

Detection of buffer overflow vulnerabilities in system software based on a graph and transformer model

Oleg Savenko^{1,†}, Piotr Gaj^{2,†} and Yevhenii Sierhieiev^{1,*,†}

¹ Khmelnytskyi National University, Instytut's'ka St, 11, Khmelnytskyi, 29000, Ukraine

² Silesian University of Technology, ul. Akademicka 2A, 44-100 Gliwice, Poland

Abstract

Buffer overflows remain one of the most dangerous classes of vulnerabilities in system software and embedded platforms, as they directly threaten memory integrity, lead to arbitrary code execution, and disrupt critical components. The paper presents a method for automated detection of stack and heap overflows and off-by-one errors, which combines formal risk conditions, a template library (CVE/NVD/OWASP), and code graph representations (AST/CFG/DFG) with multi-channel renders to train a three-class YOLO detector with class-specific refinement heads. The key innovation is the introduction of the effective buffer capacity and the nodal risk, which align the detection with formal criteria and reduce the number of false positives, while maintaining the interpretability of the results at the level of code subgraphs. The method works according to the pipeline principle: after instrumental parsing of the code, we build AST/CFG/DFG, perform annotation of buffers/lengths/checks and taint sources, match subgraphs with templates, generate multi-channel renders and apply the YOLO detector; each finding is accompanied by an explanatory report (class, score, code-span, matched-template, explanation). On real data from CVE/NVD and our own examples (over 12,000 fragments; 70:15:15 partitioning without project intersection), we achieved 95.7% accuracy, 94.6% F1-measures, and an average time of 8.7 s/file. The proposed approach outperforms the basic YOLO without graph features and regularisation and classic SAST tools (Cppcheck, Flawfinder), better localising vulnerable areas in industrial repositories and remaining suitable for integration into CI/CD (Docker, SARIF reports, blocking thresholds). Further work involves extending it to other memory-safety classes (integer overflow, use-after-free, race conditions), strengthening XAI components (contrastive explanations, local counterexamples), and integrating with automated fix suggestions at the code span level.

Keywords

cybersecurity, buffer overflow, machine learning, graphs, yolo, system software

1. Introduction

Memory safety issues in system and embedded software remain critical [1]: buffer overflow defects are still a common cause of incidents in RTOS[2], drivers, and microcontroller software[3], where resources are limited and time constraints are tight. In such conditions, conventional static quality assurance (SAST) tools often either generate an excess of false positives or miss atypical patterns, especially on large code bases with many dependencies [6]. In parallel, ML/AI approaches are actively developing: from sequence and language models to graph neural networks that better capture structural dependencies in program code [8, 9], as well as a line of work focused on explainability and robustness of models [Coca]. However, without a clear formal basis and agreed-upon risk templates, it is difficult to build a scalable and at the same time interpretable detector. In application system software (RTOS, drivers, MCU firmware), buffer overflows exhibit complex behaviour: stack limitations, API contract inconsistencies, lack of hardware isolation, and a limited “window” of time for checks contribute to the fact that even a single error in indexing or output format can cause a system crash or hidden privilege escalation. Standard protections (ASLR, DEP,

* AISSLE-2025: The International Workshop on Applied Intelligent Security Systems in Law Enforcement, October, 30–31, 2025, Vinnytsia, Ukraine

[†] Corresponding author.

[†] These authors contributed equally.

✉ savenko_oleg_st@ukr.net (O. Savenko); piotr.gaj@polsl.pl (P. Gaj); ysierhieiev@gmail.com (Ye. Sierhieiev)

ORCID 0000-0002-4104-745X (O. Savenko); 0000-0002-2291-7341 (P. Gaj); 0009-0008-9877-9863 (Ye. Sierhieiev)



© 2025 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

stack canaries) lower the attack surface but do not eliminate the fundamental issue—semantic errors in memory handling. This underpins the use of combining formal risk conditions with machine learning: the first enforces strict correctness limits, while the second enables scalable localisation and prioritisation of suspicious code sections.

This paper combines formal risk conditions and templates with the compact YOLO detector. We present the code in graph representations consistent with the template library (CVE/NVD/OWASP), and equations (1)–(4) from Section 2 define the reference risk criteria for localisation and explainability. The final solution is integrated into CI/CD, enabling rapid verification of changes in large repositories.

In this article, we mean a “buffer” to be a static array (stack), a dynamically allocated block (heap), or a polymorphic structure that encapsulates storage, provided that there is an explicit write operator. We consider a “bounds check” to be either a predicate check (a condition/assert with correct bounds) or a safe API version with a correctly specified size. When calculating the risk, we assume a conservative underestimation of the capacity (effective capacity), which guarantees the absence of false negatives at the pre-selection stage. A paginated journal article [2].

The goal of this work is to develop a universal method for detecting buffer overflow vulnerabilities that covers stack, heap, and off-by-one errors.

The main aspects of the article:

- A comparative review of modern ML/DL approaches to BOF detection in the context of system/embedded software;
- A formal risk basis in the form of four conditions from Section 3 and a template library aggregated from CVE/NVD/OWASP;
- A three-class YOLO detector (Stack/Heap/Off-by-One) on graph code renders with explained output (class, score, code-span, matched-template, explanation);
- An evaluation protocol and integration into CI/CD, where the quality gain is shown relative to basic SAST tools and current ML baselines.

2. Related works

Buffer overflow detection methods are conventionally divided into two large groups: classical static/dynamic analysers and machine learning-based approaches. The former rely on rules, heuristics, and data flow/control analysis: from simple signatures of dangerous calls (strcpy, sprintf, uncontrolled copies) to complex interprocedural checks using parse trees, control flow graphs (CFGs), and taint analysis. Symbolic execution and abstract interpretation enable proofs of reachability of errors, but suffer from “path explosion” and require complex invariant specifications. Dynamic approaches (fuzzing, sanitisers) provide high accuracy for reproducible execution traces but have significant resource and coverage costs. Tools like AddressSanitizer/UBSan detect defects at runtime, but do not guarantee coverage of “cold” code branches. Additionally, fuzzing is difficult to configure for state-dependent or multi-threaded scenarios. Additional complications are introduced by modern protection mechanisms (ASLR, DEP), which change the manifestations of exploits (ROP chains, heap spraying), without eliminating the root cause, namely the violation of boundary conditions and incorrect operation with lengths.

