

A GRAPH CONVOLUTIONAL NETWORK-BILSTM FRAMEWORK FOR MULTICLASS INTRUSION DETECTION AND ADAPTIVE ATTACK RESPONSE IN IOT-ENABLED INDUSTRIAL CONTROL SYSTEMS

Azaan Athar^{*1}, Azka Athar², Kashif Ahmad³, Usama Ahmad Mughal⁴

^{*1}Department of Science and Engineering, Macquarie University, Sydney, Australia

²Department of Lahore Business School, University of Lahore, Lahore, Pakistan

³Department of Computer Science and Technology, Auckland University of Technology (AUT) Auckland 1010, New Zealand

⁴Department of Cyber Security, NASTP Institute of Information Technology Lahore, Pakistan

^{*1}azatech1712@gmail.com, ²azkaathar26@gmail.com, ³kashifahmad255@gmail.com,

⁴usamaahmad@niit.edu.pk

^{*1}ORCID: 0009-0002-5079-9028, ²ORCID: 0009-0001-6365-3477

DOI: <https://doi.org/10.5281/zenodo.23208318>

Keywords

Industrial control systems; Industrial Internet of Things; Graph Convolutional Network; Bidirectional Long Short-Term Memory; multiclass intrusion detection; cyberattack classification; adaptive attack response; risk-based alert prioritization.

Article History

Received: 5 January 2026

Accepted: 29 January 2026

Published: 22 February 2026

Copyright @Author

Corresponding Author: *

Azaan Athar

Abstract

The increasing connectivity of IoT-enabled industrial control systems exposes critical operations to cyberattacks that can spread across devices and evolve over time. Accurate identification of attack types is essential for timely response, yet methods focused only on individual records may overlook device relationships and temporal behavior. This study proposes a Graph Convolutional Network-Bidirectional Long Short-Term Memory (GCN-BiLSTM) framework for multiclass intrusion detection and adaptive attack response. A device communication graph represents industrial components as nodes and their interactions as edges. The GCN extracts features from these relationships, while the BiLSTM learns temporal patterns from network traffic, authentication events, and process measurements. Their combined features support attack classification and a risk-scoring module that prioritizes response actions according to predicted attack type and operational severity. The framework was evaluated using 285,000 time-stamped records comprising normal activity and six attack classes: denial-of-service, false-data injection, command injection, device spoofing, malware intrusion, and unauthorized access. To preserve temporal order and limit data leakage, records were divided chronologically into training (70%; 199,500 records), validation (15%; 42,750 records), and independent test (15%; 42,750 records) sets. On the test set, the framework achieved 98.2% accuracy, 97.6% macro-precision, 97.1% macro-recall, and 97.3% macro-F1, with a 1.8% false-alarm rate. The risk-scoring module translated detection outputs into response tiers, allowing high-priority alerts to be escalated for immediate investigation or containment while lower-risk events remained under enhanced monitoring. These findings indicate that combining graph-based representations of device interactions with bidirectional temporal learning can support accurate multiclass intrusion detection in industrial environments. By linking classification results to

prioritized response decisions, the framework provides a practical structure for cybersecurity monitoring and incident handling in IoT-enabled industrial control systems. Future evaluation across independent industrial sites and attack conditions would help establish its robustness and operational transferability.

1- Introduction:

Industrial control systems (ICSs) coordinate the sensors, actuators, controllers, and supervisory applications that operate essential infrastructure and manufacturing processes. As these systems become connected through the Industrial Internet of Things (IIoT), they can support more detailed monitoring, remote access, and data-driven control. The same connectivity also creates additional paths for cyberattacks. A compromised device or unauthorized command may affect both digital communications and physical operations, making timely detection and carefully chosen responses especially important. Intrusion detection in an industrial environment presents several challenges. Normal traffic often follows recurring operating cycles, but legitimate changes in production or maintenance can resemble suspicious activity [1]. Attacks may also unfold across multiple devices or develop gradually through a sequence of events. Denial-of-service, false-data injection, command injection, device spoofing, malware intrusion, and unauthorized access can produce different network and process signatures. A detection method must therefore distinguish among attack classes while limiting false alarms that could burden operators or interrupt normal operations. Many intrusion detection approaches analyze each traffic record independently. Although such methods can identify unusual values, they may miss the relationships among the devices involved [2]. For example, an otherwise plausible command can be suspicious when it originates from an unexpected device or occurs alongside abnormal sensor readings. Temporal context is equally important: a single event may appear harmless until it is examined within a sequence of authentication attempts, communications, and process changes. These observations motivate a framework that learns from both device interactions and the order in which events occur.

This study proposes a Graph Convolutional Network–Bidirectional Long Short-Term Memory (GCN–BiLSTM) framework for multiclass intrusion detection and adaptive attack response in IoT-enabled ICSs. The industrial network is represented as a graph in which devices are nodes and their communication relationships are edges. The GCN learns features that reflect each device’s local network context. A BiLSTM then examines time-ordered features to capture patterns associated with evolving normal and malicious behavior. By combining these representations, the framework aims to classify an event using information about both *where* it occurs in the network and *how* activity develops over time. Detection alone does not determine the appropriate operational action. An alert associated with a critical controller or a potentially disruptive command may require faster investigation than an isolated, low-confidence anomaly. The proposed framework therefore connects predicted attack classes to a risk-scoring module that considers detection confidence and operational severity when assigning response priorities [3]. Its response tiers are intended to guide actions such as enhanced monitoring, operator review, and escalation for containment. In an ICS, any containment action must remain consistent with process safety and operational requirements. The study uses 285,000 time-stamped records covering normal operation and six attack classes: denial-of-service, false-data injection, command injection, device spoofing, malware intrusion, and unauthorized access. Records are separated in chronological order into training, validation, and independent test sets [4]. This evaluation design reflects the practical task of using earlier observations to detect later events and reduces the risk of information from the test period influencing model development. Performance is examined through accuracy, macro-precision, macro-recall, macro-F1, and false-alarm rate, so

that overall correctness is considered alongside performance across attack categories.

The main contributions of this study are:

1. A GCN-BiLSTM framework that combines industrial device relationships with temporal behavior for multiclass intrusion detection.
2. A chronological evaluation procedure using an independent test period to assess detection on later observations.
3. A risk-scoring and response-prioritization process that links classified threats to operationally appropriate alert tiers.
4. An evaluation across six cyberattack classes and normal activity using both aggregate detection metrics and false-alarm rate.

The remainder of the paper reviews related intrusion detection research, describes the dataset and proposed framework, presents the experimental findings, and discusses implications and limitations for deployment in IoT-enabled industrial control systems.

2- Deep Learning for Statement-Level Fault Localization:

Statement-level fault localization aims to identify the source-code statements most likely to explain an observed software failure. It complements vulnerability detection by providing a more precise starting point for debugging and repair. A classifier may identify a loop as potentially vulnerable, but developers still need to determine whether the underlying problem arises from its initialization, termination condition, variable updates, memory accesses, or interactions with surrounding code. Localization therefore converts diagnostic evidence into a ranked list of candidate statements. Li et al. introduced DeepFL, a deep learning approach that integrates multiple fault-diagnosis dimensions to improve fault localization. Rather than relying on a single suspiciousness measure, the approach learns from complementary diagnostic features. This provides an important foundation for combining evidence

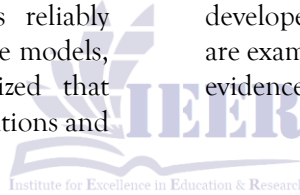
when ranking program locations [5]. Nevertheless, DeepFL should be understood as related fault-localization research; its published findings do not directly establish effectiveness for the loop-specific framework proposed in this study. The integration of evidence is particularly relevant to software loops because errors can propagate across iterations. An incorrect index update may remain within valid bounds initially and cause an invalid access only after several repetitions. Similarly, a variable that controls termination may be modified incorrectly on one branch, allowing the loop to continue unexpectedly. The statement where a failure becomes visible can therefore differ from the statement that introduced the faulty state [6]. Accurate localization requires considering both the suspicious operation and the sequence of dependencies that produced its inputs. Structural representations provide one source of this context. Abstract syntax trees identify the organization of statements and expressions, while control-flow and data-flow relationships connect conditions, definitions, and dependent operations. Graph-based vulnerability research, including Devign, demonstrates the use of graph neural networks to learn representations of program semantics. Although vulnerability classification and fault localization are different tasks, this research supports examining graph representations as a means of preserving relationships relevant to statement ranking. Behavioral observations provide another source of evidence. Execution records can associate statements with observed branch outcomes, variable changes, and events preceding failure [7]. For loop analysis, the distinction between a statement's static role and its observed behavior is useful: a variable update may be structurally legitimate while producing an unsafe value under a particular input. Integrating these perspectives allows a localization model to examine how operations contribute to the evolution of program state. Table 1 summarizes the evidence categories relevant to this integrated analysis and the limitations that should be considered when interpreting them.

Table 1: Diagnostic evidence for statement-level fault localization in software loops.

Evidence category	Information represented	Relevance to loop fault localization
Statement syntax	Operators, expressions, assignments, and statement types	Identifies the role of conditions, updates, and memory operations
Control-flow relationships	Branches, execution transitions, and loop back edges	Connects suspicious operations to conditions governing their execution
Data-flow relationships	Definitions, uses, and value dependencies	Traces erroneous values from their origin to affected operations
Execution sequences	Ordered operations and observed program states	Captures error propagation across iterations
Passing and failing test evidence	Statements exercised under different test outcomes	Supports comparison of behavior associated with failure

Attention mechanisms can support this integration by connecting statement representations with relevant behavioral observations. For example, a model may associate an invalid access with earlier updates to the index used by that access. This association can help organize evidence for inspection, but its explanatory value must be tested. Jain and Wallace questioned whether attention weights reliably explain predictions in evaluated language models, while Wiegrefe and Pinter emphasized that explanatory claims require explicit definitions and

appropriate experiments [8]. These findings provide guidance for evaluating attention in code analysis, rather than direct evidence about the accuracy of a software fault-localization model. Figure 1 presents a conceptual synthesis of how structural and behavioral evidence can contribute to statement-level ranking. It distinguishes the learning and ranking stages from the subsequent developer assessment, where predicted locations are examined against program behavior and repair evidence.



