Machine Learning with Applications 22 (2025) 100759

Contents lists available at ScienceDirect

Machine Learning with Applications

journal homepage: www.elsevier.com/locate/mlwa

A machine learning approach to vulnerability detection combining software metrics and topic modelling: Evidence from smart contracts

Giacomo Ibba [a,b], Rumyana Neykova [b], Marco Ortu [a], Roberto Tonelli [a], Steve Counsell [b], Giuseppe Destefanis [c,*]

[a] University of Cagliari, Cagliari, Italy
[b] Brunel University of London, London, United Kingdom
[c] University College London, London, United Kingdom

ARTICLE INFO

Keywords:
Vulnerability detection
Software metrics
Topic modelling
Machine learning
Source code analysis
Smart contracts

ABSTRACT

This paper introduces a methodology for software vulnerability detection that combines structural and semantic analysis through software metrics and topic modelling. We evaluate the approach using smart contracts as a case study, focusing on their structural properties and the presence of known security vulnerabilities. We identify the most relevant metrics for vulnerability detection, evaluate multiple machine learning classifiers for both binary and multi-label classification, and improve classification performance by integrating topic modelling techniques.

Our analysis shows that metrics such as cyclomatic complexity, nesting depth, and function calls are strongly associated with vulnerability presence. Using these metrics, the Random Forest classifier achieved strong performance in binary classification (AUC: 0.982, accuracy: 0.977, F1-score: 0.808) and multi-label classification (AUC: 0.951, accuracy: 0.729, F1-score: 0.839). The addition of topic modelling using Non-Negative Matrix Factorisation further improved results, increasing the F1-score to 0.881. The evaluation is conducted on Ethereum smart contracts written in Solidity.

1. Introduction

Understanding how structural and semantic properties of code relate to security vulnerabilities remains a challenge in software analysis. While software metrics have long been used to support defect prediction in conventional systems (Okutan & Yıldız, 2014; Singh & Chug, 2017; Singh et al., 2010), their role in identifying security-related issues is less clear, particularly when applied to newer software artefacts. This paper introduces a methodology that combines metrics-based analysis with topic modelling to improve the detection and classification of software vulnerabilities.

We evaluate this approach in the context of *smart contracts*, which are programs deployed on a blockchain that execute automatically when predefined conditions are met. Like conventional software components, they are written in programming languages such as Solidity and can be analysed through structural metrics. They differ from traditional software in that they operate in a decentralised environment where code directly manages financial assets, and once deployed they cannot be updated through standard release cycles. These characteristics increase the impact of vulnerabilities, since flaws may lead to immediate and irreversible financial losses (Atzei et al., 2017; Aufiero et al., 2024; Zheng et al., 2018). This connection highlights why techniques from software engineering, such as the use of metrics and semantic analysis, are applicable to smart contracts while also requiring adaptation to address their specific execution and risk environment.

Although metrics such as complexity, coupling, and cohesion are widely studied in traditional systems (Chidamber & Kemerer, 1994; Zhang et al., 2007a), their effectiveness in smart contracts is still uncertain. Preliminary studies focusing on metric-based analysis of smart contracts are limited (Tonelli et al., 2023), and their connection to security has not been examined in detail (Destefanis et al., 2018; Pinna et al., 2019). Moreover, the potential benefit of incorporating semantic information, such as lexical patterns or latent topics (Ortu et al., 2022), remains largely unexplored.

This paper presents the first large-scale study on smart contracts' vulnerabilities prediction and classification combining software metrics and topic modelling. The evaluation is conducted exclusively on Ethereum smart contracts written in Solidity. Whilst the general principles of combining structural and semantic analysis may inform approaches for other programming languages, our empirical findings and performance metrics apply only to the Solidity smart contract domain.

* Corresponding author.
E-mail addresses: giacomof.ibba@unica.it (G. Ibba), rumyana.neykova@brunel.ac.uk (R. Neykova), marco.ortu@unica.it (M. Ortu), roberto.tonelli@unica.it (R. Tonelli), steve.counsell@brunel.ac.uk (S. Counsell), g.destefanis@ucl.ac.uk (G. Destefanis).

https://doi.org/10.1016/j.mlwa.2025.100759
Received 21 June 2025; Received in revised form 7 October 2025; Accepted 12 October 2025
Available online 17 October 2025

Specifically, we address the following research questions:

RQ1: *Which traditional software metrics contribute the most to vulnerability detection in smart contracts?*

RQ2: *How effective are standard classifiers in predicting and classifying vulnerabilities based on software metrics alone?*

RQ3: *Can topic modelling techniques improve classifier performance for vulnerability classification tasks?*

We evaluate the role of software metrics in vulnerability prediction using five widely adopted classifiers: Random Forest, Support Vector Machine (SVM), Gradient Boosting, Logistic Regression, and Multi-layer Perceptron (MLP). We adopt a two-step process to first detect vulnerable contracts and then classify the specific types of vulnerabilities. Our methodology integrates topic modelling techniques — Latent Dirichlet Allocation (LDA) and Non-negative Matrix Factorisation (NMF) — to include semantic features alongside structural ones. Treating each contract as a *document* and its associated metrics and vulnerabilities as *words*, we extract latent patterns that relate to specific vulnerability types. To our knowledge, this is the first study to quantify the predictive value of software metrics within smart contract domain and to combine them with topic modelling for vulnerability classification.

The integration of structural metrics with semantic analysis addresses a limitation in vulnerability detection: individual approaches capture only partial aspects of security-relevant code features. As we demonstrate in Section 3.2, vulnerabilities manifest through specific combinations of architectural properties (measurable via metrics) and implementation patterns (detectable through semantic analysis). For example, reentrancy vulnerabilities require both external coupling (structural) and specific call sequences involving financial transfers (semantic).

Vulnerabilities in smart contracts manifest through predictable combinations of structural and semantic patterns. Reentrancy attacks require both architectural conditions (external calls interacting with state variables, measurable through coupling metrics) and implementation features (specific function call sequences and state modification patterns, detectable through semantic analysis). Access control vulnerabilities combine structural indicators (function visibility and modifier usage) with semantic signals (authentication-related naming patterns and permission checking logic).

Topic modelling enables identification of latent semantic themes that reflect common programming constructs, design patterns, and anti-patterns within smart contract code. These themes capture developer intent and implementation approaches that correlate with vulnerability issues. Contracts exhibiting topic distributions associated with financial transfer operations may correlate with transfer-related vulnerabilities, while contracts showing patterns related to external interactions may indicate reentrancy exposures.

The main contributions of this paper are as follows:

1. A method for classifying vulnerabilities using both software metrics and topic modelling, based on source code analysis and metric extraction.
2. An assessment of software metrics' effectiveness for vulnerability prediction, identifying those with higher predictive value (e.g., *Cyclomatic Complexity, Nesting Depth, Function Calls, Local Variable Count, Coupling Between Contracts*) and those with limited utility (e.g., *Fan-In, Inheritance Depth*).
3. An evaluation of five classifiers for vulnerability detection using metrics, with Random Forest achieving the best results in both binary and multi-label classification. In addition, we deliberately evaluated multiple classifiers representing different modelling paradigms to reduce model selection bias and to ensure that our findings are not specific to a single learning approach.
4. A topic-modelling-based classification approach combining LDA and NMF with software metrics, improving accuracy by compared to metric-only models and including improvements for complex vulnerabilities such as *reentrancy-eth*, *unused-return*, and *tx-origin*.