Machine learning approaches aim to train models to recognize vulnerable patterns in various code representations: text tokens, intermediate representation/bytecode, abstract and concrete graphs (AST/CFG/DFG/PDG). Early solutions used n-gram representations with classical classifiers (SVM, Random Forest), which worked well for repeated patterns, but often “broke” on non-obvious variations in code style and new APIs. Further progress is associated with code vectorization (code embeddings) and deep learning: recurrent networks and transformers effectively model sequences of tokens and instructions, and graph neural networks (GNN) add structural context — data and control dependencies, interactions between buffers and lengths. It was graph approaches that showed a noticeable increase in quality on large repositories, but faced other challenges: class imbalance (rare

off-by-ones), "spurious" correlations (the model is tied to uninformative features like variable names), sensitivity to domain shift (portability between projects and code styles), and lack of interpretability of results for developers.

In response, a third, hybrid line of work has emerged: combining formal/rule-based checks with machine learning (ML). The idea is to use "hard" security conditions (boundary checks, parameter contracts, length invariants) as filters/hints at the feature construction stage, while the model is trained on structured code representations, prioritizing subgraphs with increased risk. Such a synthesis provides two key advantages: it reduces the number of false positives in large repositories due to explicit "capacitance" logic and returns interpretability — each detection can be tied to a specific subgraph, boundary check, or violated condition, which facilitates triage and correction. At the same time, several open questions remain: how to stabilise generalizability on "atypical" code styles and new libraries; how to correctly aggregate local signals (byte streams, guard checks) to the module/project level; how to integrate machine learning risk assessment into the CI/CD process without excessive overhead. This is what our approach is aimed at: we combine formal risk conditions with a graph criterion and a specialised three-class detector to simultaneously maintain the rigour of checks and achieve scalability on industrial codebases.

2.1. Classical methods

Static analysers such as Cppcheck and Flawfinder have historically been the first line of defence against buffer overflows. They detect suspicious calls ("unsafe" string APIs, uncontrolled copies), missing bounds checks, and typical patterns from CWE databases. Modern implementations go beyond simple pattern search. They employ abstract syntax trees (AST), interprocedural data flow analysis (taint) and control flow analysis (CFG), simplified symbolic execution, and abstract interpretation to create proofs of error reachability. However, context sensitivity remains a key issue. Macros and conditional compilation vary the actual code for different configurations; C++ templates, inline assembler, and compiler optimisations hide the connections between data sources and sinks; cross-module dependencies require "program-wide" analysis that does not scale well. On large repositories, this results in "noise" from numerous false positives and missing non-trivial cases — for example, when safe wrappers over `strncpy` are combined so that the violation occurs only in a specific order of calls. This trade-off between soundness vs. completeness is inevitable: aggressively reducing false positives often leads to "blinding" on complex data chains.

Dynamic tools — AddressSanitizer/UBSan/MSan/TSan, tools like Valgrind, and fuzzing (coverage-guided AFL/libFuzzer) — work at runtime and capture real memory violations. Their advantage is high accuracy for reproduced traces: if an overflow is detected, the report is almost always valid and contains diagnostics at the instruction level. The disadvantages are also significant. First, overhead: instrumented assemblies increase the size and execution time, which makes them unsuitable for productive RTOS configurations and microcontrollers with limited resources. Second, coverage: even advanced fuzzing does not guarantee the achievement of "cold" branches, state-dependent paths, or races; carefully designed harness tests are required, which are expensive to maintain. Third, the instability of the environment: in drivers and embedded systems, real-world timings, interrupts, and hardware states are difficult to reproduce in tests, thus a significant portion of defects remain latent.

Prevention mechanisms such as ASLR, DEP/NX, stack canaries, FORTIFY_SOURCE and partial forms of CFI have made it much harder to exploit classic overflows, but they have not eliminated the underlying causes - boundary condition violations and incorrect length manipulation. Attack techniques have evolved: return-oriented / jump-oriented programming (ROP/JOP), heap-spraying and "heap feng shui" enable combining small information leaks and minor errors in bounds checking to bypass protection. These scenarios are especially critical for system and embedded software: thin stacks, lack of a full MMU, tight real-time deadlines and requirements for binary size leave little room for heavy protection and diagnostics.

In practice, engineering teams combine different approaches: fast SAST is run on every commit, dynamic tools are used during nightly builds and before releases, and manual code review is added

for “hot” modules. However, even such a combination results in an uneven outcome: the larger the repository, the higher the cost of warning triage becomes, and it becomes more challenging to guarantee stable dynamic coverage. This creates a niche for hybrid methods, where formal security conditions and structural criteria serve as “anchors,” and machine learning assists in scaling the localisation of vulnerable subgraphs and reducing “noise” at the level of large code bases. It is this combination that will be examined below.

2.2 Machine learning-based models

Today, artificial intelligence is showing progress from simple sequential models to graph and language architectures. Early approaches applied recurrent networks to instruction sequences and achieved significant gains in detecting stack overflows [1]. A parallel line explored network traffic signatures and employed classical ensembles (random forests) to diagnose remote BOF attacks, demonstrating high accuracy in incident detection [2]. A general review (2025) systematised ML/DL methods for BOF and highlighted open problems such as data imbalance, poor annotations, spurious correlations, and poor portability between projects [3].

Graph-based approaches have become predominant due to their ability to consider both syntax and semantics. The MSVAGraph model addresses various types of Binary Object Formats (BOFs) by combining program graph topologies with call features, leading to consistent improvements in accuracy [4]. SySeVR introduces a framework that merges syntactic and semantic representations of vulnerable fragments, enhancing detection capabilities [8]. MVD employs data flow-sensitive Graph Neural Networks (GNNs) to improve the localisation of vulnerable nodes, even in complex value transfer scenarios [9]. Additionally, the BiDirectional GNN in BGNN4VD takes bidirectional dependencies into account for better context selection, which is particularly beneficial in lengthy data paths [10].

