [1, 5]<br>criterion: Entropy |
| KNN | k: 3 [2, 3, 5, 7, 9]<br>distance_metric: Euclidean<br>algorithm: CoverTree<br>weight: Uniform |
| LR | alpha: 1 [0, 0.001, 0.01, 0.1, 1, 10]<br>solver: LBFGS |

compilation to WebAssembly and integrates into the framework's compilation layer. To accelerate training, we utilized the `burn-cuda = { version = "0.17.1"}` Rust crate, which offers a CUDA-based GPU backend for `burn`.

We fine-tuned the learning rate of each model using a parameter search with stratified five-fold cross-validation, as described in Section IV-A, with the values `0.0001`, `0.001`, and `0.01`. Final learning rates were chosen based on precision and recall, with a slight preference for recall to reflect the security-sensitive nature of intrusion detection. For each model, the batch size was fixed at `128`, training ran for `30` epochs, and a fixed random seed of `42` was used to ensure reproducibility.

#### 1) CONVOLUTIONAL NEURAL NETWORK

The first architecture, a lightweight Convolutional Neural Network (CNN), was adopted from Ericson et al. [41] and was chosen for its detailed descriptions of the model architecture and reported success in intrusion detection. It consists of a convolutional layer with `1` input channel, `64` output channels, and a kernel size of `3`, followed by ReLU activation and a max-pooling layer with a pool size of `2`. The output is then flattened and passed through a fully connected layer of `50` neurons with ReLU activation, before reaching a final dense layer with a single output neuron. A sigmoid activation is applied to the final logit to produce the binary classification result. The Rust implementation of the CNN architecture is shown in Listing 2. Training was performed using the `Adam` optimizer with a learning rate of `0.0001`, as determined by the parameter search.

#### 2) CNN-BiLSTM

Building on this, the second architecture, a CNN-BiLSTM hybrid proposed by Jouhari and Guizani [42], was selected for its reported effectiveness in combining local feature extraction with sequential modeling. The model begins with a one-dimensional convolutional layer using `32` output channels, and a kernel size of `3`, followed by ReLU activation and max pooling with a pool size of `2`. The output is then passed to a bidirectional LSTM with hidden size `32` in each direction, followed by a dropout layer with a rate of `0.2`. After flattening, the result is processed by a fully connected layer of `25` neurons with ReLU activation and a final dense layer with sigmoid activation for binary classification. The corresponding Rust implementation is shown in Listing 3. Training was performed using the `Adam` optimizer with a learning rate of `0.001`, as determined by the parameter search.

**LISTING 2.** CNN model architecture implemented in Rust.

```
impl<B: Backend> CnnModel<B> {
    pub fn forward(&self, input: Tensor<B, 3>) ->
        Tensor<B, 2> {
        // 1D convolution with ReLU activation
        let x = self.conv.forward(input);
        let x = relu(x);

        // Max pooling to reduce dimensionality
        let x = self.pool.forward(x);

        // Flatten from 3D [batch, channel, seq] to
            2D [batch, features]
        let x = x.flatten(1, 2);

        // Fully connected layer with ReLU
            activation
        let x = self.fc1.forward(x);
        let x = relu(x);

        // Output layer with sigmoid for binary
            classification
        let x = self.fc2.forward(x);
        sigmoid(x)
    }
}
```

#### 3) ARTIFICIAL NEURAL NETWORK

Finally, as a contrasting lightweight baseline for the deep learning approaches, we designed a simple Artificial Neural Network (ANN). This architecture consists of three fully connected layers. The first with `64` neurons followed by ReLU activation and a dropout layer with rate `0.2`, the second with `32` neurons also followed by ReLU activation and dropout, and a final fully connected output layer with a single neuron and a sigmoid activation function for binary classification. The Rust implementation of this ANN is shown in Listing 4. Training was performed using the `Adam` optimizer with a learning rate of `0.0001`.

### D. PERFORMANCE ANALYSIS AND MEASUREMENT METHODOLOGY

In the following subsection, we present the results obtained through the measurement layer of the testing framework, covering detection accuracy, inference latency, computational overhead, memory usage, and binary size. The intrusion detection task is evaluated as a binary classification problem, with performance reported using the common metrics precision, recall, and F1-score. To minimize the impact of short-lived runtime noise (e.g., CPU spikes), all metrics

**LISTING 3.** CNN-BiLSTM model architecture implemented in Rust.

```
impl<B: Backend> CnnBilstmModel<B> {
    pub fn forward(&self, input: Tensor<B, 3>) ->
        Tensor<B, 2> {
        // Conv1D layer with ReLU activation
        let x = self.conv.forward(input);
    let x = relu(x);

    // MaxPooling to reduce dimensionality
    let x = self.pool.forward(x);

    // Reshape for BiLSTM
    let x = x.swap_dims(1, 2);

    // BiLSTM layer
    let (output, _) = self.bilstm.forward(x,
        None);

    // Dropout for regularization
    let output = self.dropout.forward(output);

    // Flatten for dense layers
    let output = output.flatten(1, 2);

    // Fully connected layers
    let x = self.fc1.forward(output);
    let x = relu(x);

    // Output layer with sigmoid for binary
        classification
    let x = self.fc2.forward(x);
    sigmoid(x)
  }
}
```

**LISTING 4.** Lightweight ANN architecture implemented in Rust.

```
impl<B: Backend> AnnModel<B> {
  pub fn forward(&self, input: Tensor<B, 2>) ->
      Tensor<B, 2> {
    // First hidden layer with ReLU and dropout
    let x = self.fc1.forward(input);
    let x = relu(x);
    let x = self.dropout1.forward(x);

    // Second hidden layer with ReLU and dropout
    let x = self.fc2.forward(x);
    let x = relu(x);
    let x = self.dropout2.forward(x);

    // Output layer with sigmoid activation
    let x = self.fc3.forward(x);
    sigmoid(x)
  }
}
```

were aggregated across the entire UNSW-NB15 test set (82,332 records), ensuring consistent results. All experiments were executed on three different devices: (i) a Raspberry Pi 4 (R4) running the Raspberry Pi OS 64-bit, (ii) an Nvidia Jetson Xavier NX (JX) running Jetpack 5.1.4, and (iii) a desktop-class machine (DM) equipped with an Intel Core i5-13600K CPU and 32 GB RAM running Ubuntu 20.04.