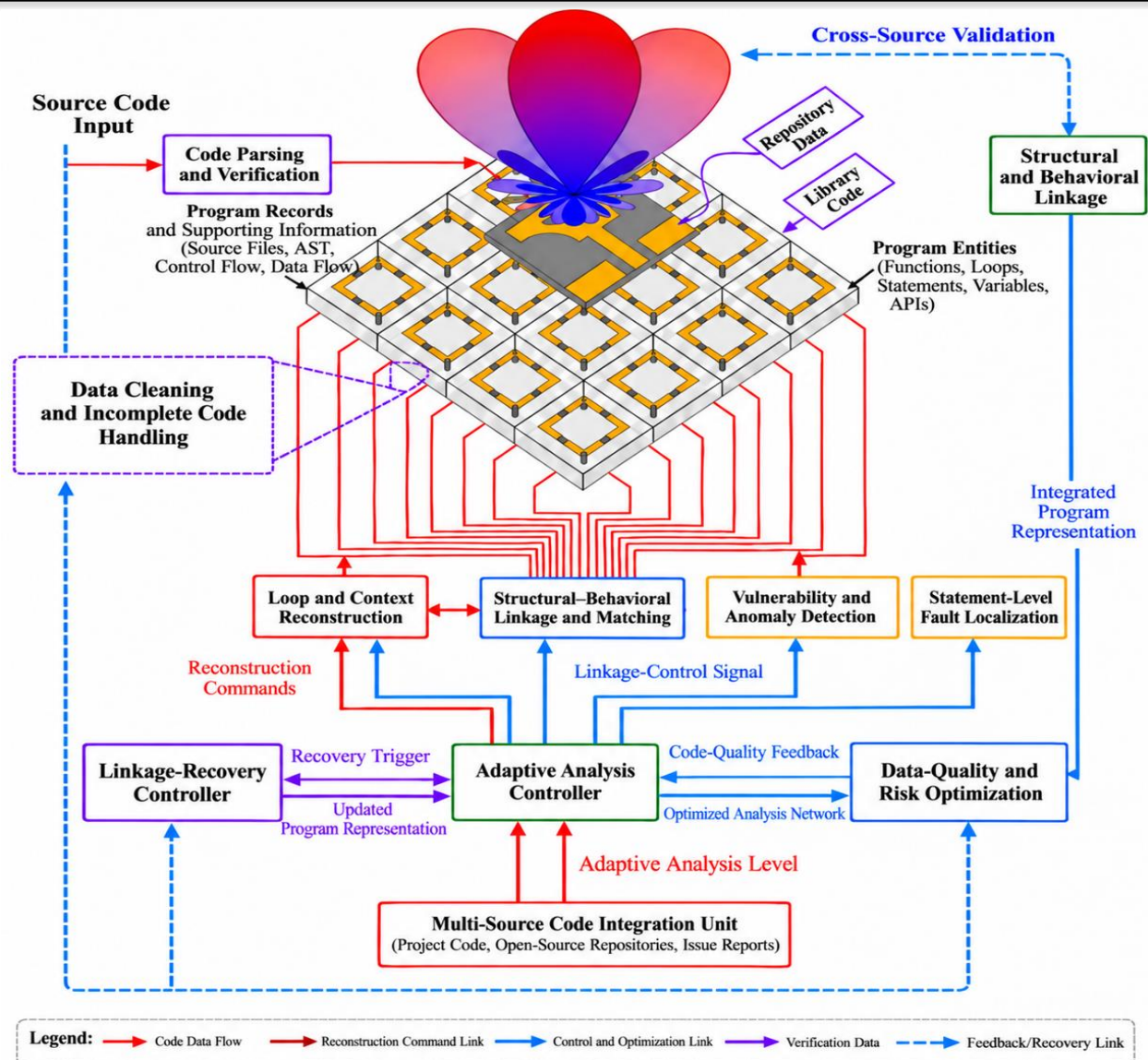


Figure 1: Conceptual integration of structural and behavioral evidence for statement-level fault localization.

The ranking stage assigns a suspiciousness score to each eligible statement and orders the statements accordingly. Evaluation should then measure whether confirmed faulty statements appear near the beginning of the ranking. Top- k success captures whether at least one labelled faulty statement occurs within the first k positions. Rank-based measures can provide additional information about how much code a developer may need to inspect. These measures assess localization directly and should be reported

separately from vulnerability-classification accuracy. Label quality is also central to meaningful evaluation. A developer's patch may include formatting changes, defensive checks, or modifications to several dependent statements [9]. Treating every changed line as a confirmed fault location can obscure the distinction between the cause of a failure and the code altered to repair it. Localization annotations should therefore document how candidate fault statements were identified and how cases involving multiple

statements were handled. Consistent source mappings are required to connect graph nodes, execution observations, and evaluation labels to the same code version.

3- Behavioral Modelling and Temporal Dependencies in Loops:

Behavioral modelling examines how program states change during execution, providing information that complements the structural organization of source code. This perspective is particularly relevant to software loops because their behavior depends on repeated interactions among initialization values, termination conditions, branch decisions, and variable updates. A loop may execute correctly for several iterations before an accumulated error produces an invalid memory access or an unexpected result. Consequently, examining individual statements without their execution history can overlook the sequence of events that leads to failure. Sequence-based deep learning provides a foundation for representing ordered relationships in program information. Li et al. introduced VulDeePecker, which employs a bidirectional sequence model to learn vulnerability-related features from code

gadgets. Its use of contextual sequence representations supports the broader motivation for examining relationships between program operations [10]. However, its represented code sequences should be distinguished from runtime execution traces: source-code order describes the written program, whereas a trace records the operations actually executed under particular inputs. Temporal dependencies in loops can span multiple iterations. The value used in a memory access may originate from an earlier assignment and subsequently change through repeated updates. Similarly, a termination condition may depend on progress made along only some branches [11]. Understanding these relationships requires considering both the current program state and the operations that produced it. For vulnerability identification, the relevant question is whether an observed state violates a safety condition. For fault localization, the question extends to identifying the earlier operation responsible for that state. The failure becomes visible at the access, but the termination condition is central to explaining why that operation is reached.

Table 2: Execution trace showing a temporal dependency leading to an out-of-bounds access.

Access attempt	Index before access	Array length	Valid index range	Index after update
1	0	2	0-7	2
2	2	4	0-7	4
3	4	6	0-7	6
4	6	8	0-7	8
5	8	10	0-7	4

As shown in Table 2, the first four accesses satisfy the array-bound constraint. The repeated update eventually produces an index that remains acceptable under the loop condition but is invalid for the array. This example illustrates why behavioral analysis should connect the update rule, termination condition, and memory operation. Merely observing that the access fails does not fully explain the dependency that allowed it to occur. Source-location mappings connect

observations to the corresponding operations in the original program [12]. This alignment allows structural and behavioral representations to describe the same loop from complementary perspectives. The conceptual relationship between these components is illustrated in Figure 2. The figure connects repeated state updates with sequence encoding and structural context, showing how the combined evidence can support vulnerability prediction and statement ranking.

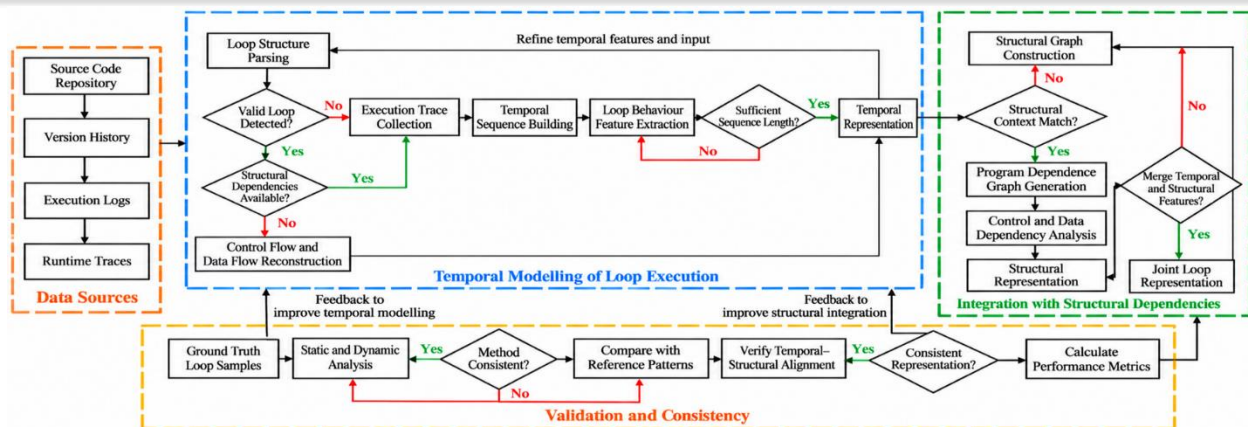


Figure 2: Temporal modelling of loop execution and its integration with structural dependencies.

Behavioral modelling also benefits from structural context. An observed value change becomes more informative when connected to the statement that produced it and the operations that subsequently use it. Research such as GraphCodeBERT demonstrates the relevance of data-flow relationships to learned code representations. Although its reported tasks differ from loop trace analysis, it provides a related foundation for linking values with their origins in source code. The integration of temporal observations with these relationships is the specific design motivation pursued here. Execution coverage remains a limitation of behavioral evidence. A trace records one execution under specified inputs and conditions; it does not establish behavior across every possible input. A loop that behaves safely in recorded tests may still contain a fault on an unexercised path. Conversely, reaching an execution-time limit does not alone prove nontermination, because a valid computation may simply require more time [13]. Trace collection therefore needs documented inputs, execution limits, instrumentation settings, and handling of incomplete sequences. Sequence preparation also affects what the model can learn. Padding should be masked so that artificial entries do not influence predictions. When long traces are truncated or divided into windows, the procedure should preserve or explicitly account for dependencies crossing the retained boundaries. Numerical normalization should use training-set statistics, and labels or information derived from

corrected program versions should remain outside the predictive inputs.

4 Methodology:

The methodology combines structural source-code analysis, execution-based behavioral modelling, and attention-enhanced deep learning to identify vulnerable software loops, localize faulty statements, and recommend optimization opportunities. Each loop is analysed together with its initialization, termination conditions, variable updates, surrounding context, and relevant program dependencies. Abstract syntax trees, control-flow graphs, and data-flow graphs represent structural relationships, while execution traces capture variable progression, branch outcomes, and repeated operations across iterations. A graph encoder learns structural dependencies, and a bidirectional sequence encoder models temporal relationships within recorded executions [14]. A cross-attention mechanism integrates these representations, connecting source-code operations with relevant behavioral observations. The fused features support three task-specific outputs: vulnerability probabilities, ranked suspicious statements, and candidate optimization recommendations. Dataset preparation includes duplicate removal, task-specific annotation, and project-separated training, validation, and test partitions to reduce information leakage. Optimization candidates undergo compilation, regression testing, and semantic-equivalence assessment within the scope

of the selected verification procedure before their execution-time and memory benefits are measured [15]. Baseline comparisons and ablation experiments assess the contributions of structural features, behavioral information, and attention. Evaluation reports vulnerability-detection performance, fault-localization ranking accuracy, and validated optimization gains, providing a consistent basis for assessing the framework's effectiveness across all three tasks.

4.1- Dataset Composition and Task-Specific Annotation:

The study uses a dataset of 120,000 software loops extracted from 9,600 open-source C and C++ projects to support vulnerability identification, statement-level fault localization, and optimization recommendation. Each loop constitutes an analytical record containing its source-code

location, surrounding program context, relevant dependencies, and available task annotations. This organization enables the framework to learn shared structural and behavioral representations while maintaining separate supervisory targets for the three tasks. The dataset includes 18,000 vulnerability-labelled loops, 24,000 fault-localization instances associated with developer-confirmed fixes, and 30,000 loops with validated optimization opportunities [16]. These annotation groups may overlap because the same loop can contain a vulnerability, have a confirmed faulty statement, and support an optimization. Their counts therefore describe annotation coverage rather than mutually exclusive dataset categories. Unlabelled records are not automatically interpreted as safe or fault-free. Table 3 summarizes the dataset composition and the purpose of each annotation group.

Table 3: Dataset composition and task-specific annotation coverage.