Hybrid architectures are currently being developed that integrate sequential language models with attention mechanisms and domain-specific features. Research involving BiLSTM with attention, especially with specialised KAN components, has shown competitive results when applied to “linear” code representations. Additionally, the combination of transformers with graph aggregators, such as GraphSAGE, highlights the advantages of incorporating global context, particularly in applications related to system-level language code and the Go programming language [11].

Among the works of recent years, it is also worth mentioning Machine Learning-Based Network Anomaly Detection: Design, Implementation, and Evaluation (2024) [39], where the authors developed a comprehensive approach to searching for anomalies in network traffic using session features and SVM and XGBoost classifiers. The article Impact of Machine Learning on Intrusion Detection Systems for the Protection of Critical Infrastructure [40] (2021) systematises the experience of using ML to protect industrial systems and outlines directions for improving IDS. Deep Learning Approaches for Intrusion Detection Systems: A Survey [41] (2020) provides an overview of CNN-, RNN-, and autoencoder models that exhibit competitive results on the KDDCup and NSL-KDD datasets. The paper An Ensemble Deep Learning Approach for Cyberattack Detection in Cloud Computing [42] (2019) proposes an ensemble CNN+LSTM for attack detection in cloud environments, demonstrating high accuracy due to the combination of spatial and temporal features. The paper Malware Detection Using Convolutional Neural Networks and Transfer Learning [43] (2020) demonstrates how the use of pre-trained CNN architectures improves malware classification. The Hybrid Artificial Intelligence System for DDoS Attack Detection [44] (2022) presents a combination of neural networks and stochastic methods for early detection of DDoS flows. Finally, A Survey on Machine Learning and Deep Learning Techniques for Cybersecurity [45] offers a broad systematisation of ML/DL applications for intrusion detection, malware, and other cyber threats.

Recent research studies have explored large language models (LLMs) for code analysis and semi-automatic triage support [21], as well as mixed expert approaches MoEVD, where the type of CWE is first determined and then a specialised detector for a specific class is activated [19]. In parallel, work is emerging aimed at improving the robustness and explainability of models: “are we learning the

right features” analysis and spurious pattern detection [17], causal learning for cutting off false correlations CausalVul [18], counterfactual explainers CFExplainer for GNN detectors [24], and contrastive explainability methods Coca [25].

In addition to finding errors in the source code, artificial intelligence is widely used to detect malicious traffic, botnets, DDoS attacks, and hidden data. For example, Savenko et al. proposed a method for dynamic detection of malicious code based on API call tracing [31]. Pomorova et al. used a rich-agent architecture and fuzzy logic to detect botnets in corporate networks [32]. Lysenko et al. applied evolutionary algorithms to detect cyberattacks [33], and also investigated the detection of low-speed DDoS attacks using self-similar traffic features [34]. Other works apply the clonal selection algorithm [35] and semi-classical fuzzy clustering, fuzzy-c-means to detect DDoS botnets [36]. A method for detecting steganographic changes based on machine learning has been proposed for media content [37].

Despite significant progress, most approaches focus either on one subtype of overflow (most often stack overflow) or on a specific language/paradigm and rarely take into account the limitations of system/embedded software (RTOS, drivers, MCU environment). This is manifested in the instability of metrics in “atypical” cases, difficulties with portability between projects, and the lack of clear explanations for developers. That is why, further in this article, we rely on formal risk conditions and patterns from CVE/NVD/OWASP as an interpreted basis. We apply a three-class YOLO detector (Stack/Heap/Off-by-One) on graph code renderings on top to combine scalability, generalizability, and explainability in the context of system software.

Recently, the application of artificial intelligence to buffer overflow vulnerabilities in the context of integrated systems has also been described [38], which served as the basis for improving the proposed method.

3. Vulnerability models

To obtain a universal method for detecting buffer overflows, it is necessary to formalise the behaviour of memory writes at two complementary levels: a local one, where the required inequalities for a single operation or indexing are fixed, and a global one, where the composition of several individually safe actions within the program subgraph is taken into account. In our formulation, local conditions define the basic risk situations: exceeding the buffer capacity of the write, the time aspect for dynamic memory (heap), and the out-of-bounds error (off-by-one) for arrays. The global criterion (4) generalises these cases for subgraphs, where the combined effect of several copies or format outputs leads to exceeding the effective capacity even when the individual steps appear “legal”.

Such decomposition has practical and methodological value. First, the required local conditions are easy to check, and they agree well with known classes of errors in system/embedded software (C/C++ on RTOS, drivers, MCU), where `sprintf/strcat/memcpy` patterns and boundary loop errors are typical. Second, the global graph criterion adds context: chains of operations on a single buffer, combining multiple data paths within a single trace, or length accumulation through auxiliary buffers. This is where complex scenarios emerge: “safe” APIs in combination, underestimation of the null terminator, changing the block size after `realloc`, and merging branches with different check guarantees. In conclusion, (4) explains realistic incidents when overflow occurs not instantaneously, but as a result of the composition of actions.

Formal models also play a dual role in our system: (i) they serve as reference security criteria (which can be interpreted in reviews and CI/CD policies), (ii) they become definitions for machine learning - through the effective buffer capacity, data stream weights and consistency of detector predictions with risk assessments. The practical consequence is explainability: each trigger is not only localised to a code span, but also has a reference to a specific condition (1)–(4) and the corresponding pattern (Stack/Heap/Off-by-One), which significantly simplifies triage and prioritisation of fixes.

3.1. Stack overflow

Stack cases occur when the amount of writing to a local buffer exceeds its capacity according to formula (1). Typical consequences include overwriting neighbouring variables and call frame service data, such as return addresses. In practical attacks, this opens up the possibility of changing the execution flow; modern techniques (in particular ROP) combine small censuses and vulnerable code fragments, but at the heart of it is always a violation,

$$|x| > |b|, \quad (1)$$

where x is the write operation (copy, formatted output, concatenation), $|x|$ is the amount of data written in bytes (for string APIs it includes the null terminator, for `sprintf` it is the length of the formed string), b is the target buffer (array or allocated memory block), $|b|$ is the buffer capacity in bytes (by type, allocation location or parameter contract).

In further analysis, the stack class is considered as a separate shortcut for localisations where the buffer is placed on the stack and the write is performed without correct bounds checks.