This paper is organised as follows: Section 2 discusses related work, Section 3 introduces the methodology; Section 4 assesses the role of software metrics; Section 5 evaluates classifier performance. Section 6 presents multi-label classification results; Section 7 focuses on classification using topic modelling. Section 8 outlines limitations, Section 9 discusses future research directions, and Section 10 concludes. The replication package is available at this link.[1]

2. Related work

We review alternative approaches to vulnerability detection in smart contracts, focusing on how they differ from our combined metrics and topic modelling methodology.

Static analysis tools. Traditional tools such as KEVM (Hildenbrandt et al., 2018), Oyente (Luu et al., 2016), and ContractFuzzer (Jiang et al., 2018) rely on formal verification, symbolic execution, or fuzzing. In contrast to our data-driven approach, these tools depend on predefined vulnerability patterns and cannot adapt to variations in how known vulnerabilities manifest across different contracts. While they achieve high precision for known issues, they do not learn from code structure. We do not include them in our experimental comparison, since their goal involves pattern detection through program analysis rather than statistical learning.

Software metrics for vulnerability detection. Zhang et al. (2007b) showed that metrics such as LOC and Cyclomatic Complexity predict defects in traditional software. Although we adopt similar metrics, smart contracts present distinct challenges due to immutability and blockchain-specific behaviours such as reentrancy, which require adapted interpretations.

VCCFinder (Perl et al., 2015) combined software metrics with repository metadata to predict vulnerable components, achieving strong results in traditional systems. However, their reliance on commit history is not applicable to our setting, where smart contracts are often deployed without subsequent updates.

Medeiros et al. (2017, 2020) reported 93.59 percent accuracy using metrics alone. Our approach extends this line of work by integrating semantic features through topic modelling, which improves performance (97.7 percent accuracy) and reduces false positives.

ML approaches for smart contract vulnerabilities. Recent machine learning efforts for vulnerability detection fall into two broad categories:

Opcode- or N-gram-based Methods. Song et al. (2019) and ContractWard (Wang et al., 2021) use n-gram or bigram representations of EVM bytecode and achieve high binary classification accuracy. In contrast to our metric-based approach, these methods represent code as token sequences and do not capture architectural features such as coupling or nesting depth. Moreover, they target only a small number of vulnerability types, typically between three and five, whereas our model handles thirty-three.

Graph-based Methods. Han et al. (2022) apply graph neural networks to smart contract structures. Although they also aim to capture structural relationships, their approach involves costly graph construction and is limited to a few vulnerability classes. In comparison, our method is computationally efficient and generalises to a wider set of vulnerabilities.

Pattern-specific Detectors. SCScan Hao et al. (2020) and Lou et al. (2020) focus on identifying specific vulnerabilities such as general security flaws or Ponzi schemes. These tools are not intended for general-purpose vulnerability detection and cannot scale beyond their predefined scope.

Recent advances in code representation learning, including transformer-based models such as (Feng et al., 2020) and Graph Neural

[1] https://figshare.com/s/5d0129e78d0cf0c61274

Networks, offer alternative approaches for capturing code semantics. These methods can address some limitations of traditional topic modelling by maintaining syntactic awareness and capturing long-range dependencies in code. However, they require substantial computational resources and large-scale pre-training datasets, making them less accessible for many research contexts and limiting reproducibility. Our metrics-based approach with topic modelling provides a computationally efficient alternative that maintains interpretability while delivering measurable performance improvements across several vulnerability types.

Transformer and graph neural network approaches. Recent advances in code representation learning, including transformer-based models such as CodeBERT (Feng et al., 2020) and GraphCodeBERT, offer alternative approaches for capturing code semantics. These models address some limitations of traditional topic modelling by maintaining syntactic awareness and capturing long-range dependencies in code through self-attention mechanisms. Graph Neural Networks (GNNs) applied to control flow or data flow graphs provide another promising direction, as demonstrated by Han et al. (2022) for smart contract analysis.

However, these approaches require substantial computational resources and large-scale pre-training datasets, limiting accessibility for many research contexts and reducing reproducibility. Transformer-based models like CodeBERT require pre-training on millions of code samples and significant GPU resources. GNN-based approaches necessitate costly graph construction from abstract syntax trees or program dependence graphs, with Han et al. (2022) reporting their approach was limited to analysing only a few vulnerability classes due to computational constraints.

Our metrics-based approach with topic modelling provides a more computationally efficient alternative compared to transformer and GNN approaches, maintains interpretability whilst delivering measurable performance improvements across 33 vulnerability types. The two paradigms are complementary rather than competing: transformer models excel at capturing contextual semantics within individual functions, whilst our approach identifies architectural patterns across entire contracts. Metric-based features provide interpretable indicators of structural risk factors that remain accessible to security auditors, whereas deep learning embeddings offer limited explainability. Future work combining our interpretable structural analysis with transformer-based semantic embeddings could potentially achieve superior performance whilst maintaining the transparency required for security-critical applications.

Key differentiators. To our knowledge, no prior work combines software metrics with topic modelling for smart contract vulnerability detection. Existing approaches either apply metrics without capturing semantics (Medeiros et al., 2020), use sequence-based models without structural context (Song et al., 2019; Wang et al., 2021), or focus on a narrow set of vulnerability types (Han et al., 2022; Lou et al., 2020). Our hybrid method combines structural information from metrics with semantic information from topic distributions, enabling efficient classification across thirty-three vulnerability types. The inclusion of semantic features is particularly effective for detecting complex cases, such as reentrancy, that are difficult to capture through metrics alone.

3. Methodology

3.1. Problem formulation

Let $C = \{c_1, c_2, \ldots, c_n\}$ denote a set of $n = 74{,}225$ smart contracts. For each contract c_i, we extract a feature vector $x_i \in \mathbb{R}^d$ containing $d = 8$ software metrics: fan-out, cyclomatic complexity, nesting depth, function calls, local variable count, coupling between contracts, average local variables, and number of raw lines.

We address the following classification tasks:

Binary classification: Learn a function $f : \mathbb{R}^d \to \{0, 1\}$ that predicts whether a contract contains any medium- or high-severity vulnerability, where $y_i = 1$ indicates vulnerable and $y_i = 0$ indicates non-vulnerable.

Multi-label classification: Learn a function $g : \mathbb{R}^d \to \{0, 1\}^{33}$ that predicts the presence of $m = 33$ specific vulnerability types, where $y_{ij} = 1$ if contract c_i contains vulnerability v_j.

Enhanced multi-label classification: To improve multi-label performance, we extract topic distributions $t_i \in \mathbb{R}^k$ from contract source code, where $k = 25$ for LDA or $k = 45$ for NMF. We then learn an enhanced function $g' : \mathbb{R}^{d+k} \to \{0, 1\}^{33}$ using the augmented feature representation $z_i = [x_i, t_i]$.

The main distinction is that topic modelling is applied only in the enhanced multi-label setting. Binary classification uses structural metrics alone. Our evaluation assesses the model's ability to generalise to previously unseen patterns affected by the 33 vulnerability types present in our dataset, following standard machine learning evaluation protocols.