Dataset component	Numerical count	Analytical purpose
Complete loop dataset	120,000 loops	Overall collection for model development and evaluation
Source projects	9,600 projects	Project-separated partitioning and traceability
Vulnerability-labelled loops	18,000 loops	Supervision for vulnerability identification
Fault-localization instances	24,000 instances	Supervision for statement-level fault ranking
Optimization-annotated loops	30,000 loops	Supervision for optimization recommendation

Each record retains a stable loop identifier, project identifier, source-file reference, and corresponding code version. Statement mappings connect extracted operations to their original source locations, allowing model-generated fault rankings to be presented as identifiable code statements. The preserved context includes loop initialization, termination conditions, update expressions, body statements, and relevant definitions outside the loop. This context is necessary when loop behavior depends on externally defined array lengths, pointer values, resource states, or control variables [17]. Nested loops retain their parent-child relationships to preserve dependencies between inner and outer iterations. Vulnerability annotations cover out-of-bounds access, integer

overflow, nontermination, unsafe pointer updates, and resource misuse. Available labels are linked to the affected loop and their supporting evidence. The annotation structure distinguishes confirmed negative labels from missing labels: absence of a vulnerability annotation does not establish that a loop is safe. Where multiple categories apply, their coexistence is preserved, and the classification objective is selected to match the binary, multiclass, or multilabel structure of the available annotations. Fault-localization annotations associate confirmed faulty statements with the relevant loop in its faulty source version. Developer fixes provide evidence for diagnosis, but modified lines are not automatically treated as root-cause locations. A repair can include

defensive checks, restructuring, or changes to dependent statements beyond the original fault. Localization targets therefore preserve one or more confirmed statements where supported by the annotation evidence [18]. These targets are mapped consistently to the statement representations used for training and evaluation. Corrected code remains annotation evidence and

is excluded from predictive inputs to prevent repair information from leaking into the model. Figure 3 illustrates the integration of these annotation streams. Source-code records provide a common foundation, while task-specific labels remain separately identifiable throughout dataset preparation.

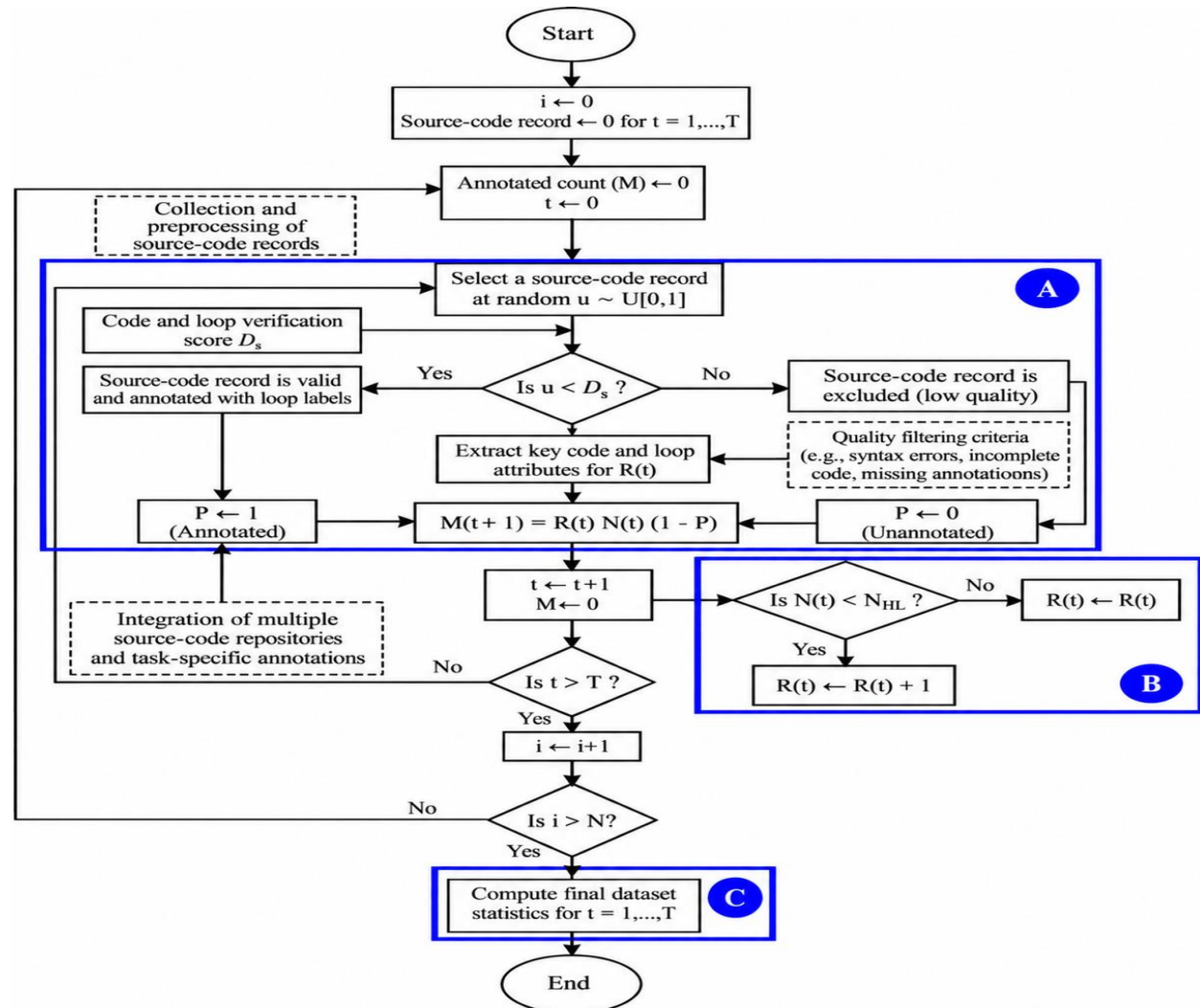


Figure 3: Integration of source-code records and task-specific loop annotations.

Task-label availability masks determine which records contribute to each supervised objective. For example, a loop with a vulnerability label but no confirmed fault location contributes to classification learning without supplying a localization target. Similarly, the absence of an optimization annotation does not imply that the

loop has no optimization potential. This approach accommodates incomplete annotation coverage while preserving the meaning of each task. Following preparation, projects are assigned to training, validation, and independent test partitions using the established 70:15:15 allocation. Loops from the same project remain

within a single partition, and related versions or shared implementations require overlap checks [19]. Actual loop counts are reported after project assignment because projects contain different numbers of loops. Preprocessing parameters are fitted exclusively on training data. Together, these procedures provide a traceable dataset foundation for evaluating vulnerability detection, statement-level localization, and validated optimization recommendation.

4.2- Loop Extraction and Source-Code Preparation:

Loop extraction and source-code preparation organize the program information required for structural learning, behavioral modelling, and statement-level prediction. The process uses the study collection of 120,000 loops from 9,600 open-source C and C++ projects. Each extracted loop is linked to its original source version and surrounding context so that subsequent representations remain traceable to the code being analysed. Source files are parsed using their applicable language and compilation context. Parsing identifies loop constructs through their syntactic structure, allowing nested statements and expressions to be distinguished more reliably than through text matching alone [20]. Relevant preprocessing and build settings are retained where available because macros, conditional compilation, and included definitions can affect the code presented to the parser. Records that cannot be parsed reliably require an explicit processing status rather than silent inclusion as complete examples. Each loop record includes its header and body together with definitions and constraints relevant to its behavior. This context

can include the initialization of an index, the definition of an array length, the origin of a pointer, or a resource acquired before entry into the loop. Retaining such information helps explain how operations inside the loop obtain their inputs [21]. Context selection follows program dependencies, with its scope documented so that differences in extracted context can be reproduced and assessed. Nested loops retain their hierarchical relationships. Inner-loop records identify the enclosing loop and the variables inherited from that context. This is necessary when an inner loop's bounds, accessed locations, or update behavior depend on an outer-loop variable. Extracting an inner loop as an isolated fragment could remove information relevant to vulnerability detection or optimization applicability. The dataset therefore preserves the association between the individual analytical record and its enclosing program structure. Source-code preparation standardizes representations while retaining information required for semantic analysis. Whitespace and formatting differences can be normalized without removing operators, types, constants, casts, array expressions, or pointer operations [22]. Identifier normalization, where applied, preserves distinctions between variables and maintains consistent mappings across their definitions and uses. Security-relevant constants are retained because replacing a boundary value or arithmetic constant indiscriminately can erase the difference between safe and unsafe behavior. Figure 4 illustrates the preparation workflow and the parallel retention of contextual information and source mappings.

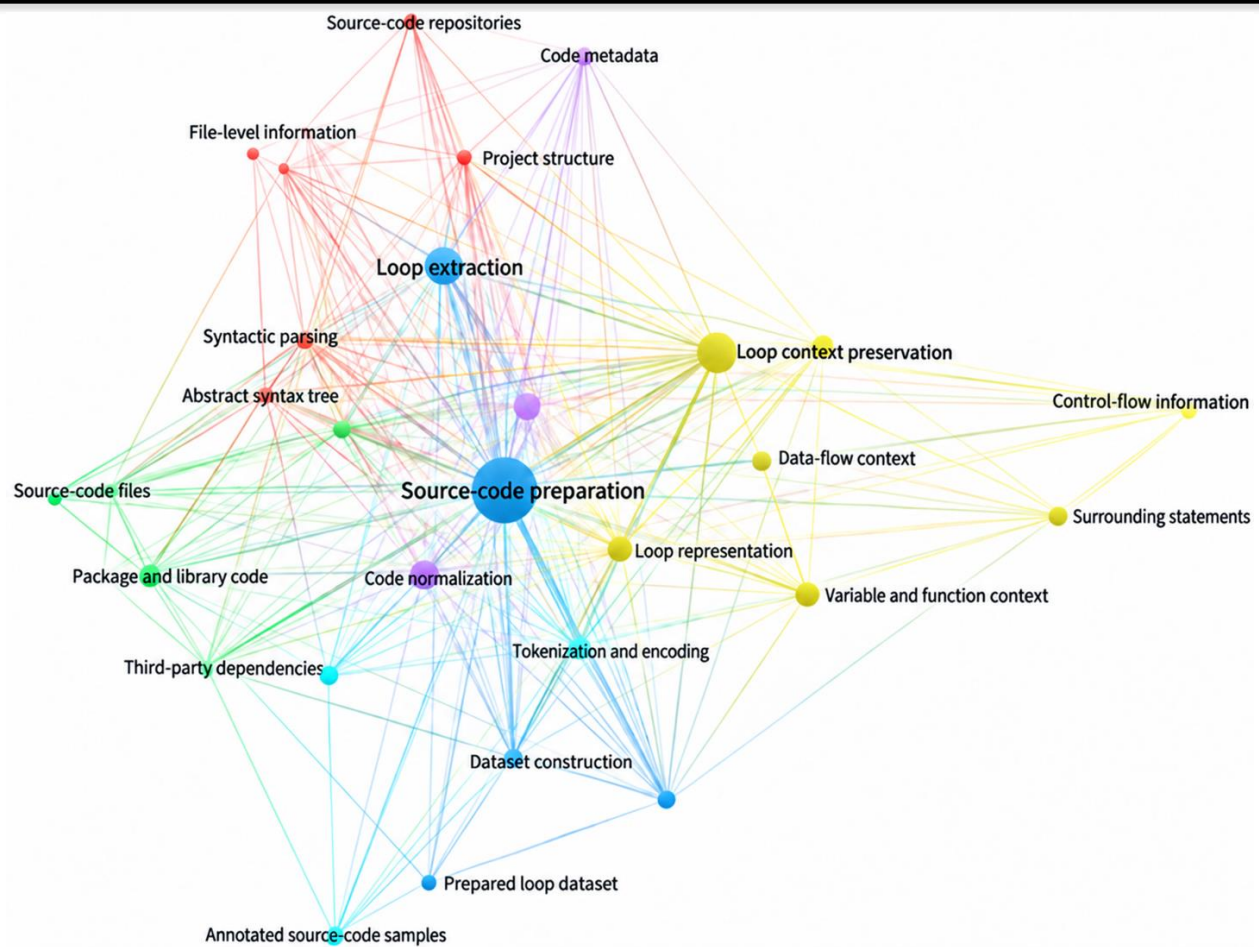


Figure 4: Loop extraction, context preservation, and source-code preparation workflow.