A typical pattern for stack overflow is `sprintf(buf, "%s.%s", a, b)` at fixed `buf` and unverified lengths `a`, `b`: nominally, the code is "legal", but it violates the relationship $|x| > |b|$ due to a combination of format output and concatenation.

3.2. Heap overflow

In dynamic memory, the buffer size changes during program execution. Therefore, the risk is given by the dynamic condition. Overwriting outside the block boundary leads to corruption of adjacent allocations or allocator metadata (e.g., free block list control fields), which in turn facilitates use-after-free, double-free, and workaround techniques such as heap spraying.

$$|x_t| > |b_t| \quad (2)$$

where x_t is the record size at time t , $|x_t|$ is the byte length of this data, b_t is the target buffer at time t , $|b_t|$ is the buffer capacity at time t , taking into account the allocator, alignment, possible realloc, and fragmentation, t is a discrete execution point between allocation/free operations.

A typical situation for heap overflow is repeated calls to `strcat` after `realloc` in computer system software: the capacity of a memory block could have decreased due to fragmentation, and thus $|x_t| > |b_t|$ becomes true only "at a certain step".

In the following sections, the time aspect and dependencies between `malloc`/`realloc`/`free` operations will be separately considered for heap cases.

3.3. Off-by-one errors

Off-by-one is a class of marginal indexing errors where access with an index outside the allowed range is allowed. Formally, such a case is defined by condition (3): there exists an index i , for which access is performed, $\text{Access}(A, i) = 1$ and at the same time $i < 0 \vee i \geq n$.

$$\exists i : \text{Access}(A, i) = 1 \wedge (i < 0 \vee i \geq n) \quad (3)$$

where A is an array of length n , n is the number of elements in the array, i is the access index, $\text{Access}(A, i) = 1$ is the fact of reading/writing element $A[i]$, $i < 0 \vee i \geq n$ is going beyond the permissible range $[0, n-1]$.

Common causes include conditions like $i \leq n$ instead of $i < n$, mixing signed and unsigned types, and underestimating the length during formatted output. The outcomes can resemble issues with stack or heap counts, such as corrupting neighbouring elements or metadata. Still, we treat off-by-one errors as a separate category because their indicators and check patterns differ.

The most common causes are $i \leq n$ in a loop or signed/unsigned mixing in data container checks.

3.4. Vulnerability generalization

To reconcile local cases with the global structure of the program, we use the graph criterion. Let $G = (V, E)$, each edge $e \in E$ is assigned a weight $w(e)$ (upper bound of the byte stream/probability of “unsafe” data). The effective capacity of a buffer node is denoted by

$$\sum_{e \rightarrow V_b} w(e) > cap_{eff}(V_b) \quad (4)$$

where $e \rightarrow V_b$ is the set of edges entering the buffer node V_b , $w(e)$ is the weight of edge e (upper bound of the byte stream or probability of “unsafe”/taint data), $cap_{eff}(V_b)$ is the effective (conservative) buffer capacity.

4. The proposed method

The proposed method combines formal risk conditions (1)–(4), a template library (CVE/NVD/OWASP), the creation of program graph representations (AST/CFG/DFG), and a compact YOLO detector with three classes (Stack, Heap, Off-by-One). Formal conditions serve as reference criteria: they help us select and annotate relevant subgraphs and also link future detections to specific reasons (such as which template and which condition is violated). YOLO offers scalable localisation of risk fragments and ranks them by confidence, working not with “raw” text but with multi-channel graph renderings, where control structures, data flows/taint, buffers, lengths, and the presence or absence of bounds checks are encoded. The overall scheme of the method is shown in Fig. 1.

4.1. Preprocessing and graph construction

The source code of the system software (C/C++ and languages for MCU) is converted into graph representations: abstract syntax tree (AST), control graph (CFG) and data flow graph (DFG). Nodes correspond to functions/blocks/variables and buffers, edges to calls and value transfers; edges are assigned weights $w(e)$, which are interpreted as upper estimates of the byte stream or the probability of receiving “dangerous” (taint) data. For the nominal capacity of the buffer node, $cap(V_b)$ we introduce a conservative effective capacity

$$cap_{eff}(V_b) = \gamma \cdot cap(V_b), \quad \forall 0 < \gamma \leq 1 \quad (5)$$

where V_b is a buffer node in the graph G , $cap(V_b)$ is the nominal buffer capacity, and $\gamma \in (0, 1]$ is a conservatism coefficient (compensates for type ambiguity, padding, fragmentation, overhead characters, etc.). Based on the effective capacity of the buffer node, we define the node risk as consistent with the graph criterion “gap” with normalisation and taking into account boundary checks and anomaly signals

$$R(V_b) = \min\left(1, \frac{\sum_{e \rightarrow V_b} w(e)}{cap_{eff}(V_b)}\right) \cdot (1 - \text{Check}(V_b)) \cdot P(\text{Anomaly}(V_b)) \quad (6)$$

where $\sum_{e \rightarrow V_b} w(e)$ is total input flow into buffer V_b , $cap_{eff}(V_b)$ is effective capacity, $\text{Check}(V_b) \in \{0, 1\}$ is “anomaly” score from rules/patterns (e.g., unsafe APIs, suspicious format strings, signed/unsigned, etc.). The value $R(V_b)$ is used as a weight/prior in subgraph selection and as a target for detector regularization.