The metrics for the testing framework were collected in a batch-based inference setup, as shown in Figure 6. The Wasm host program writes a batch into Wasm memory and calls the Wasm module's `predict_batch` function. The module processes the batch, performs inference, and writes the predictions back to Wasm memory, which the host

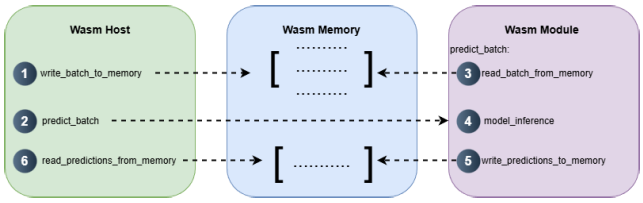



**FIGURE 6.** Inference setup.

**TABLE 5.** Inference latency of DT execution in native Rust and Wasm.

| Device | Runtime | mean (µs) | median (µs) | p95 (µs) |
|---|---|---|---|---|
| R4 | Native | 197.76 | 209.0 | 231.0 |
| R4 | Wasm | 275.72 | 292.0 | 313.7 |
| JX | Native | 202.69 | 206.0 | 240.9 |
| JX | Wasm | 191.20 | 207.0 | 290.9 |
| DM | Native | 29.69 | 31.0 | 37.0 |
| DM | Wasm | 46.78 | 40.0 | 71.0 |

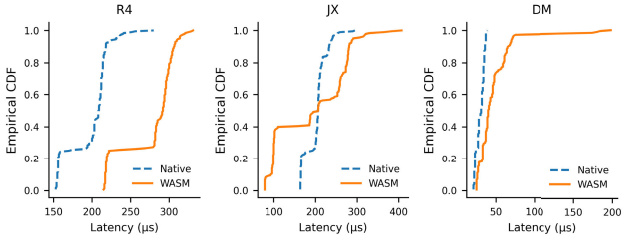



**FIGURE 7.** Inference latency of DT execution in native Rust and Wasm.

then retrieves. These metrics measure end-to-end inference performance, including reading and writing the input batch, creating data structures in Wasm, and handling predictions. However, they do not include dataset preparation, model initialization, or memory allocation. Thus, the computed overhead in Wasm fuel also represents the total fuel requirements for model inference.

### 1) WebAssembly OVERHEAD

To quantify the overhead of WebAssembly during model execution, we created a native Rust host that runs in parallel to the Wasm host program, with the same sample processing and inference logic. We chose the DT model as a representative case for this experiment because of its simple inference logic, which makes it suitable for isolating WebAssembly-specific execution overhead while minimizing the influence of model-dependent implementation details.

Table 5 and Figure 7 compare the end-to-end batch-based inference latency of the native Rust DT implementation and the Wasm DT implementation. On the R4, a clear shift to the right in the latency distribution indicates a consistent runtime overhead across both typical and tail latencies. In contrast, on the JX, the median latency remains nearly unchanged between the native Rust and Wasm execution. Along with a noticeable extension of the upper tail of the Wasm execution distribution, the overhead appears primarily as an increase in latency variance rather than

**TABLE 6.** No-operation Wasm invocation latency across devices.

| Device | mean ($\mu$s) | median ($\mu$s) | p95 ($\mu$s) |
|---|---|---|---|
| R4 | 14.46 | 14.00 | 15.00 |
| JX | 8.70 | 7.00 | 12.00 |
| DM | 1.00 | 1.00 | 1.00 |

a systematic slowdown. Similarly, the DM results show this as well, though with a more pronounced tail on the latency curve, indicating significant latency outliers in Wasm execution compared to native Rust. Overall, these results suggest that Wasm achieves near-native performance in typical cases on more powerful hardware, while introducing a predictable, uniform overhead on resource-constrained devices.

To isolate the fixed overhead of Wasm end-to-end inference, including method invocations and read and write operations, we designed a No-Operation experiment by replacing the predict function of the Wasm module with an empty execution and returning a fixed array of dummy predictions. Table 6 displays the results of this experiment on all 3 devices. Due to the absence of model execution, the latency distributions show only slight variance across devices, which explains the close alignment of the median and p95 values. Both the computational overhead of 10303 Wasm fuel and the Wasm memory footprint of 1216KiB in this experiment are identical across all devices, due to their platform-independent nature.

### 2) DETECTION ACCURACY AND TRAINING TIME

As highlighted in Table 7, RF achieved the highest overall F1-score (0.8800) and a strong balance between precision (0.7984) and recall (0.9802). This indicates that the ensemble-based method is well-suited to capture the diverse attack patterns in the UNSW-NB15 dataset. The closely related DT demonstrated a tendency to classify nearly all suspicious flows as attacks, achieving perfect recall (1.0), but significantly increased the likelihood of false positives, with low precision (0.7025). Lagging behind with the lowest F1-score of 0.7893 is LR, which reflects the limitations of linear decision boundaries in capturing the dataset's non-linear structure. KNN and CNN both obtained an identical F1-score of 0.8402, although with considerably different performance characteristics. While KNN exhibited solid recall (0.9073) and precision (0.7824), CNN reached the second-highest recall (0.9529) after RF.

The ANN attained an F1-score of 0.8614, placing it between RF and CNN. Its higher precision (0.8087) compared to CNN suggests that it produced fewer false alarms, though at a slight cost in recall. Notably, the CNN-BiLSTM was the only model with a higher precision (0.9041) than recall (0.7985), yet it still yielded a competitive F1-score of 0.8480. Given that the UNSW-NB15 dataset does not provide temporally ordered flows, the bidirectional LSTM layer of the BiLSTM had little sequential structure to exploit, which likely limited its effectiveness.