The remainder of this section describes the data collection, feature extraction, modelling steps, and evaluation protocol, following the workflow illustrated in Fig. 1. Our methodology addresses both binary classification (vulnerable vs. non-vulnerable) and multi-label classification (identifying specific vulnerability types). The main distinction between the two tasks lies in the use of topic modelling, which is applied only to the multi-label classification.

We use the latest update[2] from dataset provided by Ibba et al. (2024, 2024c), from which we selected a random sample of 74,225 contracts. This is a diverse dataset, including vulnerability reports generated by Slither (Feist et al., 2019) (a leading static analysis tool, which categorises vulnerabilities as low, medium, or high[3]). The dataset encompasses contracts with a temporal distribution spanning from 2018 to July 2023, when Smart Sanctuary's last update occurred, providing a wide array of different Solidity pragma versions. The wide temporal span ensures the presence of different smart contract design patterns, ranging from several outdated functionalities (e.g., Crowdsale, Initial Coin Offering, and Gambling systems) to complex contracts employed for financial tasks and trading control.

Whilst Slither represents one of the most widely adopted static analysis tools in smart contract security research, its detection capabilities vary substantially by vulnerability type. Comparative benchmarking by Durieux et al. (2020) found that Slither detected 17% of known vulnerabilities across nine categories, with performance ranging from 88% detection for reentrancy vulnerabilities to 0% for arithmetic vulnerabilities. When applied to 47,518 real contracts, static analysis tools collectively flagged 93% as vulnerable, indicating potential false positive issues, though precise false positive rates for specific tools remain unquantified in the literature. Despite these limitations, Slither provides a consistent and reproducible labelling framework for large-scale empirical studies, and our focus on medium and high-severity classifications reduces the impact of tool-specific detection biases.

We considered as vulnerable only contracts that have at least one vulnerability labelled by Slither as medium (resulting in 5377 contracts) or high (resulting in 1759 contracts), both high and medium (resulting in 3957). In total, we have 11,093 vulnerable contracts, and 59,310 non-vulnerable. These vulnerabilities can lead to significant financial losses or operational impacts, based on Slither's severity classification. We excluded low-severity vulnerabilities as they primarily represent code quality recommendations (style suggestions, version update advisories, optimisation recommendations, advisories for missing events) rather than actual security threats. This filtering aligns with standard smart contract security assessment practices that prioritise vulnerabilities with potential for exploitation and financial impact.

[2] https://github.com/giacomofi/SmarthER

[3] https://github.com/crytic/slither/wiki/Detector-Documentation

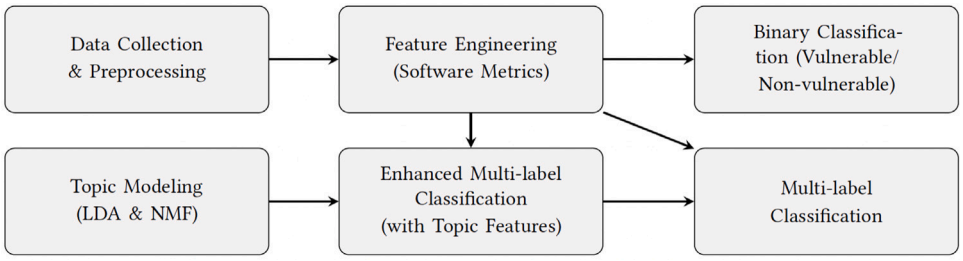


Fig. 1. Workflow for smart contract vulnerability detection and classification.

Table 1
Key software metrics used in smart contract analysis.

Metric	Description
Lines of Code (LOC)	Total No. of code lines in the smart contract
No. Of Contracts	No. of contracts defined within the smart contract.
Local Variable Count	No. of local variables within a function.
Max Local Variables	Maximum No. of local variables across all functions in a contract.
Avg. Variable Count	Average No. of local variables in all functions in a contract.
Number of Parameters	No. of parameters provided as input to functions.
Function Calls	No. of calls for a specific function within the contract.
Cyclomatic Complexity (CC)	No. of linearly independent paths through a function's source code
Inheritance Depth (ID)	the layers of inheritance of a specific contract.
Nested depth (ND)	the depth of nested loops and conditionals within a function
Coupling Between Contracts (CBC)	No. of other contracts or libraries that a contract interacts with
State Variable Count (SVC)	No. of state variables in a contract
Function Count	No. of functions in a contract
Fan-In (FI) and Fan-out (FO)	No. of functions that call (FI) or are called by (FO) a function

Including low-severity classifications would conflate code quality issues with security vulnerabilities, introducing noise into our detection task. This resulted in a total of 33 vulnerabilities, available for reference in our replication package.

In addition to vulnerabilities, the dataset offers a rich set of software metrics at both the contract and function levels, including metrics such as inheritance depth, coupling between contracts, state variable count, and cyclomatic complexity. The full table of supported metrics are available in Table 1. To enhance the existing metrics, we introduced two additional structural metrics, *fan-in (FI)* and *fan-out (FO)*, which respectively measure the number of functions that call or are called by a given function. These metrics offer insights into function call relationships within smart contracts. To extract these new metrics, we employed *MindTheDApp* (Ibba et al., 2024), a static analysis tool that constructs call graphs by traversing the abstract syntax trees (ASTs) of interconnected contracts.

Our experimental evaluation consisted of four main phases, each addressing a different aspect of vulnerability detection and classification:

Feature Selection: We employed a two-stage hybrid feature selection approach combining statistical selection with domain knowledge validation. The primary selection mechanism used Adaptive LASSO regression, which extends standard LASSO by applying individual penalty weights to each coefficient based on an initial estimator. This approach addresses potential instabilities in feature selection that can arise from correlated predictors whilst maintaining the coefficient shrinkage properties that enable automatic feature selection. Recognising that purely algorithmic selection may exclude features with established theoretical importance in software engineering contexts, we implemented a secondary validation stage. Features excluded by Adaptive LASSO underwent evaluation against defect prediction literature to identify metrics with documented predictive value.

Binary Classification: To distinguish between vulnerable and non-vulnerable contracts, we employed multiple classifiers: Logistic Regression, Random Forest, Support Vector Machines (SVM), Gradient Boosting, and Multi-Layer Perceptron (MLP). Synthetic Minority Oversampling Technique (SMOTE) was applied to address class imbalance.

Multi-label Classification: For identifying specific vulnerability types, we evaluate the same set of classifiers as for binary; and applied class weighting to handle class imbalance.

Topic Modelling for Multi-label Classification: To enhance the multi-label classification, we incorporated topic modelling techniques. Latent Dirichlet Allocation (LDA) and Non-negative Matrix Factorisation (NMF) were applied to extract latent topics. The derived topic distributions were combined with software metrics as additional features for classification.

We evaluated the models using 10-fold stratified cross-validation and assessed using several metrics: Area Under the Curve (AUC), Accuracy, Precision, Recall, and F1-Score.

For our classification tasks, we considered five learners: Logistic Regression, Random Forest, Support Vector Machine (SVM), Gradient Boosting, and Multi-Layer Perceptron (MLP). These represent distinct modelling approaches: linear models (Logistic Regression), ensemble tree methods that reduce variance or bias (Random Forest reduces variance through bootstrap aggregation and Gradient Boosting reduces bias via sequential error correction), kernel-based classification (SVM), and neural networks (MLP). A detailed discussion of their characteristics and hyperparameter tuning is provided in Sections 5–5.2.