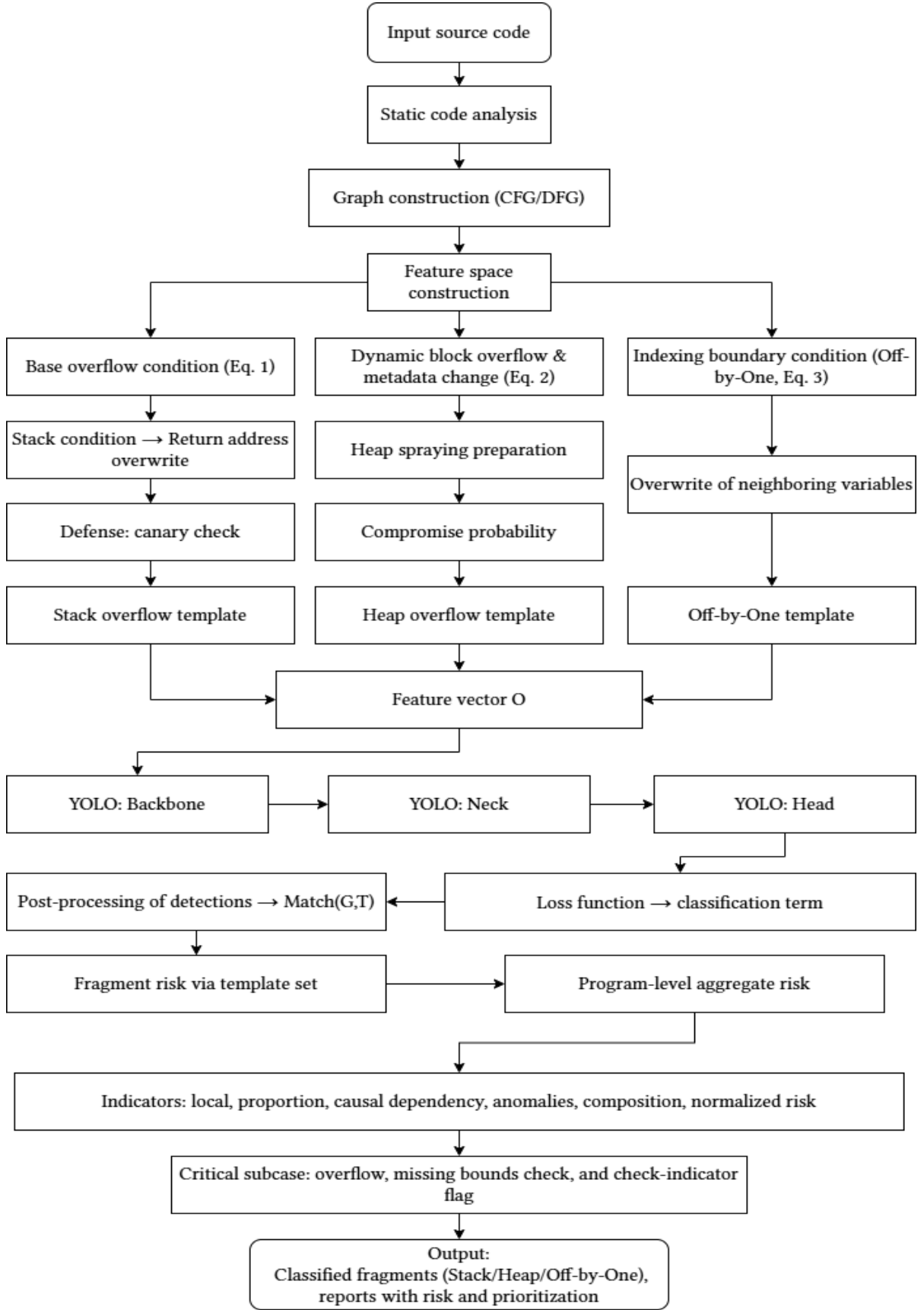


Figure 1: Architecture of the proposed model: the source code is converted to annotated AST/CFG/DFG (buffers, lengths, checks, taint), and matched with templates and calculates stream weights, then converts to a multi-channel image renderer, and passes YOLO (3 classes) with class-specific refinement heads, after which a report is generated (class, score, code-span, matched-template, explanation).

After the abstract, we generate multi-channel subgraph renders: channels encode control structures, data flows/taint, buffer/length indicators, and pattern matches; if necessary, an $R(V_b)$ channel is added.

4.2. Detector architecture

To localise and classify risky fragments, we use a compact YOLO detector with three classes (Stack/Heap/Off-by-One). The basic YOLO head produces candidate frames with objectivity scores and class logit values. To increase sensitivity to class-specific patterns, class-specific refinement heads (lightweight “experts”) are added, which receive features from the basic “neck” and are trained on the corresponding subclass (stack/heap/off-by-one). The selection of the refinement head is performed using arg max of the basic YOLO class-logits (without a separate router or heavy transformers).

The learning function combines standard YOLO components and risk-consistency regularisation

$$L = \lambda_{obj} L_{obj} + \lambda_{cls} L_{cls} + \lambda_{loc} L_{loc} + \lambda_{cons} L_{cons}, \quad (7)$$

where L_{obj} is objectivity loss, L_{cls} is classification (3 classes), L_{loc} is localization (frame regression), L_{cons} is consistency regularizer aligns the detector output estimate with scalar $R(V_b)$ (e.g., via BCE/Huber between normalised risk and score). The coefficients λ are selected for validation, taking into account class imbalance.

4.3. Integration into CI/CD

The detector is delivered as an isolated OCI-compliant Docker container that connects to Jenkins, GitLab CI or GitHub Actions pipelines without changing the build infrastructure. At each launch, the pipeline performs: verified source code download on a specific commit SHA or merge ref; incremental parsing of only changed modules with preprocessor reconfiguration for target profiles (debug/release, RTOS flags); AST/CFG/DFG construction with buffer, guard and taint source annotations; estimation of thread weights $w(e)$, effective capacity cap_{eff} and local risk indicators $R(V_b)$; rendering of multi-channel subgraph images; YOLO inference with three classes (Stack/Heap/Off-by-One); post-processing of detections, mapping of “frames” to code spans and report generation.

There are two standard execution profiles. The “fast” profile analyses the delta of changed files for each commit, applies increased thresholds for score and/or R, limits the number of top-k detections, and guarantees short feedback within CI time budgets. The “full” profile runs before a release or upon request from a security audit, covers all modules, saves render artefacts, and returns a detailed report for review. The project policy defines blocking thresholds: for example, any critical finding in the stack paths of core components immediately “fails” the job; in less critical modules, triggers are translated into mandatory “warning fixes” before merging.

Reports are generated in SARIF for automatic annotation of merge/pull requests and in machine-readable JSON for further analysis. Each record contains: class (Stack/Heap/Off-by-One), score, code-span (file and line range), matched-template (template identifier from CVE/NVD/OWASP library), short explanation (which bounds check is missing or why the composition of operations generates a violation), as well as service fields (render ID, model version, launch profile). To accelerate triage, reports are grouped by modules and priorities, and deep links are added to the code viewer with highlighting of the relevant lines.