**TABLE 7.** Detection accuracy and training time on DM (i.e., a desktop-class machine).

| Model | Precision | Recall | F1 Score | Training Time |
|---|---|---|---|---|
| DT | 0.7025 | **1.0** | 0.8253 | 0s |
| RF | 0.7984 | 0.9802 | **0.8800** | 12s |
| KNN | 0.7824 | 0.9073 | 0.8402 | 0s |
| LR | 0.7222 | 0.8702 | 0.7893 | 3s |
| CNN | 0.7514 | 0.9529 | 0.8402 | 893s |
| CNN-BiLSTM | 0.9041 | 0.7985 | 0.8480 | 3305s |
| ANN | 0.8087 | 0.9214 | 0.8614 | 847s |

Moving on to the training times, Table 7 reveals clear distinctions between model families. Traditional methods were nearly instantaneous: DT required almost no training time due to its shallow parameters, and LR finished in 3 seconds, while RF completed in 12 seconds despite its ensemble structure. KNN requires no training time because all computation is deferred to the prediction phase. The models, part of the Deep learning family, in contrast, required substantially longer training cycles. ANN and CNN were both trained in approximately 14–15 minutes, while CNN-BiLSTM again took close to 55 minutes to fully train. It can be assumed that the substantially longer training time of the CNN-BiLSTM compared to purely feed-forward architectures is due to the BiLSTM layers' sequential nature. Overall, these results illustrate that no single model dominates across all accuracy dimensions. RF stands out as the most balanced choice, while DT guarantees full recall at the expense of false positives, and neural networks exhibit mixed trade-offs depending on the architecture.

### 3) INFERENCE LATENCY

Table 8 presents the end-to-end inference latency across all evaluated lightweight intrusion detection models and devices, revealing substantial differences attributable primarily to model complexity and, secondarily, to hardware capabilities. Latency spans a broad spectrum (i.e., from tens of microseconds for the simplest traditional models to nearly one second for the most computationally intensive deep learning architectures), highlighting the importance of computational efficiency for supporting ZT enforcement at the edge.

Across all devices, three clear latency tiers emerge. First, the ultra-lightweight DT and LR models consistently achieve the lowest inference times, with median latencies well below one millisecond even on resource-constrained hardware. On the DM, DT reaches a latency of 40 $\mu$s, while LR completes inference in 69 $\mu$s, and even on the R4, both models remain highly responsive. Their shallow decision logic and minimal resource requirements contribute to tight latency distributions, making them well-suited for predictable ZT enforcement. Second, the RF and ANN models form

**TABLE 8.** End-to-end inference latency of WASM-executed models across devices.

| Device | Model | mean ($\mu$s) | median ($\mu$s) | p95 ($\mu$s) |
|---|---|---|---|---|
| R4 | DT | 275.72 | 292.0 | 313.7 |
| R4 | LR | 427.48 | 426.0 | 449.7 |
| R4 | RF | 4121.04 | 4872.0 | 6611.7 |
| R4 | ANN | 6066.50 | 6060.0 | 6134.4 |
| R4 | CNN | 201416.99 | 201385.0 | 201745.4 |
| R4 | KNN | 583591.96 | 613627.0 | 827457.7 |
| R4 | CNN-BiLSTM | 831813.15 | 830948.0 | 837668.0 |
| JX | DT | 191.20 | 207.0 | 290.9 |
| JX | LR | 428.01 | 253.0 | 1011.3 |
| JX | RF | 1294.41 | 1312.0 | 2312.1 |
| JX | ANN | 3529.02 | 3345.0 | 4942.3 |
| JX | CNN | 74967.50 | 74833.0 | 78005.2 |
| JX | KNN | 283429.36 | 309923.0 | 369731.6 |
| JX | CNN-BiLSTM | 517526.59 | 522993.0 | 533379.9 |
| DM | DT | 46.78 | 40.0 | 71.0 |
| DM | LR | 111.37 | 69.0 | 278.4 |
| DM | RF | 528.07 | 604.0 | 890.3 |
| DM | ANN | 865.84 | 865.0 | 874.0 |
| DM | CNN | 32829.28 | 32862.0 | 33228.6 |
| DM | KNN | 74400.69 | 78481.0 | 101172.6 |
| DM | CNN-BiLSTM | 108285.72 | 107886.0 | 110408.5 |

a middle-tier of latency. Although more computationally demanding than DT and LR, both remain within manageable millisecond-level latencies. On R4, RF requires 4.87 ms (median) and ANN approximately 6.06 ms, while on the DM, these values drop to 0.604 ms and 0.865 ms, respectively. Despite modestly higher delays and increased tail latencies (i.e., particularly for RF due to ensemble aggregation), their performance remains within real-time verification thresholds. Third, the high-complexity models (i.e., CNN, KNN, and CNN-BiLSTM) exhibit latencies that are orders of magnitude higher. For instance, the CNN-BiLSTM reaches 830.9 ms (median) on R4 and 107.9 ms on DM, while KNN peaks at 613.6 ms on R4 and 78.5 ms on DM. These results reflect the computational depth of convolutional and recurrent architectures, as well as the input-dependent cost of model families such as KNN. Although still technically near real-time, these latencies severely constrain their suitability for highly time-sensitive ZT points of enforcement, especially in multi-request-per-second scenarios.

Figure 8 visualizes these trends and highlights the consistency of model ranking across hardware. DT and ANN show extremely tight spreads, demonstrating predictable behavior. LR and KNN, however, present noticeably larger tail deviations, with KNN particularly affected due to variance in distance computations. CNN-based models, while slow in absolute terms, display stable variance profiles, indicating consistent Wasm-level execution despite their computational intensity.