3.2. Rationale for combining structural metrics with semantic analysis

The integration of structural metrics with semantic analysis addresses a fundamental limitation in vulnerability detection: individual approaches capture only partial aspects of security-relevant code features. Structural metrics quantify architectural properties such as complexity and coupling, which correlate with defect probability, but cannot distinguish between different types of vulnerable implementations. Semantic analysis can identify implementation patterns and naming conventions that indicate security-relevant constructs, but lacks awareness of broader architectural context.

Vulnerabilities in smart contracts manifest through predictable combinations of structural and semantic patterns. We illustrate this with three examples.

Reentrancy vulnerabilities. require both architectural conditions and implementation features. Architecturally, reentrancy occurs when external calls interact with state variables, measurable through coupling metrics (Coupling Between Contracts) and function call counts. Semantically, reentrancy involves specific function call sequences and state modification patterns. Consider this vulnerable pattern:

Listing 1: Reentrancy vulnerability example

```
function withdraw(uint amount) public {
  require(balances[msg.sender] >= amount);
  msg.sender.call{value: amount}(""); // external call
  balances[msg.sender] -= amount; // state change after call
}
```

Structural metrics capture high cyclomatic complexity, external coupling, and multiple function calls. Topic modelling captures the semantic pattern through terms like "call", "send", "balance", "transfer", and "eth". Neither approach alone distinguishes this vulnerable pattern from secure implementations that perform state updates before external calls. The combination identifies contracts with both high coupling (structural signal) and financial transfer semantics (semantic signal).

Access control vulnerabilities. combine structural indicators with semantic signals. Structurally, these vulnerabilities appear in functions with specific visibility modifiers and parameter counts. Semantically, they involve authentication-related naming patterns and permission checking logic. The tx.origin vulnerability demonstrates this:

Listing 2: Access control vulnerability example

```
function transferOwnership(address newOwner) public {
  require(tx.origin == owner); // vulnerable: uses tx.origin
  owner = newOwner;
}
```

Metrics capture function visibility and parameter structure. Topics capture terms like "owner", "require", "tx", "origin", and "msg.sender". Secure implementations use msg.sender instead of tx.origin, producing different topic distributions whilst maintaining similar structural properties.

Weak randomness vulnerabilities. depend on predictable data sources for pseudorandom number generation. These vulnerabilities have minimal structural signatures but strong semantic patterns:

Listing 3: Weak randomness generation example

```
function random() internal view returns (uint) {
  return uint(keccak256(abi.encodePacked(
    block.timestamp, block.difficulty)));
}
```

Structural metrics show standard function complexity. Topic modelling identifies the problematic semantic pattern through terms like "timestamp", "block", "difficulty", "hash", and "random". Contracts using external oracles for randomness produce different topic distributions containing terms like "oracle", "chainlink", or "vrf".

Topic modelling enables identification of latent semantic themes that reflect common programming constructs, design patterns, and anti-patterns within smart contract code. These themes capture developer intent and implementation approaches that correlate with vulnerability patterns. Contracts exhibiting topic distributions associated with financial transfer operations correlate with transfer-related vulnerabilities, whilst contracts showing patterns related to external interactions indicate potential reentrancy exposures.

This mechanistic understanding justifies our hybrid approach: structural metrics identify architectural risk factors, whilst topic distributions identify the specific implementation patterns that transform architectural risk into actual vulnerabilities.

3.3. Relationship between vulnerabilities and software metrics

Our analysis reveals specific correlations between software metrics and vulnerability patterns. State management metrics (State Variable Count and Local Variable Count) show strong associations with initialisation and shadowing vulnerabilities, while Function Count and Parameter numbers serve as indicators for unused-return and unchecked-transfer issues.

Cyclomatic Complexity proves particularly useful for identifying control flow vulnerabilities in functions with multiple execution paths. Inheritance Depth correlates with constructor and token-related vulnerabilities, while Coupling Between Contracts (CBC) indicates potential reentrancy and transfer-related issues.

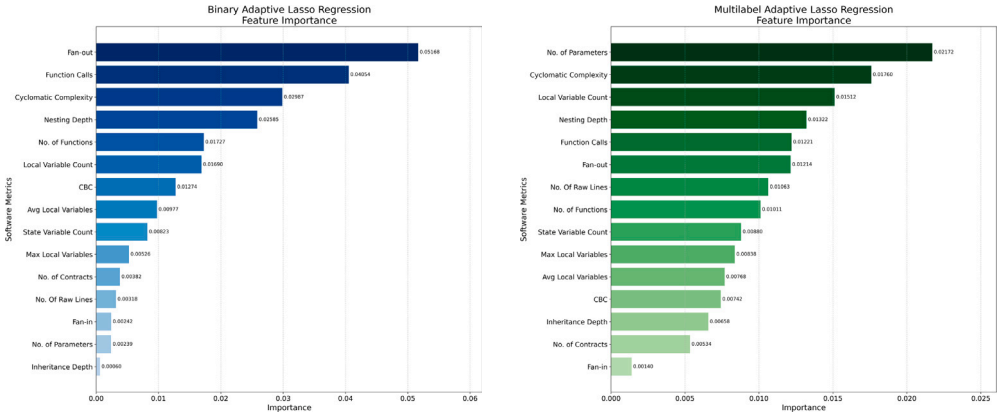
Nested Depth (ND) emerges as a key indicator for tx-origin and msg-value-loop vulnerabilities, especially in complex control structures. When combined with high Cyclomatic Complexity or numerous state variables, increased Nesting Depth often signals difficult-to-verify state transitions.

The effectiveness of these metrics varies by vulnerability type, suggesting the importance of a multi-metric approach. This is particularly evident in detecting complex vulnerabilities like reentrancy attacks, which typically manifest through patterns across multiple metrics rather than through individual indicators.

4. Metrics and their contribution to vulnerability detection

Feature selection in vulnerability prediction requires balancing statistical relevance with domain expertise, as purely data-driven approaches may overlook theoretically important predictors. We implemented a systematic two-stage hybrid feature selection methodology that combines statistical selection with literature-informed validation. We employed Adaptive LASSO as our primary feature selection technique. Adaptive LASSO extends standard LASSO regression by applying individual penalty weights to each coefficient, derived from an initial estimator, to improve feature selection consistency. To optimise the configuration, we performed a grid search across various combinations of alpha (regularisation strength) and gamma (adaptive weights exponent) parameters. We tested alpha values ranging from 10^{-4} to 10^{4} (20 logarithmically spaced values) combined with gamma values of [0.5, 1, 2], resulting in 60 parameter combinations evaluated through 10-fold cross-validation to ensure robust hyperparameter selection.

Justification for hybrid feature selection approach. We selected Adaptive LASSO over alternative methods for several theoretical and practical reasons. Random Forest feature importance, whilst interpretable, relies on mean decrease in impurity scores that can be biased towards high-cardinality features and do not provide the embedded regularisation properties necessary for coefficient shrinkage. Recursive Feature Elimination offers theoretical guarantees similar to LASSO but requires multiple model training iterations as features are eliminated sequentially, resulting in substantially higher computational cost compared to a single LASSO fit, making it impractical for our grid search across 60 hyperparameter combinations. However, purely algorithmic selection risks excluding features with established theoretical importance. Software engineering literature consistently identifies Lines of Code as a fundamental defect predictor (Singh & Chug, 2017; Singh et al., 2010), yet LASSO excluded this metric. We therefore implemented a two-stage validation: statistical selection via Adaptive LASSO followed by literature review of excluded features. We empirically validated this hybrid approach by comparing two configurations: LASSO features alone versus LASSO features plus Number of Raw Lines. The inclusion of



(a) Binary Feature Importance (b) Multilabel Feature Importance

Fig. 2. Comparison of top software metrics features selected using binary and multilabel lasso regression.