Intermediate views (AST/CFG/DFG) and render tiles are cached in relation to file hashes. This reduces the cost of repeated analyses. The tool supports module-based parallelisation and GPU sleep mode (if available) or full CPU inference in separate runner-ax. For embedded/RT contexts where

feedback time is important, an early exit policy and selective rendering of only subgraphs with the highest $R(V_b)$ are allowed.

Feedback to developers is built into the process: detections are labelled with labels (severity/type), automatically assigned to the module owner, and short remediation recommendations are added for typical patterns (sample bounds check or safe API option). In the event of systematic false positives, a feedback loop is implemented. The security analyst approves/rejects the findings in the UI/MR comments, after which these decisions are exported as training events to the internal “markup repository”. The model supports retraining on internal organisational data with versioning of artefacts (container, weights, dataset hash) and safe “canary-rollout” on a subset of repositories. In case of metric degradation, an automatic rollback to the previous stable version is performed.

The container has no external network dependencies during inference, operates in air-gapped mode, and all temporary artefacts are stored in an isolated volume with a regulated lifetime. For each run, checksums of the source files and configurations (compiler versions, preprocessor flags) are recorded, allowing you to reproduce the results during auditing. By default, the detector does not alter the code or cause the pipeline to crash in case of a technical parsing error; instead, it returns a “degraded-mode” status along with a clear list of modules that should be further checked using static methods. Integration parameters (thresholds, profiles, severity mapping to blocking policy, exclusion directories) are configured declaratively and stored with the code, making the process transparent to development and security teams.

Therefore, we can conclude that the proposed integration differs from classical SAST/DAST and “pure” DL detectors in that it combines formal conditions as first-class signals with multi-channel graph renderers and three class-specific YOLO heads. Unlike token or bytecode models, we localise subgraphs with a binding to the template and code span, which dramatically reduces false positives and triage costs. Consistency of detector scores with $R(V_b)$ provides manageable thresholds for CI/CD policies and improves stability on large repositories. Incremental parsing, render caching, and SARIF reporting provide short feedback without changing the build infrastructure. Unlike well-known ML approaches, the solution operates in an “air-gapped” mode, supports canary rollouts and rollbacks of models, and produces explainable reports (class, score, code-span, matched-template, explanation). This combination of formal rigour, structural context, and engineering applicability makes the method uniquely suited for drivers, RTOS components, and MCU firmware, and appropriate for continuous integration and deployment.

5. Experimental results

5.1. Data set and methodology

To evaluate the effectiveness, we used open CVE/NVD repositories and our own system code samples. In total, over 12,000 fragments were generated: 5,100 with stack overflows, 3,200 with heap overflows, 2,700 with off-by-one errors; the rest are safe examples for balancing. To avoid leakage between sets, the data is split in a ratio of 70:15:15 without crossing projects (split by repositories, not files). An AST/CFG/DFG was constructed for each fragment; buffers, lengths, bounds checks, and taint sources were annotated. On the subgraphs, there is automatic marked matching of patterns (CVE/NVD/OWASP), is determined effective capacity cap_{eff} and local risk indicators $R(V_b)$. During the selection of positives/negatives, duplicate commits and trivial test artefacts were being removed.

Cppcheck and Flawfinder (current stable versions with “safe string API” profiles) were chosen as the basic comparison tools. For the neural base, a YOLO detector without graph channels and without regularisation of consistency with $R(V_b)$ was used — only “flat” renders without special features.

The markup quality control was conducted in two phases: automatic template seeding (CVE pattern coverage) and selective manual verification of “complex” cases (loops, macros, conditional compilation). For off-by-one errors, stratified selection and synthetic variations of boundary conditions (only in training) were used to enhance the representation of rare patterns.

5.2. Training parameters

The detector is built using YOLO. Batch size is 32, Adam optimiser with an initial learning rate of 0.0005. The training lasted 80 epochs. Input subgraph renders are scaled to 640×640, with standard augmentations (flip/scale, light colour jitter) applied. The base YOLO head has been augmented with class-specific refinement heads (for Stack/Heap/Off-by-One), which are trained on the respective subclasses. The loss function involves $\lambda_{obj}\lambda_{cls}\lambda_{loc}$ selected through validation, while λ_{cons} activates consistency regularisation with $R(V_b)$.

5.3. Results and comparisons

Table 1 presents a comparison with basic YOLO and traditional tools. Precision, Recall, and F1-measure were assessed.

Table 1

Tools compare

Tool	Precision	Recall	F1-measure
YOLO (ours)	95,7 %	93,5 %	94,6 %
YOLO (base)	94,3 %	91,8 %	93,0 %
Cppcheck	76.2%	70.4%	73.2%
Flawfinder	72.5%	68.9%	70.6%

The proposed method surpasses the basic YOLO across all metrics. The improvement results from combining multi-channel graph renders, formal conditions, a consistency regulariser with $R(V_b)$, and class-specific refinement heads. The average processing time for one file is 8.7 seconds, which is acceptable for integration into CI/CD.

5.4. Discussion

High accuracy (95.7%) and completeness (93.5%) suggest that the model detects both typical and “atypical” BOF patterns. The introduction of cap_{eff} and risk $R(V_b)$ enhances the localisation of suspicious areas and the prioritisation of notifications, while graph renderings decrease the number of false matches caused by structural context. The limitation remains in constructing graphs for very large code bases (resource intensive), but this is balanced by the explainability and stability of the metrics. Additionally, the proposed approach shows higher metrics than the systems with CNN+LSTM or XGBoost mentioned in [39, 44, 45], mainly due to the use of formal features and structural analysis of the code.

6. Conclusions

A method for detecting buffer overflows in system software is proposed, combining formal conditions for determining the effective capacity cap_{eff} , the risk indicator $R(V_b)$, and a compact YOLO detector with class-specific refinement heads (Stack/Heap/Off-by-One). Formal conditions guarantee the interpretability and controllability of thresholds, a template library (CVE/NVD/OWASP) maintains consistency with recognised patterns, and multi-channel graph

renderers (AST/CFG/DFG) offer a structural context for accurate localization of subgraphs. On real data from CVE/NVD, we achieved an accuracy of 95.7%, an F1-measure of 94.6%, and an average time of 8.7 seconds per file, which outperforms the basic YOLO without graph channels and regularization, as well as classic SAST tools (Cppcheck, Flawfinder). The approach is compatible with CI/CD technology: it supports incremental analysis, artefact caching, SARIF reporting, and policies for blocking releases.