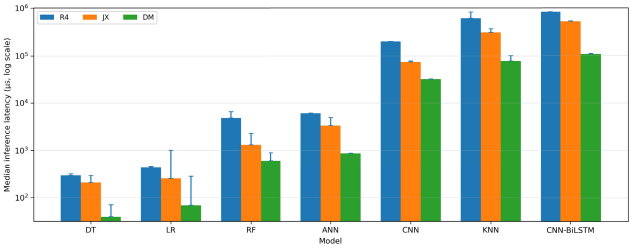



**FIGURE 8.** Median end-to-end inference latency for all models across devices.

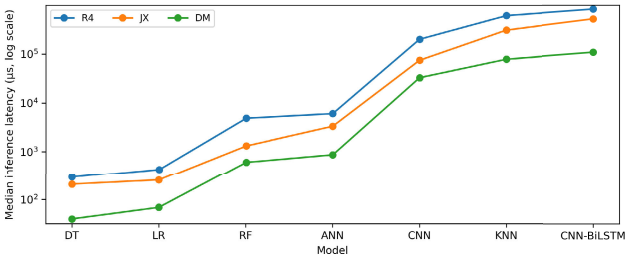



**FIGURE 9.** Hardware-dependent median end-to-end inference latency scaling per model.

Figure 9 examines cross-device scaling and shows that latency curves for different hardware remain approximately parallel on a logarithmic scale. This indicates that performance differences across devices are dominated by raw hardware capability rather than runtime-level irregularities or Wasm execution unpredictability. As a result, WebAssembly preserves deterministic scaling behavior: while faster processors yield proportionate improvements, the relative performance ordering of models remains unchanged.

Overall, the results show that while all models complete inference fast enough for certain forms of ZT enforcement, *predictability and worst-case latency are equally important*. DT, LR, RF, and ANN provide both strong stability and low latency, making them suitable for continuous or high-frequency verification at edge PEPs. In contrast, CNN, CNN-BiLSTM, and KNN require significantly more computational time and therefore impose non-trivial constraints on throughput, responsiveness, and deployment feasibility, particularly in severely resource-constrained edge devices.

#### 4) COMPUTATIONAL OVERHEAD

Table 9 shows the measured fuel consumption for all models, where "fuel" represents the number of instructions executed during inference, a runtime-agnostic measure of computational complexity provided by the Wasm metering interface.

Traditional approaches remained highly competitive, as they deliver decisions at very low computational cost. DT and LR consumed remarkably few fuel units, averaging only 0.902 and 1.942 million fuel units per inference (i.e., ≈902,000 and 1,942,000), respectively. Their control

**TABLE 9.** End-to-end computational overhead in Wasm fuel consumption (values in millions, $\times 10^6$).

| Model | mean | median | p95 |
|---|---|---|---|
| DT | 0.902 | 0.967 | 0.969 |
| LR | 1.942 | 1.945 | 1.950 |
| RF | 11.987 | 14.212 | 18.247 |
| ANN | 30.979 | 30.970 | 31.003 |
| CNN | 960.067 | 960.059 | 960.093 |
| KNN | 1405.194 | 1435.346 | 1942.872 |
| CNN-BiLSTM | 3217.151 | 3214.889 | 3233.204 |

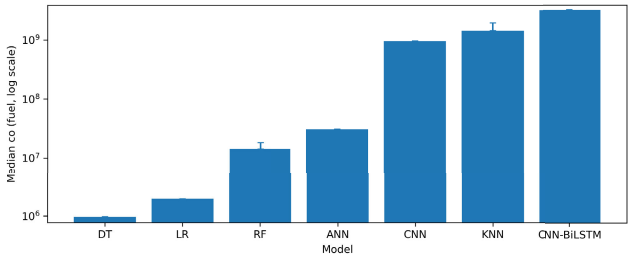



**FIGURE 10.** Median computational overhead measured in Wasm fuel consumption for all models across devices.

flow is shallow and predictable, which explains the minimal instruction counts. Even RF, despite combining many trees, required only 11.987 million fuel units on average (≈11,987,000).

The picture changes drastically for neural models. The ANN consumed approximately 30.979 million units per inference (≈30,979,000), already about 2.6× higher than RF. The CNN's mean fuel usage rose to 960.067 million units (≈960,067,000), while the CNN-BiLSTM reached 3,217.151 million units (≈3,217,151,000). This increase directly reflects the extra arithmetic operations and memory access patterns introduced by convolutional and recurrent layers. The BiLSTM, in particular, must perform sequential processing at each time step, which multiplies its instruction count; in other words, it pays a high computational price for a temporal structure that the static UNSW-NB15 dataset cannot fully exploit.

KNN displayed another kind of inefficiency. Its mean fuel consumption of 1,405.194 million units (≈1,405,194,000) is high, and the gap between its mean and tail behavior further highlights variability: the p95 climbs to 1,942.872 million units (≈1,942,872,000), with a median of 1,435.346 million (≈1,435,346,000). This volatility is evident in Figure 10, where the computational cost increases with the number of neighbors and their distances, leading to a broad, uneven distribution. From a ZT viewpoint, this unpredictability is risky. Enforcement mechanisms require consistent response times and stable resource utilization; otherwise, they may become unreliable under load.

**TABLE 10.** Wasm module binary size and max memory usage of Wasm-executed models.

| Model | Size (MiB) | Max memory (MiB) |
|---|---|---|
| DT | 0.06 | 1.25 |
| LR | 0.07 | 1.25 |
| ANN | 0.85 | 2.13 |
| RF | 1.11 | 3.94 |
| CNN | 1.06 | 14.25 |
| CNN-BiLSTM | 1.29 | 10.94 |
| KNN | 25.73 | 58.18 |

#### 5) MODULE SIZE AND MEMORY USAGE

Table 10 reports the sizes of the compiled Wasm modules together with the maximum memory usage observed during inference. The traditional machine learning models produced the most compact binaries, with DT requiring only 0.06 MiB (≈62,915 bytes) and LR 0.07 MiB (≈73,400 bytes). Both remain well under 0.1 MiB, which is remarkably small by modern software standards and easily distributable even in low-bandwidth environments. In contrast, the larger classical ensemble and neural models ranged from under 1 MiB to slightly above it: ANN compiled to 0.85 MiB (≈891,290 bytes), while RF, CNN, and CNN-BiLSTM produced binaries of 1.11 MiB (≈1,163,919 bytes), 1.06 MiB (≈1,111,491 bytes), and 1.29 MiB (≈1,352,663 bytes), respectively. These sizes are still well within a manageable range for frequent updates, but they represent a clear increase relative to DT/LR and can have a noticeable impact when distributing updates across large fleets of constrained edge devices.