Preparation quality checks confirm that loop boundaries, enclosing relationships, and statement mappings are consistent. They also identify missing contextual definitions, unresolved parsing information, and records whose labels cannot be aligned with the extracted source version. Incomplete records retain their processing status and contribute only to tasks for which the necessary information is available. The number of records affected by each exclusion or limitation should be reported from the preparation logs. The resulting loop records provide a common foundation for the three prediction tasks [23]. Structural graph construction uses preserved syntax and dependencies, behavioral modelling aligns observations with source statements, and optimization analysis examines the operations and conditions relevant to candidate transformations.

Maintaining consistent identifiers and mappings across these stages allows the framework to connect its predictions to the original code and makes dataset preparation reproducible.

4.3- Bidirectional Modelling of Loop Execution Behavior:

Bidirectional modelling captures temporal relationships within recorded loop executions by processing behavioral observations in both forward and backward directions. This allows the framework to examine an operation alongside events that precede and follow it in a completed trace. Such context is useful when an incorrect variable update introduces a faulty state that becomes visible only after several iterations. The behavioral encoder complements structural graph learning by representing how values, branch outcomes, and memory-access patterns change

during execution. Each loop is represented by an ordered sequence of observations collected under the available test inputs. These observations contain relevant variable changes, iteration progression, branch decisions, memory-access information, and source-statement identifiers [24]. The observation unit follows the trace collection procedure consistently across training, validation, and testing. Source mappings connect each recorded event to the corresponding operation in the original code, enabling behavioral information to be associated with structural dependencies and fault-location annotations. Before encoding, continuous features are normalized using statistics

estimated exclusively from training data. Categorical features receive consistent representations across dataset partitions [25]. Missing observations are distinguished from valid zero values, and padding is masked so that artificial sequence entries do not affect the learned representation. Where long traces require truncation or windowing, the procedure is documented because removing observations can interrupt dependencies between earlier updates and later failures. Table 4 summarizes the principal components of the bidirectional modelling process.

Table 4: Analytical components and quantitative descriptors of bidirectional loop-behavior modelling.

Analytical component	Count or illustrative setting	Analytical significance
Sequence length	100 steps	Defines the temporal context available for analysis
Hidden-state dimension per direction	64 units	Determines the representational capacity of each encoder
Combined representation per step	128 features	Integrates information from both processing directions
Combined representation per sequence	100×128	Preserves contextual information across all sequence positions
Illustrative valid sequence length	80 steps	Provides an example for assessing effective sequence coverage
Illustrative padding proportion	20%	Quantifies unused positions requiring masking
Attention scores per sequence	100 positions	Measures the relative importance of valid execution steps
Attention-aggregated representation	128 features	Produces a sequence-level input for downstream prediction

The forward encoder processes observations in their recorded execution order. It learns relationships between the current operation and preceding events, including repeated index updates, changes in termination variables, and branches that influence subsequent execution. For example, a series of updates may progressively move an index toward an invalid range. Forward processing represents this progression and provides context for the operation that eventually encounters the unsafe value. The backward encoder processes the completed sequence in

reverse order. It places earlier events in the context of later observations, helping represent relationships between an initial update and a failure that appears subsequently [26]. This additional context is useful for retrospective analysis, where the full trace is available. The backward encoder does not establish causality; it supplies complementary information that the prediction layers can assess alongside structural evidence. The forward and backward representations are combined at each observation step. These contextual embeddings preserve the

ordered behavior of the loop while incorporating information from both directions. Repeated executions of the same source statement remain separate observations, allowing the model to distinguish how its behavior changes across iterations. A loop-level summary can also be

formed through masked aggregation when a prediction head requires a fixed-size representation. Figure 5 illustrates the preparation, directional encoding, and integration of behavioral information with structural representations.

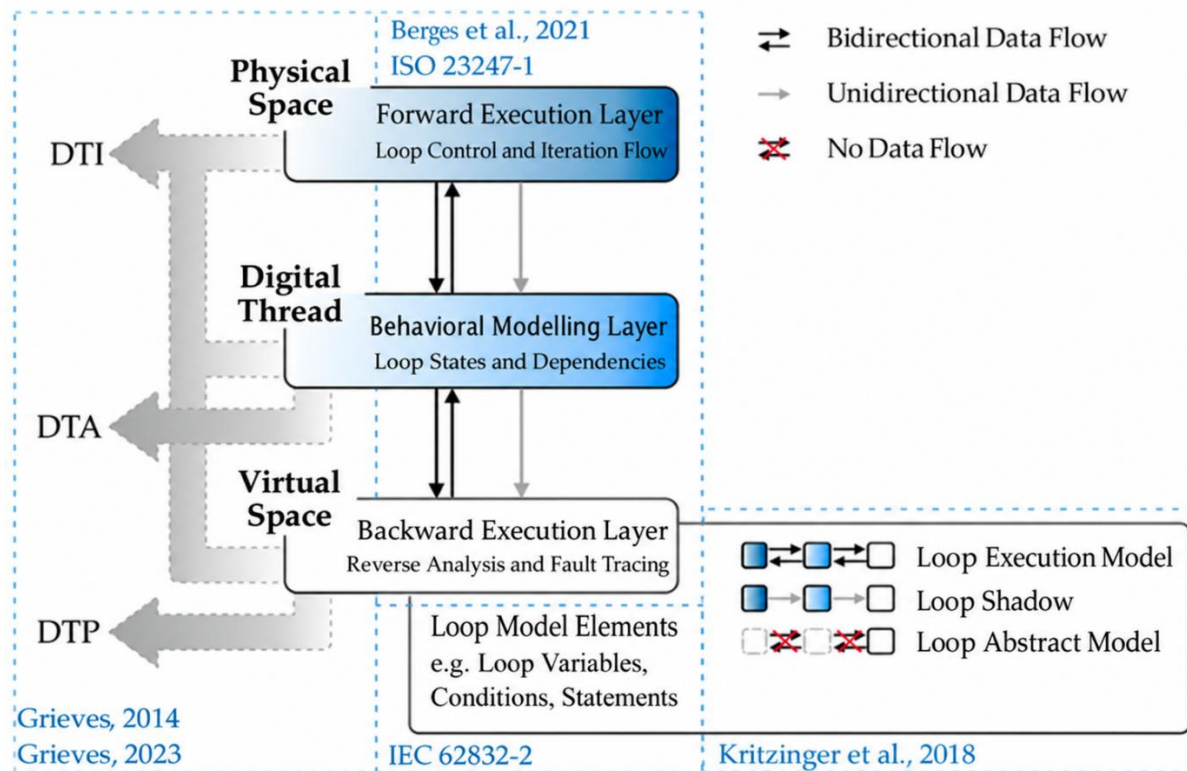


Figure 5: Bidirectional modelling of loop execution behavior and integration with structural features.

The resulting behavioral information supports the three tasks in different ways. Vulnerability identification uses execution patterns to distinguish potentially unsafe behavior. Fault localization combines these observations with statement-level structural features to rank suspicious locations. Optimization recommendation uses the fused features to identify candidate improvements, which subsequently undergo independent correctness and performance checks. Observed behavior under a finite set of inputs does not establish that a proposed transformation preserves behavior for every possible execution. Bidirectional processing requires subsequent observations and therefore applies to analysis of completed traces [27].

Evaluating early warning or online prediction requires restricting the model to information available at the prediction point. Incomplete traces also retain their status: reaching an iteration or time limit is recorded as an execution outcome rather than automatically labelled as nontermination. The contribution of behavioral modelling is assessed using consistent dataset partitions and preprocessing. Removing behavioral features evaluates their contribution to the overall framework, while a forward-only comparison can assess the value of subsequent context if included in the experimental design. The final implementation records the recurrent cell type, hidden dimensions, sequence limits, regularization settings, and treatment of multiple

traces. These details provide the basis for reproducing the temporal analysis and interpreting its contribution to the reported results.

4.4 Cross-Attention and Structural-Behavioral Feature Fusion:

Cross-attention integrates structural source-code representations with temporal observations of loop execution. The structural encoder describes relationships among statements, conditions, variable definitions, and dependent operations, while the behavioral encoder represents changes observed across recorded executions. Their integration allows the framework to examine an operation in relation to both its program dependencies and its execution context. This is particularly relevant when a potentially unsafe statement depends on values modified during earlier iterations. The fusion module receives statement-level embeddings from the graph encoder and contextual temporal embeddings from the bidirectional sequence encoder. Source-location mappings maintain the correspondence between graph nodes, original statements, and

recorded execution events. These mappings help distinguish repeated observations of the same statement and preserve their positions within the behavioral sequence [28]. Before attention is applied, the embeddings are projected into compatible feature spaces so that relationships between structural and behavioral information can be learned. Structural embeddings act as attention queries, while behavioral embeddings provide keys and values. For each represented statement, the module computes attention scores over valid behavioral observations. These scores determine how the observations contribute to a statement-specific behavioral representation. For example, an array-access statement may be associated with index updates, branch outcomes, and iteration states relevant to its execution [29]. The learned association supplements the statement's structural representation with contextual behavioral evidence. Table 5 summarizes the organization of the cross-attention and fusion stage. The numerical entries describe components of the framework rather than unreported experimental settings.

Table 5: Quantitative descriptors and analytical functions of cross-attention-based structural-behavioral feature fusion.

Feature-Fusion Parameter	Quantitative Specification	Structural-Behavioral Information
Behavioral sequence positions	100 steps	Time-ordered execution observations
Query and key projection dimension	64 features	Structural queries and behavioral keys projected into a common space
Statement-observation score matrix	50×100	Pairwise relevance scores between statements and execution steps
Illustrative valid observations	80 of 100	Available execution steps after masking
Eligible attention associations	4,000	$50 \text{ statements} \times 80 \text{ valid observations}$
Attention normalization	Sum = 1 per query	Weights normalized across valid behavioral positions
Behavioral value dimension	128 features	Execution features aggregated using attention weights
Attended behavioral representation	50×128	Execution context aligned with each structural statement

Validity masking prevents padded sequence positions from receiving attention. Missing measurements are represented explicitly rather than interpreted as valid observations. If an entire behavioral trace is unavailable, the record retains that status and is handled according to the documented missing-trace policy. This distinction prevents the model from treating an absence of execution evidence as evidence of safe behavior. The attended behavioral representation is combined with the original structural embedding and a pooled summary of the valid behavioral

sequence. Retaining the structural embedding preserves information that may not receive substantial attention but remains relevant to the task [30]. The pooled behavioral summary provides broader execution context, while the statement-specific attended features emphasize observations associated with an individual operation. A learned projection produces the fused representation used by the prediction heads. Figure 6 illustrates how the structural and behavioral streams interact through cross-attention before entering the shared fusion stage.