Number of Raw Lines significantly improved overall performance, leading to a 10% increase in accuracy, confirming that domain knowledge complements data-driven selection. This hybrid methodology balances statistical rigour with theoretical foundations from defect prediction research.

Fig. 2 shows the feature importance as determined for both binary and multilabel Adaptive LASSO regression. We established a threshold of 0.01 for feature selection to distinguish significant features from those with negligible LASSO coefficients. This threshold was selected because the binary LASSO regression chart shows a natural break in feature importance around the 0.01 mark, with features above this threshold demonstrating substantially higher LASSO coefficients compared to those below. This threshold creates a model that balances complexity (number of features) with predictive power, selecting approximately half of the available features. The consistency of top features selected by the 0.01 threshold across both binary and multilabel classifications suggests robust feature importance patterns across classification types.

Recognising that LASSO's coefficient shrinkage property can exclude features with subtle but documented importance in software engineering contexts, we systematically evaluated whether LASSO-excluded features possessed established theoretical foundations in defect prediction literature. The LASSO statistical selection identified seven significant software metrics: Fan-out, Cyclomatic Complexity, Nesting Depth, Function Calls, Local Variable Count, Coupling Between Contracts (CBC), and Average Local Variables. Following our planned validation stage, we assessed excluded features against established defect prediction research. "Number of Raw Lines" demonstrated strong theoretical justification as a fundamental predictor in software defect studies (Singh & Chug, 2017; Singh et al., 2010), warranting empirical validation.

We conducted comparative analysis using two feature configurations for binary classification: one subset, only including features with LASSO coefficients above 0.01, which excluded *"No. of Raw Lines"*, *"Inheritance Depth"*, *"No. of Parameters"*, *"Fan-In"*, and features related to the number of local and state variables. The second subset, encompasses *"No. of Raw Lines"* along with the features exhibiting a LASSO coefficient above 0.01. Our analysis revealed that including *"No. of Raw Lines"* significantly improved overall performance, leading to a 10% increase in accuracy, aligning with insights from related work on defect prediction in software.

Features such as *"Inheritance Depth"* were excluded based on both low LASSO coefficients and limited empirical support for predictive power in the literature (Okutan & Yıldız, 2014). This hybrid methodology ensures that our feature selection process combines data-driven statistical analysis with domain knowledge, creating a foundation for vulnerability prediction while avoiding the limitations of purely algorithmic or purely expert-driven approaches.

> **Answer to RQ1:** The most significant metrics for detecting vulnerabilities in smart contracts, identified through LASSO include *Fan-out, Cyclomatic Complexity, Nesting Depth, Function Calls, Local Variable Count, Coupling Between Contracts (CBC), and Avg. Local Variables*, which have been extended with *No. of Raw Lines*, according to practical evidence from related work in defect prediction. These metrics are important for understanding the structural and functional aspects that may contribute to vulnerabilities. Conversely, metrics like *Fan-In, Inheritance Depth, Average Local Variables, and Number of Parameters* exhibit a low LASSO coefficient and were discarded from our features set.

5. Binary classification: Software metrics for vulnerability prediction

In this section, we address our second research question (RQ2): *How effective are standard classifiers in predicting vulnerabilities based on software metrics alone?* Building on our previous analysis of software metrics, we explore how traditional classification algorithms perform in identifying vulnerable smart contracts.

5.1. Data preprocessing

In our data preprocessing phase, we made minimal modifications to the initial dataset, which pertains to the pragma version. For entries with missing pragma information, we inserted a placeholder "*0.0.0*" version, while for the remaining contracts, we preserved only the major Solidity version numbers (e.g., 0.5.0 from pragma solidity ^0.5.0). We chose to align with major solidity versions to ensure compatibility with our analysis tool, given that a contract with pragma solidity ^0.5.0 must be compatible with features between versions 0.5.0 and 0.6.0.

5.1.1. Handling class imbalance (SMOTE)

Our dataset showed a significant class imbalance, where vulnerabilities were distributed in 15.7% of the total sample. In our dataset, we have a total of 11,093 vulnerable contracts, vs. 59,310 non vulnerable. This imbalance can lead to biased model training, where the classifier might become more proficient at predicting the majority class while underperforming on the minority class. To mitigate this issue, we applied the Synthetic Minority Over-sampling Technique (SMOTE) (Chawla

et al., 2002), addressing class imbalance by generating synthetic instances for the minority class. SMOTE was applied within the stratified cross-validation process to ensure that each training subset undergoes the same preprocessing steps, leading to more consistent and reliable results. Moreover, this approach ensures that the class imbalance problem is addressed in the training data of each fold without affecting the natural distribution of the test data in that fold.

5.2. Model development

In selecting models for our analysis, we considered a balance between interpretability, complexity, and performance. We chose five standard classification methods: Logistic Regression (Cramer, 2002), Gradient Boosting (Friedman, 2001), Random Forest (Breiman, 2001), Support Vector Machine (SVM) (Boser et al., 1992) and Multi-layer Perceptron (MLP) (Rumelhart et al., 1986), each offering distinct advantages for our task of vulnerability prediction.

Logistic Regression served as an interpretable linear baseline to assess whether the vulnerability prediction problem requires non-linear complexity. SVM was included for its theoretical foundations in high-dimensional classification and effectiveness in distinguishing between classes through kernel methods. MLP represents neural network approaches, capturing hierarchical feature interactions through non-linear transformations to model complex patterns that simpler algorithms might miss. The inclusion of two ensemble methods reflects their fundamentally different learning philosophies. Random Forest employs bootstrap aggregation (bagging), building uncorrelated trees in parallel to reduce variance and resist overfitting. Gradient Boosting uses sequential error correction (boosting), where each tree iteratively reduces bias by correcting predecessor errors. This comparison allows us to determine whether bias reduction or variance reduction strategies are more effective for software vulnerability prediction, providing greater methodological insight than evaluating variants within the same algorithmic family (such as different neural network architectures or SVM kernels); ensemble diversity addresses fundamental aspects of the bias–variance trade-off relevant to software metrics data.

5.2.1. Hyperparameter tuning

To maximise performance of the models, we conducted hyperparameter tuning using a grid search with ten-fold stratified cross-validation. This method systematically explores a range of hyperparameter values to identify the optimal settings for each model, ensuring robust predictions. Stratified cross-validation helps in assessing the model's performance across different subsets of the data, thus preventing overfitting and ensuring that the model generalises well to unseen data. The list of hyperparameters tuned for model evaluation is presented in Table 2.

For Logistic Regression, we explored different regularisation strengths (C), penalty types, and solver algorithms. The wide range of C values (0.001 to 100) allowed us to explore from high regularisation (0.001) to almost no regularisation (100). This helped in understanding how much the model needed to be constrained to prevent overfitting.