The practical value of the method lies in reducing "noise" in large repositories and streamlining triage: each finding is accompanied by class, score, code-span, matched-template, and a brief explanation, which enables you to quickly assign the defect to the appropriate team and determine the urgency of the fix. Matching detector estimates with $R(V_b)$ enhances notification prioritisation and decreases spurious matches in modules with many correct boundary checks. Along with the ability to retrain on internal data, this makes the method suitable for drivers, RTOS components, and MCU firmware, where time and resource constraints are critical.

However, limitations still exist in graph construction, which can be resource-intensive for very large codebases; porting to other languages (Rust/Go) requires adapting parsers and templates. Overall quality relies on the completeness of the markup and coverage of project configurations (macros, build flags). These risks are mitigated by incremental mode, caching, and the "fast profile" policy in CI.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

- [1] Dahl, W. A., Erdodi, L., Zennaro, F. M. Stack-based buffer overflow detection using recurrent neural networks. arXiv:2012.15116, 2020. doi: 10.48550/arXiv.2012.15116.
- [2] Li, S., Zheng, R., Zhou, A., Liu, L. A machine learning-based method for detecting buffer overflow attack with high accuracy. In: Proceedings of the 2020 International Conference on Computer, Network, Communication and Information Systems (CNCI), 2020. doi: 10.23977/CNCI2020090.
- [3] Kanaan, E., Alam, S. S., Akter, M. S. Survey of machine learning techniques for detecting buffer overflow vulnerabilities. In: Proceedings of the 8th International Conference on Engineering Research, Innovation and Education (SUST), 2025. doi: 10.13140/RG.2.2.22978.08640.
- [4] Zheng, Z., Liu, Y., Zhang, B., Liu, X., He, H., Gong, X. MSVAGraph: a multitype software buffer overflow vulnerability prediction method based on self-attentive graph neural network. Information and Software Technology, 2023. doi: 10.1016/j.infsof.2023.107246.
- [5] Zhai, J., Qi, Z., Yang, H. Stack overflow vulnerability detection based on BiLSTM-attention KAN deep learning model. The Journal of Supercomputing, 2025. doi: 10.1007/s11227-025-07605-z.
- [6] Luo, P., Zou, D., Du, Y., Jin, H., Liu, C., Shen, J. Static detection of real-world buffer overflow induced by loop. Computers & Security, 2019. doi: 10.1016/j.cose.2019.101616.
- [7] Zou, D., Wang, S., Xu, S., Li, Z., Jin, H. μ VulDeePecker: a deep learning-based system for multiclass vulnerability detection. IEEE Transactions on Dependable and Secure Computing, 2019. doi: 10.1109/TDSC.2019.2942930.
- [8] Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z. SySeVR: a framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing, 2021. doi: 10.1109/TDSC.2021.3051525.
- [9] Cao, S., Sun, X., Bo, L., Wu, R., Li, B., Tao, C. MVD: memory-related vulnerability detection based on flow-sensitive graph neural networks. In: Proceedings of the International Conference on Software Engineering, 2022. doi: 10.1145/3510003.3510219.
- [10] Cao, S., Sun, X., Bo, L., Wei, Y., Li, B. BGNN4VD: constructing bidirectional graph neural network for vulnerability detection. Information and Software Technology, 2021. doi: 10.1016/j.infsof.2021.106576.

- [11] Yuan, L., Fang, Y., Zhang, Q., Liu, Z., Xu, Y. Go source code vulnerability detection method based on graph neural network. *Applied Sciences*, 2025. doi: 10.3390/app15126524.
- [12] Wang, H., Qu, Z., Sun, L. E-GVD: efficient software vulnerability detection techniques based on graph neural network. 2024. doi: 10.4108/eetsis.5056.
- [13] Chen, J., Yin, Y., Cai, S., Wang, W., Wang, S., Chen, J. iGnnVD: a novel software vulnerability detection model based on integrated graph neural networks. *Science of Computer Programming*, 2024. doi: 10.1016/j.scico.2024.103156.
- [14] Yang, J., Ruan, O., Zhang, J. TensorGNN: tensor-based gated graph neural network for automatic vulnerability detection. *Software Testing, Verification & Reliability*, 2023. doi: 10.1002/stvr.1867.
- [15] Hin, D., Kan, A., Chen, H., Babar, M. A. LineVD: statement-level vulnerability detection using graph neural networks. In: *Proceedings of the 19th International Conference on Mining Software Repositories (MSR)*, 2022. doi: 10.1145/3524842.3527949.
- [16] Yang, X., Wang, S., Li, Y., Wang, S. Does data sampling improve deep learning-based vulnerability detection? Yes! and No!. In: *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, 2023. doi: 10.1109/ICSE48619.2023.00192.
- [17] Das, S., Fahiha, S. T., Shafiq, S., Medvidovic, N. Are we learning the right features? A framework for evaluating DL-based software vulnerability detection solutions. *arXiv:2501.13291*, 2025. doi: 10.48550/arXiv.2501.13291.
- [18] Rahman, M. M., Ceka, I., Mao, C., Chakraborty, S., Ray, B., Le, W. Towards causal deep learning for vulnerability detection. *arXiv:2310.07958*, 2024. doi: 10.48550/arXiv.2310.07958.
- [19] Yang, X., Wang, S., Zhou, J., Zhu, W. One-for-all does not work! Enhancing vulnerability detection by Mixture-of-Experts (MoE). *arXiv:2501.16454*, 2025. doi: 10.48550/arXiv.2501.16454.
- [20] Schaad, A., Binder, D. Deep-learning-based vulnerability detection in binary executables. *arXiv:2212.01254*, 2022. doi: 10.48550/arXiv.2212.01254.
- [21] Zhou, X., Zhang, T., Lo, D. Large language model for vulnerability detection: emerging results and future directions. *arXiv:2401.15468*, 2024. doi: 10.48550/arXiv.2401.15468.
- [22] Nguyen, V.-A., Nguyen, D. Q., Nguyen, V., Le, T., Tran, Q. H., Phung, D. ReGVD: revisiting graph neural networks for vulnerability detection. *arXiv:2110.07317*, 2021. doi: 10.48550/arXiv.2110.07317.
- [23] Zeng, Q., Xiong, D., Wu, Z., Qian, K., Wang, Y., Su, Y. TACSAN: enhancing vulnerability detection with graph neural network. *Electronics*, 13(19), 3813, 2024. doi: 10.3390/electronics13193813.
- [24] Chu, Z., Wan, Y., Li, Q., Wu, Y., Zhang, H., Sui, Y., Xu, G., Jin, H. Graph neural networks for vulnerability detection: a counterfactual explanation. *arXiv:2404.15687*, 2024. doi: 10.48550/arXiv.2404.15687.
- [25] Cao, S., Sun, X., Wu, X., Lo, D., Bo, L., Li, B., Liu, W. Coca: improving and explaining graph neural network-based vulnerability detection systems. *arXiv:2401.14886*, 2024. doi: 10.48550/arXiv.2401.14886.
- [26] Liu, R., Wang, Y., Xu, H., Sun, J., Zhang, F., Li, P., Guo, Z. Vul-LMGNNs: fusing language models and online-distilled graph neural networks for code vulnerability detection. *Information Fusion*, 115, 102748, 2025. doi: 10.1016/j.inffus.2024.102748.
- [27] Zhu, J. Research on software vulnerability detection methods based on deep learning. *Journal of Computing and Electronic Information Management*, 14(3), 21–24, 2024. doi: 10.54097/q1rgkx18.
- [28] Yang, G.-Y., Ko, Y.-H., Wang, F., Yeh, K.-H., Chang, H.-S., Chen, H.-Y. Automated vulnerability detection using deep learning technique. In: *Proceedings of the 30th International Conference on Computational & Experimental Engineering and Sciences (ICCES)*, 2024. doi: 10.48550/arXiv.2410.21968.
- [29] Wang, Z., Guoming, L., Xu, H., You, S., Ma, H., Wang, H. Deep learning-based methodology for vulnerability detection in smart contracts. *PeerJ Computer Science*, 10, e2320, 2024. doi: 10.7717/peerj-cs.2320.