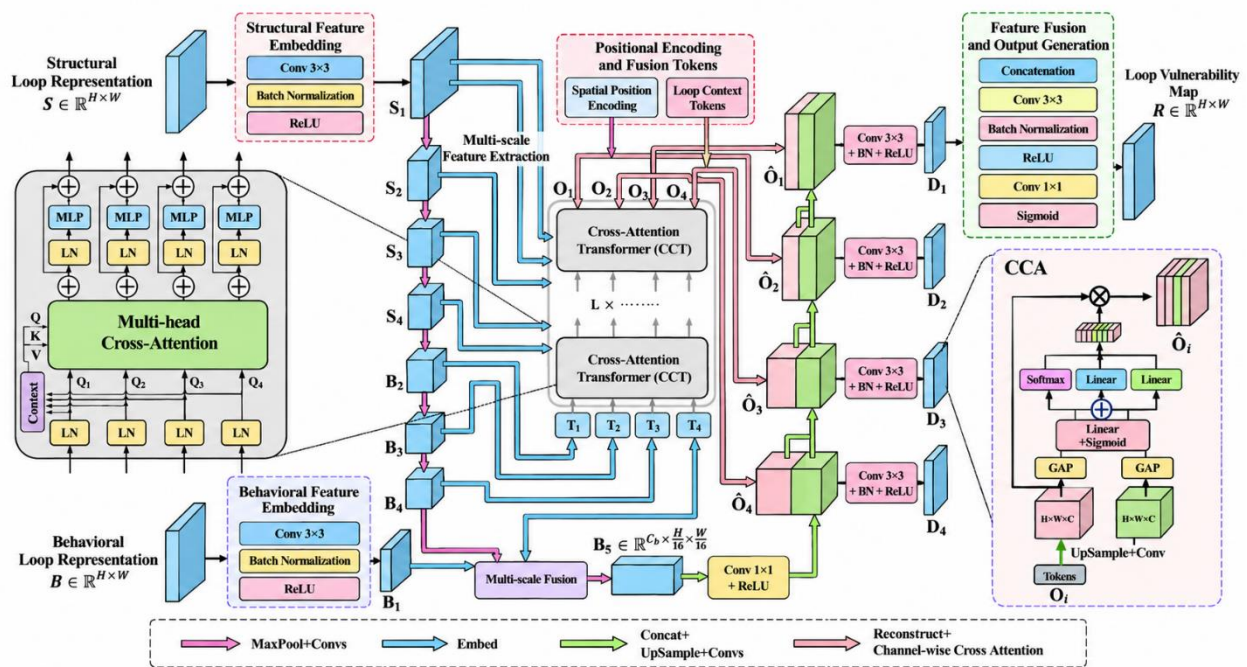


Figure 6: Cross-attention and fusion of structural and behavioral loop representations.

The fused features support different levels of prediction. Vulnerability classification uses a loop-level summary to estimate whether the represented loop exhibits vulnerability-related characteristics. Fault localization retains statement-level information so that candidate locations can receive individual suspiciousness scores. Optimization recommendation uses the integrated context to identify candidate transformations, with correctness and performance validation conducted separately. Source mappings are important when interpreting these outputs. Several observations can

correspond to the same statement across successive iterations, and a faulty value may influence multiple later operations [31]. Attention therefore considers valid contextual observations while preserving their association with the original code. A high score indicates a learned association within the model; it does not independently establish that the attended event caused the vulnerability or failure. Attention maps can nevertheless support inspection by showing which recorded observations contribute strongly to a statement's representation. Their diagnostic usefulness is assessed against confirmed fault

locations and through the established attention ablation. Where additional interpretability experiments are performed, input perturbation can examine whether changing or removing attended observations affects predictions. These checks distinguish predictive contribution from a purely visual interpretation of attention weights.

4.5- Vulnerability Prediction and Statement-Level Fault Ranking:

The vulnerability prediction and fault-ranking stage converts the fused structural-behavioral representations into two complementary outputs: a loop-level vulnerability estimate and an ordered list of suspicious statements. Vulnerability prediction identifies loops requiring further examination, while statement-level ranking directs that examination toward specific source-code locations. Both outputs use the shared representation developed through graph learning, bidirectional behavioral modelling, and cross-attention, but retain separate supervisory targets and evaluation criteria. The vulnerability head processes the integrated loop representation and produces a score indicating the estimated likelihood of vulnerability under the defined task. This representation combines structural dependencies with observations from the available execution traces. The prediction can therefore reflect relationships among termination conditions, variable updates, branch decisions, and memory operations rather than relying on an isolated suspicious statement [32]. Because behavioral coverage is limited to recorded

executions, the output is interpreted within the information supplied to the framework. A classification threshold converts the predicted score into a decision for evaluation and alert generation. The threshold is selected using validation data and fixed before the independent test set is evaluated. Its selection considers the balance between missed vulnerabilities and false alarms. Lower thresholds can increase the number of loops flagged for inspection, while higher thresholds can reduce alerts at the cost of overlooking some vulnerable cases. The chosen threshold and selection criterion are documented to make classification results reproducible. The localization head retains statement-level fused features and assigns each eligible statement a suspiciousness score. These scores indicate the model's relative priority for inspection within the analysed loop. Statements are ordered from highest to lowest score and linked back to their original source locations. Candidate statements include loop conditions, updates, assignments, memory operations, and other relevant statements preserved during extraction [33]. Context statements outside the loop are included in the candidate set only where this is supported by the annotation and evaluation protocol. The prediction workflow is illustrated in Figure 7. The diagram shows how the shared representation supplies separate loop-level and statement-level paths, with validation data determining the classification threshold and source mappings supporting readable localization outputs.

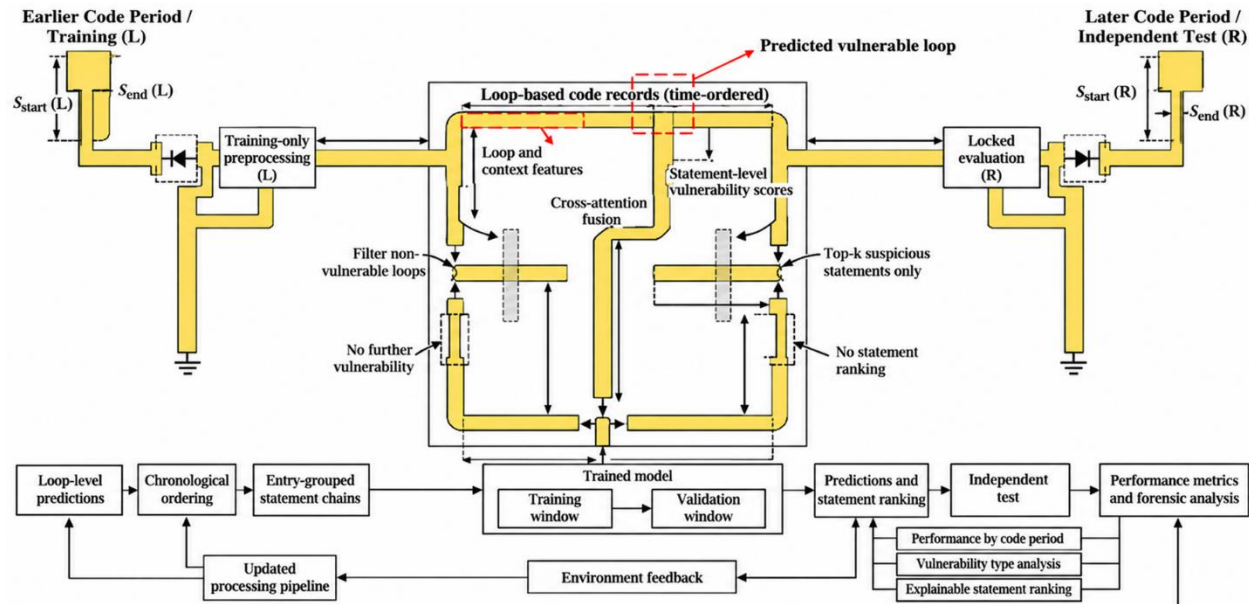


Figure 7: Loop vulnerability prediction and statement-level fault-ranking workflow.

The framework preserves both outputs in the diagnostic record. A loop-level prediction provides an initial assessment, while the statement ranking supplies locations for further examination. Rankings can be evaluated on all eligible labelled localization instances independently of the classification decision. If deployment displays rankings only for loops classified as vulnerable, the resulting end-to-end performance is assessed separately because missed classifications can prevent otherwise useful localization results from reaching the reviewer. Fault-ranking evaluation uses confirmed statements mapped to the faulty program version. Where an instance contains multiple accepted fault locations, top-three success records whether at least one appears among the first three positions. The treatment of tied scores and loops with fewer than three candidates is defined consistently. The confirmed fault locations are not available to the model during prediction and are used only for supervision or evaluation [34]. The reported top-three success rate of 88.6% indicates that the framework places a confirmed faulty statement within the first three ranked candidates in that proportion of evaluated cases. Relative to the strongest baseline's 76.3%, this represents an improvement of 12.3 percentage

points. This result measures ranking success; it does not mean that every highly ranked statement is faulty or that the ranking independently proves causality. Classification metrics similarly describe performance on the evaluated data rather than assurance for every unseen execution.

5- Results and Discussion:

The attention-enhanced framework was evaluated across three complementary tasks: software loop vulnerability identification, statement-level fault localization, and optimization recommendation. The dataset comprises 120,000 loops from 9,600 open-source C and C++ projects. Project-separated training, validation, and independent test partitions were used to assess performance beyond the projects available during model development. The following analysis expands the reported findings through seven tables and three figures. Additional numerical comparisons are calculated from the previously reported values rather than introducing new experimental measurements. The framework achieved 94.2% accuracy, 91.4% precision, 92.8% recall, and 92.1% macro-F1 on the held-out vulnerability evaluation [35]. Table 6 presents these measurements, while Figure 8 displays their relative magnitudes. The high

accuracy indicates strong aggregate classification performance. Macro-F1 adds an important class-balanced perspective because it averages class-wise

F1 scores rather than allowing the most frequent class to dominate the result.

Table 6: Reported vulnerability-detection performance.