Gradient Boosting's grid focused on the number of estimators, learning rate, and tree depth. The learning rates (0.01, 0.1, 0.2) represent a good spread from conservative to more aggressive learning. The *max_depth values* (3, 4, 5) are relatively shallow, which can help prevent overfitting in boosting models.

Random Forest's tuning involved varying the number of trees, maximum depth, and minimum samples required to split an internal node. The inclusion of None in *max_depth* allowed for full tree growth, contrasting with the limited depths of 10 and 20. This can show whether the model benefits from deeper, more complex trees or shallower, more generalisable ones.

For SVM, we adjusted the regularisation parameter (C), loss function, dual formulation, penalty type, tolerance for stopping criteria, and maximum iterations. The inclusion of both "*hinge*" and "*squared_hinge*" loss functions allows for comparison between standard SVM and a variant that penalises violations more strongly. The wide range of C values (0.1 to 100) again explores different trade-offs between margin maximisation and misclassification.

The Multilayer Perceptron grid encompassed different hidden layer configurations, activation functions, and alpha values for L2 regularisation. These hyperparameters collectively control various aspects of model complexity, learning behaviour, and regularisation, allowing us to find the optimal balance between bias and variance for each algorithm on our specific dataset. The hidden layer sizes include both single and double layer configurations, allowing us to assess if the added complexity of a second hidden layer improves performance. The inclusion of both "*relu*" and "*tanh*" activation functions lets us compare a more modern, non-saturating activation ("*ReLU*") with a traditional, bounded one ("*tanh*").

5.3. Model evaluation

We evaluated the models using combined metrics: ROC Area Under the Curve (AUC), Precision, Recall, and F1-Score as per Shepperd et al. (2019). We employed AUC to measure overall discriminative power regardless of threshold choice, Accuracy to measure overall correctness, Precision to measure the correctness of positive predictions, Recall to measure the ability to find all positive instances, and the F1-score to provide a single metric balancing precision and recall. This combined metric evaluation is crucial for vulnerability detection, where both false negatives (missed vulnerabilities) and false positives (wasting developer time) carry significant costs. This evaluation helps us comparing the strengths and weaknesses of each model in predicting vulnerabilities.

We employed an evaluation protocol using stratified train-test splits (80/20) with fixed random seeds to ensure reproducibility. Hyperparameter optimisation was conducted using 10-fold stratified cross-validation applied to the training data, with the held-out test set being separated throughout the model selection process to prevent data leakage. Final performance metrics were computed only on this independent test set after hyperparameter selection was complete.

5.4. Results and discussion of binary classification

Table 3 shows the performance of each model, including AUC, accuracy, precision, recall, and F1-score along with their standard deviations, reflecting both the average performance and consistency across cross-validation folds. Fig. 3(a) shows the ROC curves trend for binary classification models, highlighting that all models provide strong discriminative power, with AUC scores above 0.86. Random Forest achieves the highest AUC (0.982), with its curve indicating excellent true positive rates across all false positive rate thresholds. The steep initial rise of the Random Forest curve demonstrates its ability to identify vulnerable contracts with minimal false positives. Gradient Boosting and MLP show similar performance trajectories with AUC scores of 0.966 and 0.969, respectively. Logistic Regression and SVM exhibit more gradual curves reflecting their lower discriminative power. These ROC curve patterns align with the performance metrics presented in Table 3. Random Forest tuned with None limit of tree's depth, 2 samples for node split, and with 200 number of trees, outperforms the others, achieving the highest values across all metrics, indicating it offers the best balance between detecting vulnerable contracts and minimising false positives.

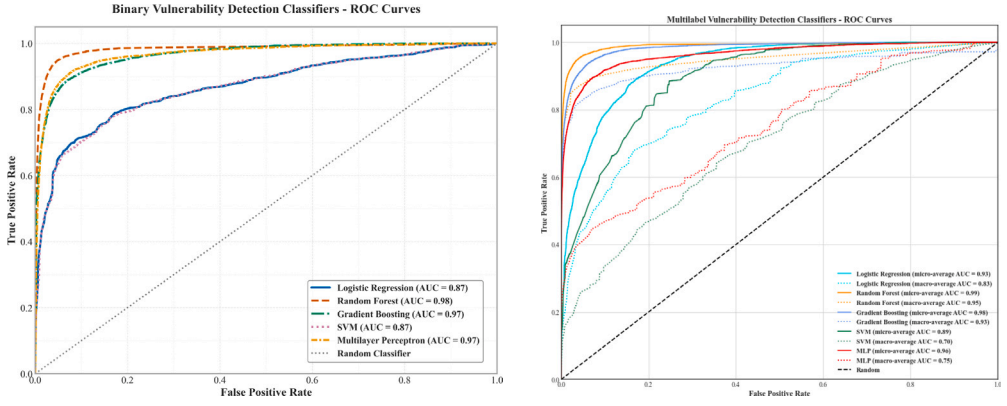
However, most models exhibit low precision, particularly Logistic Regression, SVM, MLP, and Gradient Boosting, which suggests a tendency to overpredict the majority class (non-vulnerable contracts). While this aligns with our priority of maximising recall to ensure vulnerable contracts are not missed, it leads to an increased rate of false positives.

Table 2
Key hyperparameters tuned for each model.

Model	Hyperparameter	Description
Logistic Regression	C	Inverse of regularisation strength; controls trade-off between training error and model complexity.
	Penalty	Type of regularisation: L1 (sparse models), L2 (prevents large coefficients), Elasticnet (combines both).
	Solver	Optimisation algorithm; affects convergence and efficiency.
Gradient Boosting	n_estimators	Number of boosting stages; more can increase capacity but may overfit.
	Learning_rate	Shrinks the contribution of each tree, controlling the learning rate.
	Max_depth	Limits tree complexity; deeper trees may overfit.
	Min_samples_split	Minimum samples for node split; prevents overfitting.
Random Forest	n_estimators	Number of trees; more trees improve performance but increase cost.
	Max_depth	Limits tree depth, balancing complexity and overfitting.
	Min_samples_split	Similar to Gradient Boosting; controls node formation.
Support Vector Machine	C	Regularisation parameter; it controls the trade-off between achieving a low training error and a low testing error that is, the ability to generalise.
	Loss	Loss function; "*squared_hinge*" penalises violations more strongly than regular hinge loss.
	Dual	Determines whether to solve the dual or primal optimisation problem.
	Penalty	Specifies the norm used in the penalisation, with "*l2*" being the standard Euclidean norm.
	Tol	Sets the tolerance for stopping criteria, determining the precision of the solution.
	max_iter	Defines the maximum number of iterations for the solver to converge.
Multilayer Perceptron	hidden_layer_sizes	Specifies the number and size of hidden layers in the neural network, determining the model's complexity and capacity to learn.
	Activation	Defines the non-linear function applied to the weighted sum of inputs at each neuron, affecting how information flows through the network.
	Alpha	Controls the strength of L2 regularisation (weight decay) applied to the weights, helping to prevent overfitting.

Table 3
Performance metrics for binary classification models (Test/Training).