KNN is an outlier, producing a binary of 25.73 MiB (≈26,979,860 bytes; ≈27 MB). This stems directly from its lazy-learning design, where the model must retain the full reference dataset (or an equivalent structure) to perform distance-based classification at inference. Such a footprint may be tolerable in small-scale laboratory settings. Still, for real-world deployment, it is highly impractical: distributing binaries of this size is slow and bandwidth-intensive in bandwidth-constrained networks, and storing tens of megabytes of model data on devices with limited flash capacity is often infeasible. For these reasons, KNN's otherwise acceptable accuracy and latency metrics are overshadowed by its unsuitability for deployment.

Memory usage results present a similar pattern. DT and LR both peaked at 1.25 MiB, suggesting a baseline runtime and execution-state overhead. ANN increased modestly to 2.13 MiB, while RF rose to 3.94 MiB, remaining within the limits of many contemporary edge devices. The neural convolutional/recurrent variants, however, were substantially more memory-intensive: CNN reached 14.25 MiB and

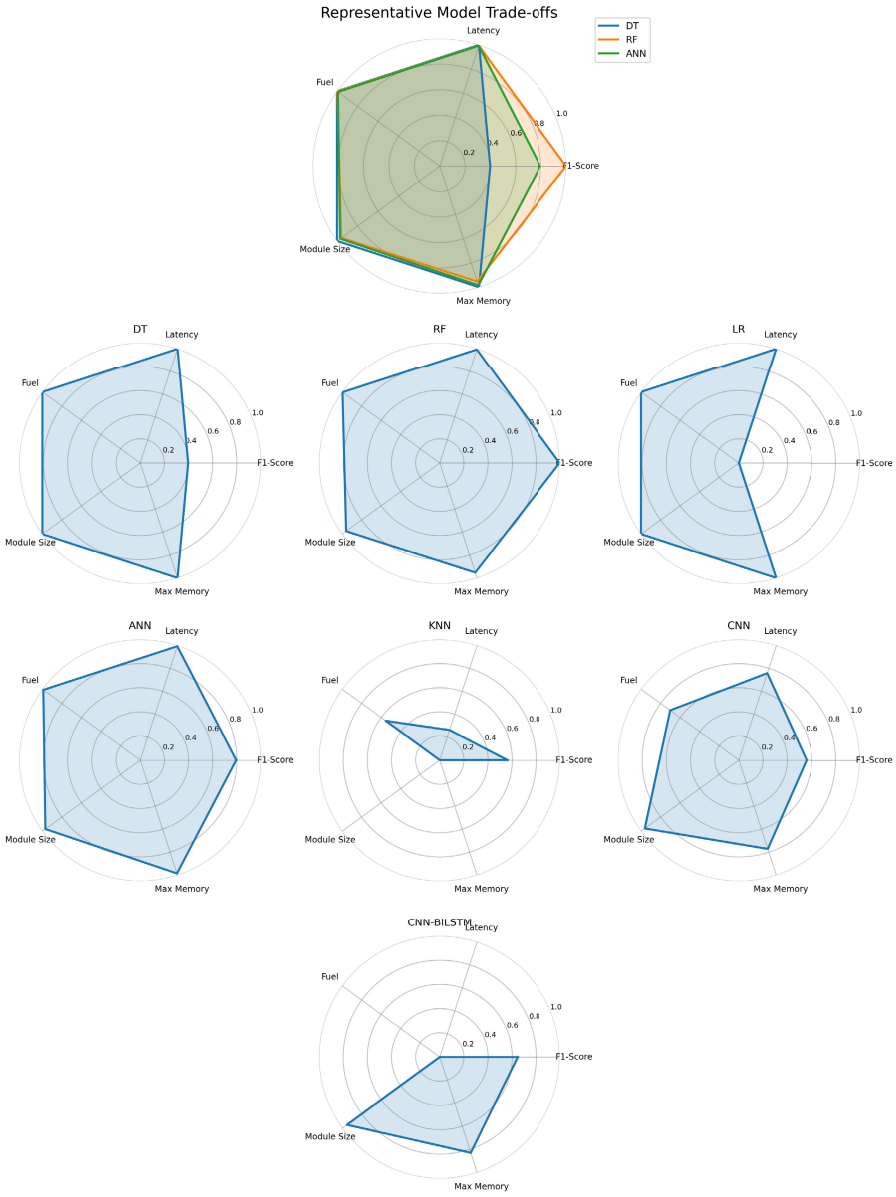



**FIGURE 11.** Multi-objective trade-offs across evaluated lightweight intrusion detection models, illustrating differences in detection performance, inference latency, computational overhead, memory usage, and binary size. Balanced candidates (DT, RF, ANN) are highlighted. For the device-dependent latency, the results from the most constrained platform, the R4, are used for a conservative trade-off representation.

CNN-BiLSTM 10.94 MiB, consistent with larger activation tensors and layer-wise intermediate buffers. Once again, KNN was the outlier, reaching a peak of 58.18 MiB, which rules out deployment on many low-end devices.

From a ZT perspective, both module size and memory usage are crucial, as enforcement mechanisms must be continuously updated without compromising standard device functionality. DT and LR excel in this respect, as they are both compact and memory-efficient. KNN, by contrast, illustrates the risks of overlooking deployability considerations, because despite acceptable accuracy and inference times, its storage and memory requirements alone can disqualify it from practical ZT enforcement.