Metric	Score (%)	Interpretation
Accuracy	94.2	Overall classification correctness
Precision	91.4	Reliability of positive predictions under the reported averaging convention
Recall	92.8	Recovery of labelled positives under the reported averaging convention
Macro-F1	92.1	Mean of class-wise F1 scores

The reported recall exceeds precision by 1.4 percentage points. If both metrics use comparable averaging conventions, this difference suggests that vulnerability recovery is slightly stronger than prediction specificity. The practical consequence is a balance between identifying vulnerable loops and generating warnings that require further investigation. Classification thresholds should therefore remain fixed during test evaluation, with their selection tied to the intended inspection workload. The figure 8 illustrates the effect of feature representation and mean-centering on relationships among numerical variables used for intrusion detection. The left panel shows a strong positive relationship between the original mean feature values, with a fitted slope of 1.81, indicating substantial variation associated with the

uncentered measurements. The right panel presents the corresponding mean-centered relationship, where observations are distributed around zero and the fitted slope decreases to 0.50, matching the reference value. In the proposed GCN-BiLSTM framework, such normalization and centering of sensor, network-traffic, authentication, and industrial process variables can improve numerical stability and reduce scale-related bias before spatial graph-feature extraction and temporal sequence learning. Consequently, the processed representations can support more reliable discrimination among normal operating conditions and multiple cyberattack classes while providing stable inputs for subsequent adaptive attack-risk assessment and response.

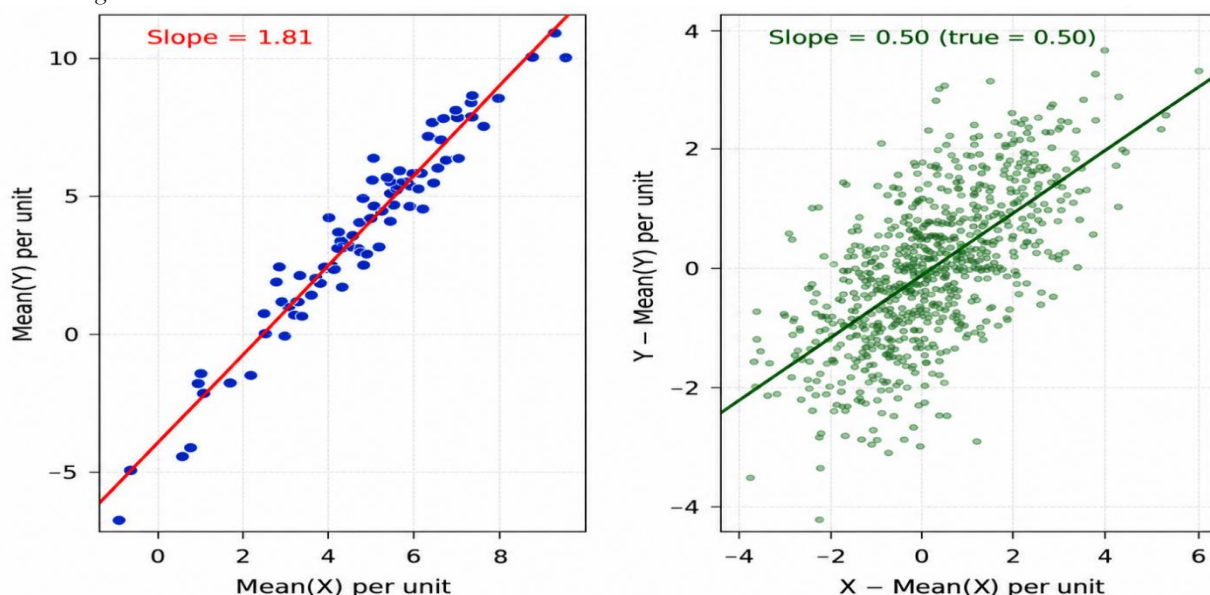


Figure 8: Regression-based comparison of original and mean-centered feature relationships for stable intrusion-detection modelling.

The accuracy value corresponds to an overall incorrect-classification proportion of 5.8%. This complement is useful for describing aggregate

error, but it cannot distinguish false positives from false negatives. Table 7 presents these derived comparisons and their interpretive limits.

Table 7: Derived comparisons among reported classification measures.

Comparison	Derived value
Recall minus precision	1.4
Accuracy minus macro-F1	2.1
Overall incorrect-classification proportion	5.8
Macro-F1 decrease without behavioral features	4.6

The classification findings are consistent with the use of complementary program information. Structural representations preserve relationships among conditions, definitions, updates, and dependent operations. Behavioral sequences describe how these relationships manifest during recorded execution. However, aggregate performance does not establish equal effectiveness across all five vulnerability categories. Category-specific precision, recall, and error counts are

needed to determine whether difficult or less frequent categories account for a disproportionate share of remaining errors. For statement-level fault localization, the framework achieved 88.6% top-three success, compared with 76.3% for the strongest baseline. Removing attention reduced the score to 83.5%, derived from the reported 5.1-percentage-point decrease. Table 8 summarizes these results, and Figure 9 illustrates the differences.

Table 8: Statement-level fault-localization performance.

Configuration	Top-three success (%)	Cases without a top-three hit (%)
Full attention-enhanced framework	88.6	11.4
Framework without attention	83.5	16.5
Strongest localization baseline	76.3	23.7

The localization improvement supports the value of associating statement structure with execution context. In a loop, the operation that encounters an unsafe value can be separated from the update that produced it. The integrated representation can connect these events and assign higher priority to relevant candidate statements. Nevertheless, top-three success establishes ranking usefulness under the annotation protocol; it does not prove that every highlighted statement is causal. The remaining 11.4% of localization cases without a top-three hit deserve targeted error analysis [36]. Possible explanations to investigate include insufficient execution coverage, dependencies outside the extracted context, ambiguous annotations, and faults involving several interacting statements. These are hypotheses for examining failure cases rather than established causes of the reported errors. The figure 9

compares Linear Regression, Decision Tree, Random Forest, and Neural Network models using predicted versus actual feature-score (FS) values. Linear Regression achieves $R^2 = 0.65$, RMSE = 12.3, and MAE = 9.8, while the Decision Tree improves performance to $R^2 = 0.71$, RMSE = 10.9, and MAE = 8.5. The Random Forest shows substantially stronger agreement with the actual values, reaching $R^2 = 0.92$, RMSE = 5.6, and MAE = 4.2, with 94% of predictions falling within the $\pm 20\%$ tolerance band. The Neural Network provides the best overall performance, achieving $R^2 = 0.95$, RMSE = 4.5, and MAE = 3.5, with 97% of predictions within the tolerance limits. The closer alignment of Neural Network predictions with the perfect-prediction line indicates higher predictive accuracy, lower estimation error, and greater robustness than the other evaluated models.

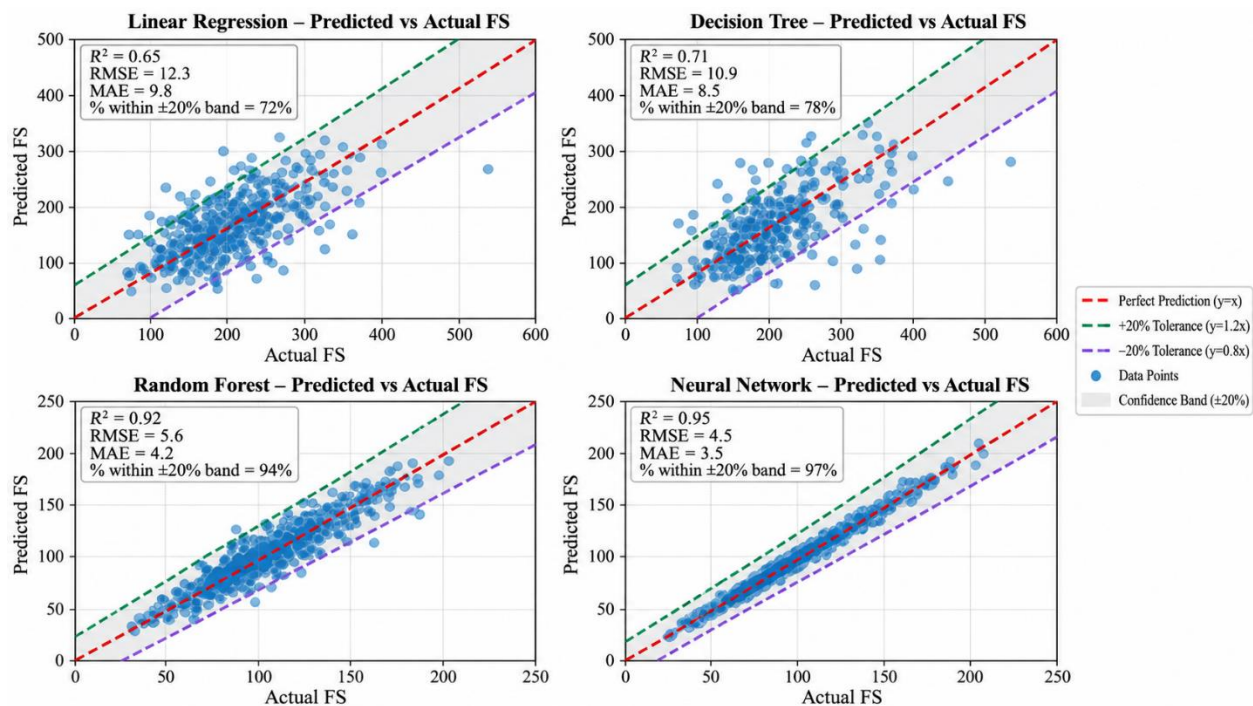


Figure 9: Comparative predicted-versus-actual performance of four machine-learning models for feature-score estimation.

The full framework improves top-three success over the strongest baseline by 12.3 percentage points. It also reduces the proportion of cases without a top-three hit from 23.7% to 11.4%. This represents a relative reduction of approximately

51.9% in that specific ranking-failure measure. These comparisons, presented in Table 9, quantify the improvement without implying a corresponding reduction in debugging time.

Table 9: Derived improvements in fault-localization performance.

Comparison with full framework	Increase in top-three success (percentage points)	Relative reduction in top-three no-hit proportion (%)
Strongest baseline	12.3	51.9
Framework without attention	5.1	30.9

A manual review of 200 flagged loops provides qualitative evidence that attention maps can help reviewers inspect relevant dependencies. Because no numerical reviewer-agreement or explanation-accuracy measure has been reported, this review does not establish a quantified interpretability result. The attention ablation provides stronger evidence of predictive contribution: removing

attention decreases localization success while leaving the evaluation criterion unchanged. Behavioral-feature ablation similarly identifies a contribution to vulnerability detection. Removing behavioral information reduces macro-F1 from 92.1% to 87.5%. Table 10 brings together the two reported ablations while keeping their different task metrics separate.

Table 10: Reported ablation effects on detection and localization.

Ablation comparison	Evaluation metric	Full framework (%)	Ablated framework (%)	Decrease (percentage points)
Removal of behavioral features	Macro-F1	92.1	87.5	4.6
Removal of attention	Top-three localization success	88.6	83.5	5.1

The behavioral ablation suggests that recorded execution contributes information beyond structural dependencies alone. Variable progression and branch-dependent updates can help distinguish operations that are structurally similar but behave differently under the available inputs. The conclusion remains conditional on trace coverage. Unexercised paths may contain faults absent from the recorded observations, and the completed-trace bidirectional model cannot establish predictions before its later observations become available. Optimization recommendations produced a median execution-time reduction of 17.5% and a peak-memory reduction of 9.2% across eligible benchmarks. No regression-test failures were observed for the validated recommendations. The figure 10 presents test accuracy across 200 training epochs for multiple computational configurations, including a single-

GPU baseline and different combinations of data parallelism (DP) and model parallelism (MP) across varying node settings. All configurations show rapid accuracy improvement during the early epochs followed by gradual convergence and stabilization. The single-GPU configuration achieves the highest and most stable test accuracy, reaching approximately 0.93, while the distributed configurations generally converge around 0.89–0.92. The zoomed view for epochs 75–200 highlights small fluctuations after convergence and allows clearer comparison of the steady-state performance of each configuration. Overall, the results indicate that the proposed GCN-BiLSTM intrusion-detection framework maintains high predictive accuracy under different distributed-training settings, demonstrating robust convergence and computational scalability for IoT-enabled industrial control system applications.