Model	AUC	Accuracy	Precision	Recall	F1-Score
Logistic Regression	0.868	0.879 ± 0.000746 / 0.803	0.281 ± 0.00104 / 0.865	0.720 ± 0.00651 / 0.717	0.404 ± 0.00158 / 0.784
Gradient Boosting	0.966	0.951 ± 0.000904 / 0.954	0.549 ± 0.00621 / 0.565	0.840 ± 0.00321 / 0.854	0.664 ± 0.00462 / 0.680
SVM	0.867	0.893 ± 0.000940 / 0.892	0.307 ± 0.00233 / 0.305	0.695 ± 0.00597 / 0.693	0.426 ± 0.00284 / 0.424
MLP	0.969	0.936 ± 0.00271 / 0.941	0.468 ± 0.0119 / 0.494	0.896 ± 0.00610 / 0.931	0.615 ± 0.00918 / 0.645
Random Forest	0.982	0.977 ± 0.000248 / 0.993	0.780 ± 0.00270 / 0.981	0.837 ± 0.00503 / 0.969	0.808 ± 0.00238 / 0.943



(a) Binary Classification (b) Multilabel Classification

Fig. 3. ROC curves trend for binary and multilabel vulnerability detection classifiers.

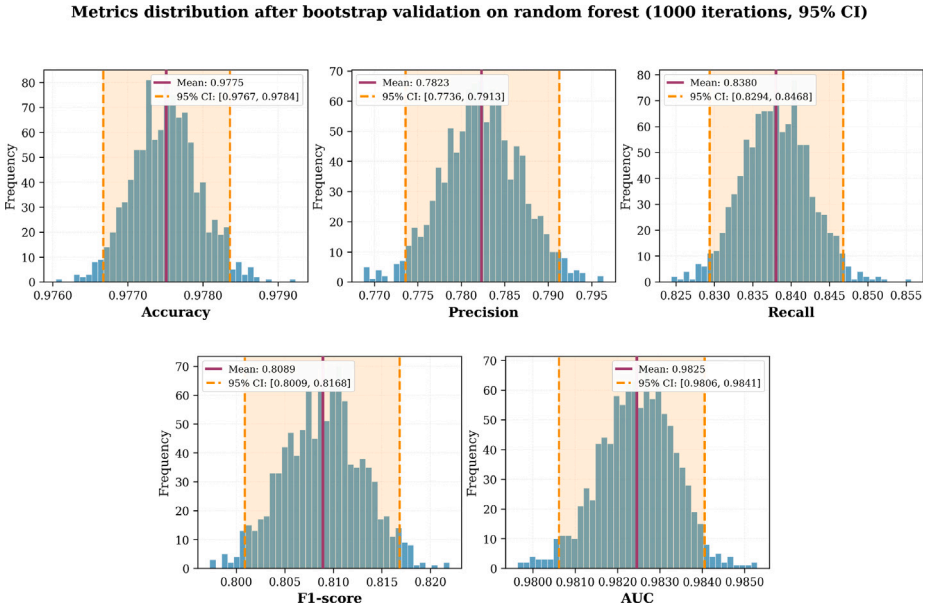


Fig. 4. Metrics distribution after bootstrap validation on random forest for binary classification.

The low standard deviations across all metrics indicate that the models are stable in their performance.

To assess the generalisation capability of our best-performing model (Random Forest), we employed a bootstrap validation procedure (Davison & Hinkley, 1997). This method involved 1000 iterations of resampling with replacement from our test set, followed by model evaluation on each resampled dataset. We computed key performance metrics—accuracy, precision, recall, F1-score, and AUC—for each iteration, enabling us to estimate both the central tendency and variability of model performance.

The resulting distributions, presented in Fig. 4, reveal consistently high performance across all metrics. The model demonstrates stability in accuracy (centred at 0.9775) and AUC (centred at 0.9825), as evidenced by their narrow distributions. While precision and recall show slightly wider distributions, centred at 0.782 and 0.838, respectively, they still indicate robust performance. The F1-score, balancing precision and recall, centres around 0.809. In addition to evaluating performance metrics, we considered the out-of-sample percentage inherent in the bootstrap validation process. Since each bootstrap sample is drawn with replacement, approximately 37% of the test set instances are left out of each resampled dataset on average (often referred as out-of-bag samples) (Efron, 1992). These out-of-bag samples instances provide an additional layer of evaluation for the model, mimicking unseen patterns and further challenging the generalisation capabilities of our Random Forest model. By estimating both the central tendency and variability of the metrics across the 1000 bootstrap iterations, we observed consistent performance and demonstrated the robustness of the model against out-of-bag variation. These results provide statistical evidence of our model's reliable and generalisable performance in identifying vulnerable smart contracts.

To further validate the outcome, we compared our best-performing model with a dummy classifier, which served as a baseline model. Our dummy model was evaluated employing the "*stratified*" strategy, respecting the training set's class distribution, and returned 0.49 in accuracy, 0.05 in precision, 0.49 in recall, and 0.10 in f1-score, and its AUC value floats around 0.49. The Random Forest model significantly outperforms the dummy classifier across all metrics, with higher accuracy (0.9772 vs. 0.4997), precision (0.7808 vs. 0.0564), recall (0.8361 vs. 0.4934), F1-score (0.8075 vs. 0.1013), and AUC (0.9825 vs. 0.4967) demonstrating that the Random Forest model effectively captures predictive patterns in the dataset, far exceeding the baseline performance of random guessing or majority class prediction.

6. Smart contract vulnerability detection using multilabel classification

Our initial approach identified whether a smart contract was vulnerable but did not specify the types of vulnerabilities. To address this gap, we reformulated the experiment as a multi-label classification problem, where each type of vulnerability was treated as a distinct label, allowing for more detailed identification and capturing feature relevance across all vulnerability types while still allowing for label-specific feature selection. Aligning with the binary classification task, we considered exclusively high and medium impacting vulnerabilities, filtering out, information reports, optimisation advice, and low-impact exposures. This filtering process outcome retains 33 different vulnerabilities.

First, we independently applied the principle of adaptive LASSO to each label while maintaining a unified feature space across all labels. We examined the feature importance for each class, observing that "*No. of Parameters*" consistently ranked high across multiple vulnerabilities, suggesting its importance in predicting various vulnerability types. "*Cyclomatic Complexity*" and " *No. of Functions*" also show strong predictive power across many categories. Some features exhibit high importance for specific vulnerabilities while being less significant overall. A significant example is the "*Fan-in*" metric, which ranks highly for "*arbitrary-send-eth*" but shows lower general importance.

Fig. 2(b) shows the aggregated feature importance, providing an overview of predictor relevance. The top five features are "*No. of Parameters*", "*Cyclomatic Complexity*", "*Local Variable Count*", "*Fan-Out*", and "*No. Of Raw Lines*". This confirms our previous findings, suggesting that function complexity and structure are key indicators of vulnerability. "*CBC*" (Coupling Between Contracts), and "*No. of Functions*", are identified as important predictors.

This approach balances general predictive power with vulnerability-specific indicators. The results show the importance of considering multiple metrics in vulnerability analysis, as different code properties contribute variously to different types of vulnerabilities.

6.1. Data preprocessing

In this analysis, we focused only on contracts identified as vulnerable, replacing the column reporting the vulnerability exposing the sample with 33 new columns (one for each vulnerability), each

representing a specific vulnerability type (1 indicating the presence of a specific vulnerability, 0 indicating its absence). This approach provided a more granular understanding of the vulnerabilities present in the contracts.

Given that our dataset encompassed samples exposed to vulnerabilities categorised according to varying degrees of impact, we implemented a strategic filtering process: we retained only those vulnerabilities classified as having high or medium impact, as these represent the most severe potential consequences for smart contracts. This methodological decision aligns with the principle of prioritising security concerns that pose the greatest risk, allowing us to concentrate our predictive efforts on vulnerabilities that could significantly compromise contract integrity, functionality, or user assets.