### E. TRADE-OFF ANALYSIS

The performance analysis in the previous section reveals clear trade-offs among the models when viewed from a multi-objective standpoint. Figure 11 visualizes the trade-offs across the measured ZT testing framework metrics. Standing out as one of the most balanced configurations is the RF, achieving the highest F1-score with mid-range memory

usage while coming in third for inference latency and fuel consumption. The DT and LR remain as the most resource-efficient by a wide margin, offering the fastest inference, the fewest Wasm instructions during end-to-end inference, and sub-100kB Wasm module binaries. The deep learning models show a more varied profile. For instance, the ANN provides competitive detection quality at a moderate cost. In contrast, the CNN and, especially, the CNN-BiLSTM incur steep increases in inference time, fuel consumption, and module size. Among the most divergent trade-offs, the KNN achieved acceptable detection accuracy. However, it suffered from high inference latency, highly inconsistent computational overhead, and extreme module binary size and memory usage due to its dataset embedding implementation.

From a ZT edge enforcement perspective, we argue that both predictable operation and resource efficiency are as crucial as raw classification performance. Furthermore, PEPs must perform continuous verification under load and apply frequent policy and model updates to remain reliable even in degraded connectivity conditions. The DT and LR excel in this regard as they offer the lowest latencies together with tight fuel distributions, compact binaries, and low memory footprints, although the LR introduce occasionally introduces fluctuating high latencies, which complicates ZT enforcement, as a model that is occasionally fast but unpredictably slow some of the time may complicate real-time guarantees, even if its average performance remains acceptable. The RF and ANN require slightly higher inference latencies and computational overheads, but still exhibit all the characteristics that support consistent, repeatable behavior across the diverse edge DCCS environment. The deep learning models CNN and CNN-BiLSTM, although accurate, introduce significantly higher inference latencies and overhead. The trade-offs of our simple KNN implementation illustrate the risks of ignoring deployability when evaluating learning models in this context, because, despite acceptable accuracy, KNN's resource profile is incompatible with continuous verification on constrained devices. Its extreme binary size, due to a fully embedded dataset, is highly impractical, as the distribution of binaries of this size would be slow and costly, particularly in bandwidth-constrained networks. This KNN implementation's memory usage and highly variable computational cost disqualify it for practical deployment on most constrained edge devices.

### F. LIMITATIONS

We acknowledge that the model evaluation was limited to the UNSW-NB15 dataset, which inherently biases its traffic composition, attack distributions, synthetic flow generation, and fixed network topology. Furthermore, each model was evaluated using the same chi-square-based feature subset to ensure comparability, leaving alternative feature sets unexplored. Moreover, the scope of the evaluated IDS model was limited to a set of lightweight traditional and deep learning models from the literature, which provided sufficient details for implementation. This excluded newer architectures, such as transformer- and graph-based approaches, as well as federated learning architectures. Likewise, the scope of threat scenarios considered was limited to the attacks included in the UNSW-NB15 dataset, which, while modern and widely adopted, remains a synthetic laboratory dataset. Furthermore, although discussed, evaluation under adversarial machine learning conditions, such as evasion and poisoning attacks, was not assessed.

The hyperparameter optimization carried out in this paper was limited to a targeted search space due to training time and computing resource constraints, as well as by the configurable parameter support of the implementation libraries. Further tuning, future Rust crate versions, or more advanced optimization strategies might yield different trade-offs between detection accuracy and efficiency. Finally, while the testing framework was implemented in Rust and Wasmtime, this is only one implementation possibility. Other Wasm-compatible languages, alternative runtimes, or different measurement backends are left for future work.

## V. CONCLUSION

In this paper, we extended our framework [8] with an evaluation approach for lightweight intrusion detection models to support learning-driven Zero Trust enforcement at the edge. By structuring the evaluation into compilation, execution, and measurement layers and implementing the approach in Rust and WebAssembly, we demonstrated a portable, reproducible method for assessing model suitability across heterogeneous edge devices. Evaluation of seven lightweight models on the UNSW-NB15 dataset revealed clear trade-offs: Random Forest offered the strongest overall balance between accuracy and efficiency, while Decision Tree and Logistic Regression models delivered highly predictable, low-overhead inference suitable for ultra-constrained devices. ANN provided a practical middle ground, whereas CNN-based and KNN models incurred substantial computational and memory costs that limit their deployability.

## ACKNOWLEDGMENT

The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

## REFERENCES

[1] V. Casamayor Pujol, P. K. Donta, A. Morichetta, I. Murturi, and S. Dustdar, "Distributed computing continuum systems–opportunities and research challenges," in *Proc. Int. Conf. Service-Oriented Comput.*, 2022, pp. 405–407.

[2] V. C. Pujol, P. K. Donta, A. Morichetta, I. Murturi, and S. Dustdar, "Edge intelligence—Research opportunities for distributed computing continuum systems," *IEEE Internet Comput.*, vol. 27, no. 4, pp. 53–74, Jul. 2023.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.

[4] A. Alwarafy, K. A. Al-Thelaya, M. Abdallah, J. Schneider, and M. Hamdi, "A survey on security and privacy issues in edge-computing-assisted Internet of Things," *IEEE Internet Things J.*, vol. 8, no. 6, pp. 4004–4022, Mar. 2021.