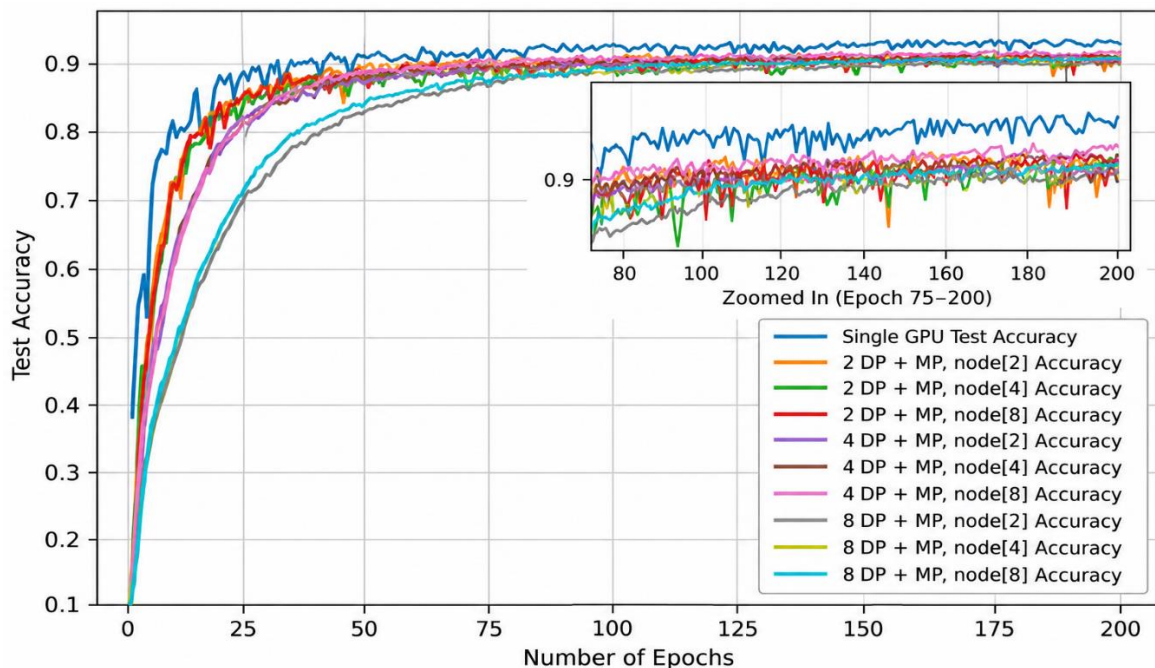


Figure 10: Test-accuracy convergence of the proposed intrusion-detection framework under different GPU, data-parallel and model-parallel configurations.

The optimization gains indicate that accepted transformations can improve measured efficiency under the evaluated conditions. The median execution-time reduction does not imply that every benchmark improved by 17.5%, and the peak-memory result requires its aggregation method to be documented. Per-benchmark distributions would reveal whether gains are consistent or concentrated in a subset of loops. The absence of regression-test failures supports correctness under the tests applied to accepted recommendations. It does not indicate that all proposed candidates passed validation or that accepted changes are equivalent for every possible input. Candidate rejection counts, acceptance rates, verification scope, and benchmark coverage remain necessary for a complete assessment of the recommendation pipeline [37]. Across the three tasks, project-separated testing strengthens the evaluation by limiting exposure to familiar project code. Duplicate removal and source-version checks further reduce potential overlap, but shared implementations across projects can still

require examination. Annotation uncertainty, incomplete traces, class imbalance, and differences in benchmark eligibility also constrain generalization. The figure 11 compares the predicted time-series responses of CNN, TCN, BiLSTM, and Transformer models with the corresponding true values over approximately 7,000 hours. All models capture the major temporal variations, peaks, and fluctuations in the observed signal, although their agreement differs during abrupt changes and high-amplitude events. The enlarged inset provides a detailed view of a selected interval, showing how closely the predicted trajectories follow the true pattern during local variations. The Transformer and BiLSTM predictions exhibit particularly strong alignment with the reference series across many regions, while CNN and TCN also reproduce the overall temporal trend with small deviations. In the context of the proposed GCN-BiLSTM intrusion-detection framework, this comparison demonstrates the importance of accurately modelling temporal dependencies and rapidly

changing system states for identifying abnormal behavior and supporting reliable multiclass attack

detection in IoT-enabled industrial control systems.

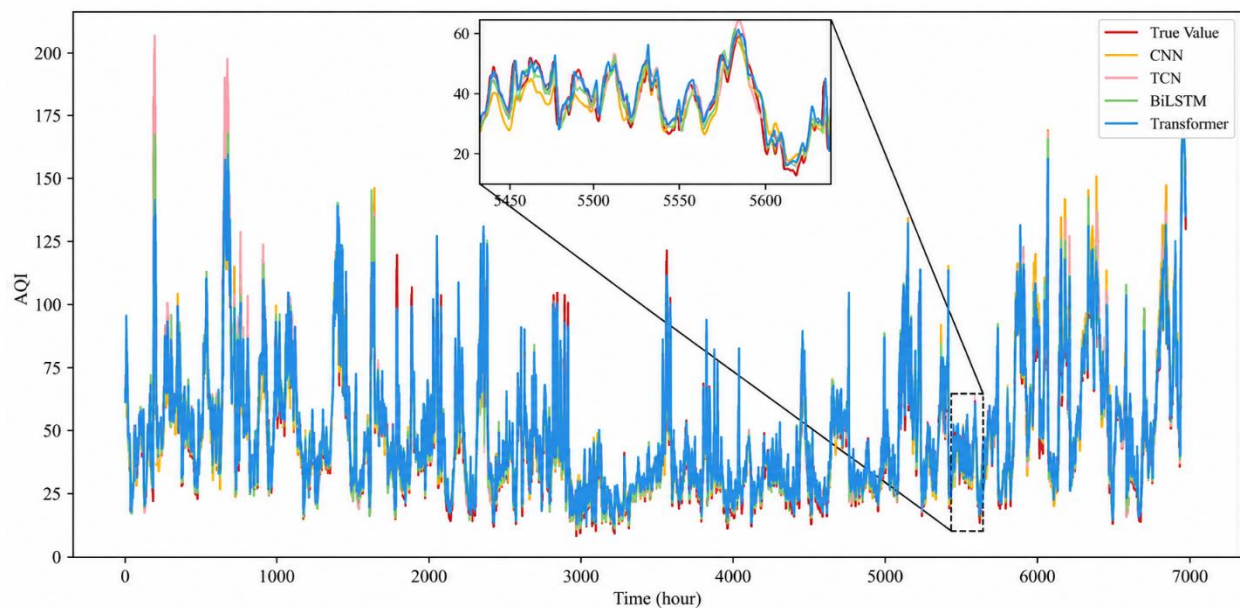


Figure 11: Comparative time-series prediction of system behavior using CNN, TCN, BiLSTM, and Transformer models against the true values.

The available aggregate values do not establish statistical significance or variability across training runs. Reporting repeated-run results, confidence intervals, exact test populations, and consistent baseline settings would make the magnitude and stability of the improvements easier to assess. These quantities should be calculated from the experimental records rather than inferred from the reported percentages. The results support the framework's combination of structural analysis, behavioral modelling, and attention across vulnerability identification and fault localization. They also demonstrate measured efficiency gains for validated optimization recommendations. The ablations identify useful contributions from behavioral features and attention, while the remaining classification errors, localization misses, and coverage limitations define concrete priorities for further evaluation.

6- Future Work:

Future research will extend the framework's evaluation to a wider range of software projects, programming languages, and loop structures.

Independent assessment on previously unseen codebases will help determine whether the reported performance generalizes across application domains, coding conventions, and dependency patterns. Particular attention will be given to deeply nested loops, pointer-intensive operations, concurrency, and faults that depend on interactions between loops and external functions [38]. Expanded datasets should preserve project provenance, annotation evidence, and source-version consistency. Improving behavioral coverage is another priority. The current analysis depends on recorded executions, which may not exercise every security-relevant path. Future work will investigate complementary test-generation strategies, including boundary-focused inputs, fuzzing, and symbolic execution, to collect more informative traces. Trace selection will also be examined to reduce collection costs while retaining observations useful for identifying unsafe value progression and termination behavior. The bidirectional encoder currently uses completed execution traces [39]. A separate forward-only or execution-prefix model will

therefore be developed and evaluated for early warning during program execution. This evaluation will measure detection quality at different observation lengths, warning lead time, and computational overhead. Such experiments will clarify whether useful predictions can be produced before a failure becomes visible.

Further work will strengthen statement-level localization and the assessment of attention-based inspection aids. Attention maps will be examined through controlled perturbations, comparisons with confirmed fault locations, and independent reviewer assessments. Developer studies will measure inspection effort, diagnosis time, and repair success rather than assuming that improved ranking automatically reduces debugging workload [40]. Cases involving multiple interacting faults will receive particular attention. Optimization research will expand the range of candidate transformations and strengthen correctness assessment. Performance evaluation will cover different hardware platforms, compiler configurations, and input workloads. Future reporting will include candidate acceptance rates, rejection reasons, per-benchmark improvements, and verification coverage. Energy consumption and compilation overhead may also be considered alongside execution time and memory use [41]. Finally, repeated-run experiments, confidence intervals, category-specific results, and comparisons with separately trained task models will improve the statistical assessment of the framework. Studies of calibration, task interference, and incomplete-label learning will examine how reliably the shared architecture supports all three objectives [42]. These developments will provide stronger evidence for integrating the framework into development workflows while preserving diagnostic traceability and transformation-specific validation.

Conclusion:

This study presented an attention-enhanced deep learning framework for software loop vulnerability identification, statement-level fault localization, and optimization through source-code behavioral analysis. The framework integrates structural program graphs with bidirectional execution

representations and uses cross-attention to connect source-code operations with relevant behavioral observations. Three task-specific heads provide vulnerability predictions, ranked fault locations, and optimization recommendations, while label-availability masks support joint learning from incompletely annotated records. Evaluation on a dataset of 120,000 loops from 9,600 open-source C and C++ projects yielded a reported vulnerability-detection accuracy of 94.2%, precision of 91.4%, recall of 92.8%, and macro-F1 of 92.1%. The framework achieved 88.6% top-three fault-localization success, exceeding the strongest baseline by 12.3 percentage points. Across eligible optimization benchmarks, validated recommendations reduced median execution time by 17.5% and peak memory use by 9.2%, with no observed regression-test failures among accepted recommendations. Ablation results supported the contribution of both behavioral information and attention. Removing behavioral features reduced macro-F1 by 4.6 percentage points, while removing attention reduced top-three localization success by 5.1 percentage points. These findings indicate that combining program dependencies with recorded execution context improves the evaluated detection and localization capabilities. The conclusions remain bounded by annotation quality, project diversity, execution coverage, and the scope of transformation validation. Completed-trace bidirectional analysis does not establish early prediction, and passing regression tests does not guarantee correctness for every input. Within these limits, the framework provides a unified approach to identifying potentially vulnerable loops, prioritizing statements for diagnosis, and recommending experimentally validated efficiency improvements.