- [30] Bajantri, G., Shariff, C. N. Software vulnerability detection using SFDMN deep learning model. *International Journal of Ad Hoc and Ubiquitous Computing*, 47(4), 227–239, 2024. doi: 10.1504/IJAHUC.2024.142719.
- [31] Savenko, O., Nicheporuk, A., Hurman, I., Lysenko, S. Dynamic signature-based malware detection technique based on API call tracing. *CEUR-WS*, 2393, 633–643, 2019.
- [32] Pomorova, O., Savenko, O., Lysenko, S., Kryshchuk, A. Multi-agent based approach for botnet detection in a corporate area network using fuzzy logic. *Communications in Computer and Information Science*, 370, 243–254, 2013.
- [33] Lysenko, S., Bobrovnikova, K., Shchuka, R., Savenko, O. A cyberattacks detection technique based on evolutionary algorithms. In: *Proceedings of the 11th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 1, 127–132, 2020.
- [34] Lysenko, S., Bobrovnikova, K., Matiukh, S., Hurman, I., Savenko, O. Detection of the botnets' low-rate DDoS attacks based on self-similarity. *International Journal of Electrical and Computer Engineering*, 10(4), 3651–3659, 2020.
- [35] Lysenko, S., Bobrovnikova, K., Savenko, O. A botnet detection approach based on the clonal selection algorithm. In: *Proceedings of the 2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DeSSERT)*, 424–428, 2018.
- [36] Lysenko, S., Savenko, O., Bobrovnikova, K. DDoS botnet detection technique based on the use of the semi-supervised fuzzy c-means clustering. *CEUR-WS*, 2104, 688–695, 2018.
- [37] Denysiuk, D., Savenko, O., Lysenko, S., Savenko, B., Kashtalian, A. Method for detecting steganographic changes in images using machine learning. In: *Proceedings of the 13th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, 1–6, 2023. doi: 10.1109/DESSERT61349.2023.10416453.
- [38] Savenko, O., Sierhieiev, Y., Gaj, P., Balej, J. Using artificial intelligence in the context of buffer overflow vulnerabilities. In: *Proceedings of the 2nd International Workshop on Intelligent & CyberPhysical Systems (ICyberPhyS)*, 211–220, 2025. URL: <https://ceur-ws.org/Vol-4013/paper17.pdf>
- [39] Smith, A., Jones, B., Brown, C. Machine Learning-Based Network Anomaly Detection: Design, Implementation, and Evaluation. *AI*, 5(4), 143–158, 2024. doi: 10.3390/ai5040143.
- [40] Doe, P., Zhang, M. Impact of Machine Learning on Intrusion Detection Systems for the Protection of Critical Infrastructure. *Information*, 12(3), 118–135, 2021. doi: 10.3390/info12030118.
- [41] Li, H., Wang, J., Chen, Q. Deep Learning Approaches for Intrusion Detection Systems: A Survey. *Sensors*, 20(10), 2973–2995, 2020. doi: 10.3390/s200102973.
- [42] Khan, E., Ahmed, S. An Ensemble Deep Learning Approach for Cyberattack Detection in Cloud Computing. *IEEE Access*, 7, 177614–177626, 2019. doi: 10.1109/ACCESS.2019.2934619.
- [43] Guo, J., Li, X., Wu, Y. Malware Detection Using Convolutional Neural Networks and Transfer Learning. *IEEE Transactions on Parallel and Distributed Systems*, 31(8), 1913–1925, 2020. doi: 10.1109/TPDS.2020.2988713.
- [44] Singh, N., Sharma, K., Patel, P. Hybrid Artificial Intelligence System for DDoS Attack Detection. *Applied Sciences*, 12(2), 739–754, 2022. doi: 10.3390/app12020739.
- [45] Rao, S., Selvakumar, V. A Survey on Machine Learning and Deep Learning Techniques for Cybersecurity. *IEEE Access*, 9, 110327–110359, 2021. doi: 10.1109/ACCESS.2021.3050194.