[5] (Jul. 2024). *EU Cybersecurity Risk Evaluation and Scenarios for the Telecommunications and Electricity Sectors*. [Online]. Available: https://digital-strategy.ec.europa.eu/en/news/

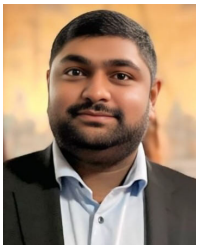
[6] *Cybersecurity Topic Page*. Accessed: Aug. 29, 2025. [Online]. Available: https://www.energy.gov/topics/cybersecurity
[7] S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, NIST Special Publication 800-207, 2020. Accessed: Feb. 27, 2026. [Online]. Available: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=930420
[8] I. Murturi, P. K. Donta, V. C. Pujol, A. Morichetta, and S. Dustdar, "Learning-driven zero trust in distributed computing continuum systems," in *Proc. IEEE Int. Conf. Dependable, Autonomic Secure Comput., Int. Conf. Pervasive Intell. Comput., Int. Conf. Cloud Big Data Comput., Int. Conf. Cyber Sci. Technol. Congr. (DASC/PiCom/CBDCom/CyberSciTech)*, Nov. 2023, pp. 0044–0049, doi: 10.1109/DASC/PICOM/CBDCOM/CY59711.2023.10361352.
[9] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 133–145.
[10] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion (UCC Companion)*, Dec. 2018, pp. 181–188.
[11] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with WebAssembly runtimes," in *Proc. 22nd IEEE Int. Symp. Cluster, Cloud Internet Comput. (CCGrid)*, May 2022, pp. 140–149.
[12] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to speed with WebAssembly," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 2017, pp. 185–200.
[13] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs. native code," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 107–120.
[14] S. E. Khelifa, M. Bagaa, A. O. Messaoud, and A. Ksentini, "Case study of WebAssembly runtimes for AI applications on the edge," in *Proc. Global Inf. Infrastruct. Netw. Symp. (GIIS)*, Feb. 2024, pp. 1–6.
[15] N. Moustafa and J. Slay, "UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *Proc. Mil. Commun. Inf. Syst. Conf. (MilCIS)*, Nov. 2015, pp. 1–6.
[16] P. K. Donta, I. Murturi, V. Casamayor Pujol, B. Sedlak, and S. Dustdar, "Exploring the potential of distributed computing continuum systems," *Computers*, vol. 12, no. 10, p. 198, Oct. 2023.
[17] I. Murturi and S. Dustdar, "DECENT: A decentralized configurator for controlling elasticity in dynamic edge networks," *ACM Trans. Internet Technol.*, vol. 22, no. 3, pp. 1–21, Aug. 2022.
[18] I. Murturi, A. Egyed, and S. Dustdar, "Utilizing AI planning on the edge," *IEEE Internet Comput.*, vol. 26, no. 2, pp. 28–35, Mar. 2022.
[19] D. A. Mihaylova, "Adversarial machine learning attacks against network intrusion detection systems: Classification analysis," in *Proc. 60th Int. Scientific Conf. Inf., Commun. Energy Syst. Technol. (ICEST)*, Jun. 2025, pp. 1–4.
[20] A. Hozouri, A. Mirzaei, and M. Effatparvar, "A comprehensive survey on intrusion detection systems with advances in machine learning, deep learning and emerging cybersecurity challenges," *Discover Artif. Intell.*, vol. 5, no. 1, p. 314, Nov. 2025.
[21] L. Diana, P. Dini, and D. Paolini, "Overview on intrusion detection systems for computers networking security," *Computers*, vol. 14, no. 3, p. 87, Mar. 2025.
[22] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: Techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, p. 20, Dec. 2019.
[23] S. Rani and S. Kumar, "A comprehensive analysis of intrusion detection datasets: Evaluation, challenges, and insights," in *Proc. 7th Int. Conf. Image Inf. Process. (ICIIP)*, Nov. 2023, pp. 547–551.
[24] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *Proc. IEEE Symp. Comput. Intell. Secur. Defense Appl.*, Jul. 2009, pp. 1–6.
[25] I. Sharafaldin, A. Habibi Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," in *Proc. 4th Int. Conf. Inf. Syst. Secur. Privacy*, 2018, pp. 108–116.
[26] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset," 2018, *arXiv:1811.00701*.
[27] J. Song, H. Takakura, Y. Okabe, M. Eto, D. Inoue, and K. Nakao, "Statistical analysis of honeypot data and building of Kyoto 2006+ dataset for NIDS evaluation," in *Proc. 1st Workshop Building Anal. Datasets Gathering Exper. Returns Secur.*, Apr. 2011, pp. 29–36.
[28] C. Kolias, G. Kambourakis, A. Stavrou, and S. Gritzalis, "Intrusion detection in 802.11 networks: Empirical evaluation of threats and a public dataset," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 1, pp. 184–208, 1st Quart., 2016.
[29] Y. Zhang, M. Liu, H. Wang, Y. Ma, G. Huang, and X. Liu, "Research on WebAssembly runtimes: A survey," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 8, pp. 1–47, Nov. 2025.
[30] *The Rust Programming Language*. Accessed: Aug. 16, 2025. [Online]. Available: https://www.rust-lang.org/
[31] *Smartcore: Machine Learning Library in Rust*. Accessed: Aug. 16, 2025. [Online]. Available: https://smartcorelib.org/
[32] *Burn: A Flexible and Extensible Deep Learning Framework for Rust*. Accessed: Aug. 16, 2025. [Online]. Available: https://burn.dev/
[33] S. Golestani and D. Makaroff, "Exploring unsupervised one-class classifiers for lightweight intrusion detection in IoT systems," in *Proc. 20th Int. Conf. Distrib. Comput. Smart Syst. Internet Things (DCOSS-IoT)*, Apr. 2024, pp. 234–238.
[34] S. De Vivo and P. Liguori, "Simulation environment for the evaluation of lightweight intrusion detection systems," in *Proc. IEEE 34th Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Oct. 2023, pp. 132–135.
[35] D. D. Priya, A. Kiran, and P. Purushotham, "Lightweight intrusion detection system(L-IDS) for the Internet of Things," in *Proc. Int. Conf. Advancements Smart, Secure Intell. Comput. (ASSIC)*, Nov. 2022, pp. 1–4.
[36] M. Fatima, O. Rehman, and I. M. H. Rehman, "Li-IDS: An approach towards a lightweight IDS for resource-constrained IoT," in *Proc. Int. Conf. Smart Appl., Commun. Netw. (SmartNets)*, Jul. 2023, pp. 1–6.
[37] A. B. Gadewar, R. V. Patil, S. A. Mahajan, and L. V. Patil, "Automated network vulnerability detection in IoT: Lightweight IDS approaches for enhanced security," in *Proc. 2nd Int. Conf. Adv. Comput. Commun. Technol. (ICACCTech)*, Nov. 2024, pp. 612–618.
[38] D. K. K. Reddy, J. Nayak, and M. Mishra, "Intrusion detection system in IoT smart city environment using tree-based approach with swarm-based optimization for multi-step cyber-attack dataset," in *Proc. 1st Int. Conf. Circuits*, 2023, pp. 1–6.
[39] M. Jouhari, H. Benaddi, and K. Ibrahimi, "Efficient intrusion detection: Combining $X^2$ feature selection with CNN-BiLSTM on the UNSW-NB15 dataset," 2024, *arXiv:2407.14945*.
[40] S. Gnanasivam, D. Tveter, and N. Dinh, "Performance evaluation of network intrusion detection using machine learning," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, Jan. 2024, pp. 1–6.
[41] A. Ericson, S. Forsström, and K. Thar, "IIoT intrusion detection using lightweight deep learning models on edge devices," in *Proc. IEEE 20th Int. Conf. Factory Commun. Syst. (WFCS)*, Apr. 2024, pp. 1–8.
[42] M. Jouhari and M. Guizani, "Lightweight CNN-BiLSTM based intrusion detection systems for resource-constrained IoT devices," in *Proc. Int. Wireless Commun. Mobile Comput. (IWCMC)*, May 2024, pp. 1558–1563.