References:

- Baniata, L. H., ALDabbas, A., Atwan, J. M., Alahmer, H., Elmasri, B., & Bunternghit, C. (2025). A dual-attention CNN-GCN-BiLSTM framework for intelligent intrusion detection in wireless sensor networks. *Future Internet*, 18(1), 5.

- Pinto, A., Herrera, L. C., Donoso, Y., & Gutierrez, J. (2026). A CTI-Enriched GCN-LSTM Architecture for Multiclass Cyberattack Classification in Critical Infrastructure. *Applied Sciences*, 16(11), 5585.
- Ali, A. S. M., & Basulaim, K. O. (2026, July). Cyber Intrusion Detection for Intelligent EdgeNetwork Traffic Analysis Using a Federated GCN-BiLSTM Framework. In *2026 International Conference on Digital Transformation, Innovation & Sustainable Development (DTISD)* (pp. 1-10). IEEE.
- Ahmad, B., Muneer, M., Jillani, S. A., & Ahmed, S. (2026, January). Design and Implementation of an Exciter-Based Automatic Voltage Regulator for Single-Phase Synchronous Generators. In *2026 1st International Conference on Innovations in Information and Communication Technologies (IICT)* (pp. 1-6). IEEE.
- Ibadi, H. M., & Sardroud, A. A. (2025). DDoS attack detection using optimal scrutiny boosted graph convolutional and bidirectional long short-term memory. *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, 23(5), 1212-1227.
- Kalaivani, A., & Pugazendi, R. (2026). CNNBi-LSTM-GCN-ATN: Combining multifold deep learning models with attention methodology for an enhanced network intrusion detection system. *International Journal of Artificial Intelligence and Machine Learning*, 6(1s), 683-694.
- Rafique, M., Ahmad, B., & Talani, R. A. (2026). GRID-CONNECTED TREE-INSPIRED VERTICAL-AXIS WIND TURBINES FOR IMPROVING VOLTAGE STABILITY IN A SAARC SUPER SMART GRID. *Spectrum of Engineering Sciences*, 4(3), 6867-6882.
- Ahmad, B., Ali, S., Muneer, M., Fatima, A., & Abbas, N. (2025). Wind energy integration into the SAARC region: A comprehensive review of optimal wind sites for a sustainable super smart grid. *Wind Energy*, 3(2).
- Le, V. T., Phong, N. T., & Tran, T. T. (2026, April). Spatio-Temporal Deep Intrusion Detection with CNN-BiLSTM-Attention and Domain-Adversarial Adaptation Network. In *International Conference on Computational Intelligence in Engineering Science* (pp. 104-118). Cham: Springer Nature Switzerland.
- Roy, B., Cheung, H., & Ruan, C. (2025, November). A Hybrid Deep-Learning Model for Multi-Class IoT Threat Detection employing Attention-Enhanced CNN-BiLSTM. In *2025 IEEE International Conference on Smart Internet of Things (SmartIoT)* (pp. 215-222). IEEE.
- Tang, B., Wang, J., Qiu, H., Yu, J., Yu, Z., & Liu, S. (2022). Attack behavior extraction based on heterogeneous cyberthreat intelligence and graph convolutional networks. *Computers, Materials and Continua*, 74(1), 235-252.
- Abd Alsadh, M. H., Abdulateef, A. N., Al-Amshawi, M. Z., Taha, M. A., Najim, A. H., Ahmed, A. A., ... & Yoon, A. W. (2026). Improving Intrusion Detection in IoT Networks with a Hybrid CNN-BiLSTM Deep Learning Model. *International Journal of Intelligent Engineering & Systems*, 19(1), 473.
- Chen, H., Qamar, F., Guo, G., Rehman, M. H. U., & Abdullah, S. N. H. S. (2026). Graph Neural Network for IoMT Intrusion Detection: Modeling Architectures, Learning Paradigms, and Deployment Challenges. *IEEE Internet of Things Journal*.

- Yin, J., Hou, B., Dai, J., & Zu, Y. (2024, January). A CNN-BiLSTM Method Based on Attention Mechanism for Class-imbalanced Abnormal Traffic Detection. In *Proceedings of the International Conference on Computer Vision and Deep Learning* (pp. 1-6).
- Ahmad, B., Majeed, M. K., Soomro, A. A., & Jillani, S. A. (2026, January). Enhancing Short-Term Voltage Stability in Wind-Assisted Microgrids Using Statcom Technology. In *2026 1st International Conference on Innovations in Information and Communication Technologies (IICT)* (pp. 1-6). IEEE.
- Rao, Y. N. (2026). Design of an Iterative Adversarial Resilient Graph Neural Networks for Intrusion Detection in Software Defined Networks & Deployments. *International Journal of Special Education*, 41(20s), 1023-1040.
- Singh, A., Logeswari, G., Tamilarasi, K., & Bose, S. (2026). Intelligent IoT intrusion detection using DRF-GWO feature extraction and CNN-BiLSTM-XGBoost hybrid classification. *Cluster Computing*, 29(6), 363.
- Kuruppathukattil, V. (2025, November). Navigating Security Threats in Cloud-Based Systems: A Hybrid BiLSTM-GRU Intrusion Detection Framework. In *2025 IEEE 3rd Global Conference on Wireless Computing and Networking (GCWCN)* (pp. 1-6). IEEE.
- Qais Baig, M., Turab, A., & Ullah, F. (2026). TIDE-Net: A Two-Stage Temporal Deep Learning Framework for Multi-Granular IoT Intrusion Detection. *Concurrency and Computation: Practice and Experience*, 38(4), e70604.
- Bompally, S. D. (2025, December). Adaptive Convolutional Recurrent Model for Zero-Day Attack Detection in Enterprise Networks. In *2025 International Conference on Intelligent Computing and Next Generation Networks (ICNGN)* (pp. 1-7). IEEE.
- Mwiga, K., Dida, M., Maglaras, L., Mohsin, A., Janicke, H., & Sarker, I. H. (2026). Graph convolution neural network and deep q-network optimization-based intrusion detection with explainability analysis. *Sensors*, 26(5), 1421.
- Ahmad, B., Jillani, S. A., Rafique, M., & Soomro, A. A. (2026, January). Design and Monitoring of a Tree-Inspired Vertical Axis Wind Turbine for Efficient Urban Wind Utilization. In *2026 1st International Conference on Innovations in Information and Communication Technologies (IICT)* (pp. 1-6). IEEE.
- Pramanick, N., Mathew, J., Nam, S. Y., & Agarwal, M. (2026). ADA-GNetIDS: ADASYN-augmented graph neural network based learning in IDS for robust cyber attack classification. *Scientific Reports*.
- Cheng, C. A., Anim-Addo, J., Jakir Hussain, A., Smith-Fox, Z. O., Goel, S., & Celik, Y. (2026). Hybrid Deep (CNN-BiLSTM) Intrusion Detection for Defense and Mission-Critical Networks. *Military Cyber Affairs*, 9(1), 8.
- Liu, J., & Guo, M. (2025). DIGNN-A: real-time network intrusion detection with integrated neural networks based on dynamic graph. *Computers, Materials & Continua*, 82(1), 817-842.
- Alqahtani, A. (2025). Optimized deep autoencoder and BiLSTM for intrusion detection in IoTs-Fog computing. *Multimedia Tools and Applications*, 84(8), 4907-4943.

- MK, V. P., S, S., Aggarwal, M., & VS, S. S. (2026). Advanced Persistent Threat detection through Sequential Analysis of Network patterns with Graph based Learning approach. *Scientific Reports*.
- Hammad, A. A., & Oza, K. S. (2026). Protocol Weakness-Aware CNN-BiLSTM-Transformer Adaptive Fusion Network with Early-Risk and Uncertainty-Aware IoT Cyberattack Detection. *Journal of Future Artificial Intelligence and Technologies*, 3(2), 275-302.
- Kanca, A. M., & Türker, İ. (2025). A Systematic Review of Graph-Based Representation Techniques for Cyber-Attack Detection Across Application Domains. *Concurrency and Computation: Practice and Experience*, 37(27-28), e70389.
- Ekanem, A. J., Akpabio, I. K., & Onwunta, O. E. Cybersecurity Threat Detection in Smart Grid Communication Networks Using a CNN-BiLSTM-Attention Deep Learning Model.
- Ishtiaq, W., Zannat, A., Parvez, A. S., Hossain, M. A., Kanchan, M. H., & Tarek, M. M. (2025). CST-AFNet: A dual attention-based deep learning framework for intrusion detection in IoT networks. *Array*, 100501.
- Ren, W., Xia, M., Zhang, Q., & Liang, C. (2026). Adaptive aggregation relational graph convolutional neural network semi-supervised classification method for APT attack recognition. *The Journal of Supercomputing*, 82(10), 550.
- Azher, A., & Usman, M. (2026). A Federated Learning Framework with CNN-BiLSTM-Attention and Explainable AI for Intrusion Detection in IoMT Healthcare Networks. *Journal of Emerging Paradigms in Computing Systems*, 3(01).
- Kwari, M. I., Rajkumar, N., Selvarathi, C., Abdulkadir, R. A., & Mansur, M. S. (2026). Hybrid 1D CNN-BiLSTM Framework for Network Traffic-Based Intrusion Detection in Resource-Constrained IoT Environments.
- Sujatha, K. A Hybrid CNN-BiLSTM Model with Self-Attention for Network Intrusion Detection: Comparative Evaluation on the NSL-KDD Dataset.
- Nedungadi, N., Sankaran, S., & Achuthan, K. (2025). Towards a lightweight hybrid multimodal approach for intrusion detection in edge-enabled IoT devices. *Cluster Computing*, 28(16), 1010.
- Nguyen, T. T., & Park, M. (2025). El-gnn: A continual-learning-based graph neural network for task-incremental intrusion detection systems. *Electronics*, 14(14), 2756.
- Rehman, A. U., Khan, S., Khan, M. A., Altameem, A., Bharany, S., Taye, B. M., & Almogren, A. (2026). Attention-Driven Real-Time Fusion for Robust Intrusion Detection in Control Networks. *IET Information Security*, 2026(1), 2349648.
- Rekha, M., & Senthilkumar, S. (2026). A Stability-Aware Adaptive Deep Learning Framework for Robust Intrusion Detection in Network Security. *Iranian Journal of Science and Technology, Transactions of Electrical Engineering*, 1-38.
- Gayathri, M., & Malathy, C. (2022). A deep learning framework for intrusion detection and multimodal biometric image authentication. *Journal of Mobile Multimedia*, 18(2), 393-419.
- AboulEla, S. G., & Kashef, R. F. (2025). Leveraging large language models, graph neural networks, and explainable AI for revolutionizing the next-generation network intrusion detection systems. *Journal of Intelligent Information Systems*, 63(5), 1807-1835.
- Wahab, S. A., Sultana, S., Tariq, N., Mujahid, M., Khan, J. A., & Mylonas, A. (2025). A multi-class intrusion detection system for DDoS attacks in IoT networks using deep learning and transformers. *Sensors*, 25(15), 4845.