**JONATHAN WEBER** received the B.Sc. and M.Sc. degrees from the TU Wien, Vienna, Austria. His current research interests include distributed and cloud systems, security, and applied machine learning.

**ILIR MURTURI** (Senior Member, IEEE) received the M.Sc. degree in computer engineering from the University of Prishtina, Kosova, and the Ph.D. degree in computer science from TU Wien, in 2022. He is currently an Assistant Professor with the University of Prishtina and an external Senior Researcher with the Distributed Systems Group. Previously, he was a Postdoctoral Researcher and a University Assistant with TU Wien. He has authored over 40 peer-reviewed publications in international journals and conferences and has was a TPC member for several international events. His current research interests include the Internet of Things, edge computing, EdgeAI, and privacy in distributed, self-adaptive, and cyber-physical systems.

**XHEVAHIR BAJRAMI** received the B.Sc. and M.Sc. degrees from the University of Prishtina, and the Ph.D. degree in mechatronics from TU Wien, Austria, where he conducted his research at the Institute of Handling Devices and Robotics. Since 2020, he has been the Vice Dean of finance and infrastructure with the Faculty of Mechanical Engineering. He is currently an Associate Professor of Mechatronics with the University of Prishtina. His research interests include advanced humanoid robotics, intelligent and predictive control of autonomous mechatronic systems, the IoT, and deep reinforcement learning for adaptive locomotion control.

**REZA FARAHANI** received the M.Sc. degree in computer science from the University of Tehran, Iran, and the Ph.D. degree in computer science from the University of Klagenfurt. From October 2019 to February 2023, he was a Project Assistant with the Christian Doppler Laboratory ATHENA. Since March 2023, he has been actively involved in the Horizon Europe Graph-Massivizer Project, where he leads WP5, focusing on the design and development of an open-source orchestration tool and a large-scale edge–cloud continuum testbed. He is currently a Postdoctoral Researcher and a Lecturer with the Institute of Information Technology (ITEC), University of Klagenfurt, Austria. Recently, he coordinates the FFG-funded EdgeAI-Drone project, overseeing the technical design and integration of edge-based AI analytics for drone-assisted bridge infrastructure inspection. From November 2022 and January 2023, he was a Visiting Scholar with the Institute for Communication Systems (ICS), University of Surrey, U.K. He has published over 40 peer-reviewed publications in international conferences and journals, has served as a TPC member for several international conferences, and received an IEEE best paper award. His research interests include distributed and networked systems, modern multimedia systems, edge–cloud continuum, edge AI, and serverless computing. Further information is available at https://www.linkedin.com/in/reza-farahani/



**PRAVEEN KUMAR DONTA** (Senior Member, IEEE) received the bachelor's and master's degrees (Hons.) in technology from the Department of Computer Science and Engineering, JNTUA, Ananthapur, in 2012 and 2014, respectively, and the Ph.D. degree from the Department of Computer Science and Engineering, Indian Institute of Technology (Indian School of Mines), Dhanbad, in June 2021. He was a Visiting Ph.D. Fellow at the Mobile and Cloud Laboratory, University of Tartu, Estonia, from July 2019 to January 2020. He was with the Distributed Systems Group, TU Wien, as a Postdoctoral Researcher, from July 2021 to June 2024. He is currently an Associate Professor (Docent) with the Department of Computer and Systems Sciences, Stockholm University, Sweden. His current research interests include learning-driven distributed computing continuum systems and intelligent data protocols. He is an ACM Professional Member. He is serving as Editorial board member in IEEE Internet of Things Journal, *Computing* (Springer), ETT Wiley, *Measurement*, and *Computer Communications* Journals (Elsevier).

**SCHAHRAM DUSTDAR** (Fellow, IEEE) is currently a Full Professor of computer science heading the Research Division of Distributed Systems with the TU Wien, Austria. He holds several honorary positions: a Honorary Professor at Nanjing University of Science and Technology, China, since 2023, and a Francqui Chair Professor with the University of Namur, Belgium, from 2021 to 2022, the University of California (USC), Los Angeles; Monash University, Melbourne, Shanghai University, Macquarie University, Sydney, University Pompeu Fabra, Barcelona, Spain. From December 2016 to January 2017, he was a Visiting Professor with the University of Sevilla, Spain, and from January to June 2017, he was a Visiting Professor with UC Berkeley, USA.