In our dataset, we also have vulnerabilities that we can classify as rare vulnerabilities. These are the vulnerabilities less represented in our dataset, such as "*msg-value-loop*", "*unchecked-send*", and "*mapping-deletion*", counting respectively 10, 32, and 44 samples. We decided not to exclude these rare vulnerabilities from our dataset for the classification task. This choice was driven by the need to evaluate our models' performance across the entire spectrum of vulnerability types, including those infrequently encountered. By including these rare instances, we aimed to assess the robustness and generalisability of our predictive models in handling imbalanced data characteristics inherent in real-world vulnerability distributions. This approach allows us to observe how model performance varies across common and rare vulnerability types, providing insights into potential limitations and areas for improvement.

We maintained the data preprocessing methods described earlier, including grid search with K-Fold stratified cross-validation for hyperparameter optimisation. We used KFold stratified cross-validation with random splits to assess the model's generalisation to unseen held-out data, and while rare cases may be unevenly distributed across folds, this approach simulates real-world conditions.

Since SMOTE is not directly compatible with multi-label data, we handled class imbalance by assigning weights to classes based on their prevalence to ensure that minority classes were adequately considered during model training, which helps reduce the risk of overfitting and preserves the integrity of the original data distribution.

We kept our original model selection and we expanded our hyperparameter search to include: "*max_features*" for Random Forest and Gradient Boosting, controlling the number of features considered for splitting, which helps manage overfitting; "*subsample*" for Gradient Boosting, determining the fraction of samples used for training each tree, aiding in variance reduction. For Multilayer Perceptron we added the learning rate schedule ("*constant*" vs. "*adaptive*"), which adjusts how quick the model learns. In contrast, "*max_iter*" and "*early_stopping*", balance training thoroughness with overfitting prevention. Finally, we added the "*solver*" choice ("*adam*" vs. "*sgd*"), affecting optimisation strategy and convergence speed. These additions allowed for finer control over model complexity, generalisation capability, and training dynamics, considering the complexity given by the reformulation of the problem as a Multilabel classification task.

6.2. Multi-label classification results

The classifier performances are presented in Table 4, with ROC curves shown in Fig. 3(b). The ROC curves reveal important performance patterns through micro-average (solid lines) and macro-average (dashed lines) comparisons. SVM and MLP showed significant gaps between micro and macro-average AUC, indicating poor performance with rare vulnerabilities. Similarly, Logistic Regression exhibited a 10% gap, suggesting inadequate prediction of less common vulnerabilities. Random Forest emerged as the top performer, showing minimal micro-macro AUC gap and achieving the highest overall metrics (AUC: 0.951, accuracy: 0.729, F1-score: 0.839). Its balanced precision (0.890) and recall (0.839) demonstrate robust vulnerability detection capabilities. While low standard deviations indicate good stability, a 0.2 difference between training and testing accuracy suggests some overfitting, particularly challenging given our 33 distinct vulnerability labels. Gradient Boosting showed strong AUC (0.927) and precision (0.909) but displayed higher variability in accuracy and F1-score. Logistic Regression and SVM, despite high precision, showed limited practical utility due to low recall and accuracy. MLP demonstrated moderate but stable performance across metrics.

The bootstrapping validation shown in Fig. 5 demonstrates the Random Forest model's robustness and generalisability. As discussed in the binary classification results 5.4, the out-of-bag samples provide an additional layer of evaluation, ensuring that the model is continuously evaluated on unseen vulnerable patterns. The narrow distributions across all metrics, particularly the high and stable AUC, indicate that our multilabel classifier performs consistently well across different subsets of the data. The slight discrepancy between precision and recall distributions suggests a potential area for fine-tuning. Adjusting the model to improve recall without significantly sacrificing precision could enhance overall performance, especially for detecting rarer vulnerability types.

Similarly to the binary classification problem, we compared the performances of our best model with a dummy classifier. Again, our best-performing model (Random Forest), outclasses the dummy classifier across all metrics in our multilabel classification task. The Random Forest achieves substantially higher accuracy (0.729 vs. 0.015), precision (0.890 vs. 0.147), recall (0.794 vs. 0.143), F1-score (0.839 vs. 0.145), and AUC (0.9508 vs. 0.5004), and this large gap demonstrates that the Random Forest model effectively captures complex patterns in the dataset, far exceeding the baseline performance of random guessing or stratified prediction strategies.

The Random Forest model demonstrates varied performance across different vulnerability types, as outlined in Table 7. This table presents a subset of results for brevity; the full dataset is available in our replication package.[4] It shows strong performance for common vulnerabilities like "*reentrancy-eth*" (F1-score: 0.945) and "*tx.origin*" (F1-score: 0.895). However, it struggles with rare or complex vulnerabilities such as "*suicidal*", and "*msg-value-loop*", failing to detect these entirely (F1-scores of 0). The model's performance on vulnerabilities like "unchecked-send" (F1-score: 0.2) and "*controlled-array-length*" (F1-score: 0.286) indicates challenges in balancing precision and recall for less frequent issues. High AUC scores (often > 0.95) for most vulnerabilities suggest good overall discriminative power, even when F1-scores are lower. These results highlight the model's effectiveness for common vulnerabilities but also reveal limitations in handling rare or complex cases.

Answer to RQ2: The random forest model demonstrated its significant effectiveness in discriminating between vulnerable and non vulnerable contracts, achieving an AUC score of 0.982, accuracy of 0.977, precision of 0.780, recall of 0.837, and F1-score of 0.808. The random forest model also outclassed the other models in the multilabel classification task, achieving the highest performance metrics, with an AUC of 0.951, accuracy of 0.729, precision of 0.890, recall of 0.794, and an F1-score of 0.839, indicating overall robust predictive capabilities.

7. Topic modelling-based classification for vulnerability detection

Building on the limitations identified in the previous section regarding traditional classifiers, we explored the potential of topic modelling techniques to enhance vulnerability classification in smart contracts.

[4] https://figshare.com/s/5d0129e78d0cf0c61274

Table 4
Performance metrics for multilabel classification models (Test/Training).

Model	AUC	Accuracy	Precision	Recall	F1-Score
Logistic Regression	0.833	0.153 ± 0.00648 / 0.159	0.686 ± 0.0201 / 0.694	0.189 ± 0.00549 / 0.187	0.296 ± 0.00818 / 0.295
Gradient Boosting	0.927	0.606 ± 0.209 / 0.672	0.909 ± 0.206 / 0.953	0.659 ± 0.00772 / 0.717	0.764 ± 0.113 / 0.818
SVM	0.702	0.136 ± 0.00537 / 0.141	0.739 ± 0.0173 / 0.748	0.153 ± 0.00582 / 0.154	0.254 ± 0.00890 / 0.255
MLP	0.748	0.571 ± 0.00772 / 0.598	0.848 ± 0.00789 / 0.886	0.642 ± 0.00900 / 0.662	0.731 ± 0.00634 / 0.758
Random Forest	0.951	0.729 ± 0.00542 / 0.955	0.890 ± 0.00504 / 0.964	0.794 ± 0.00651 / 0.996	0.839 ± 0.00551 / 0.980

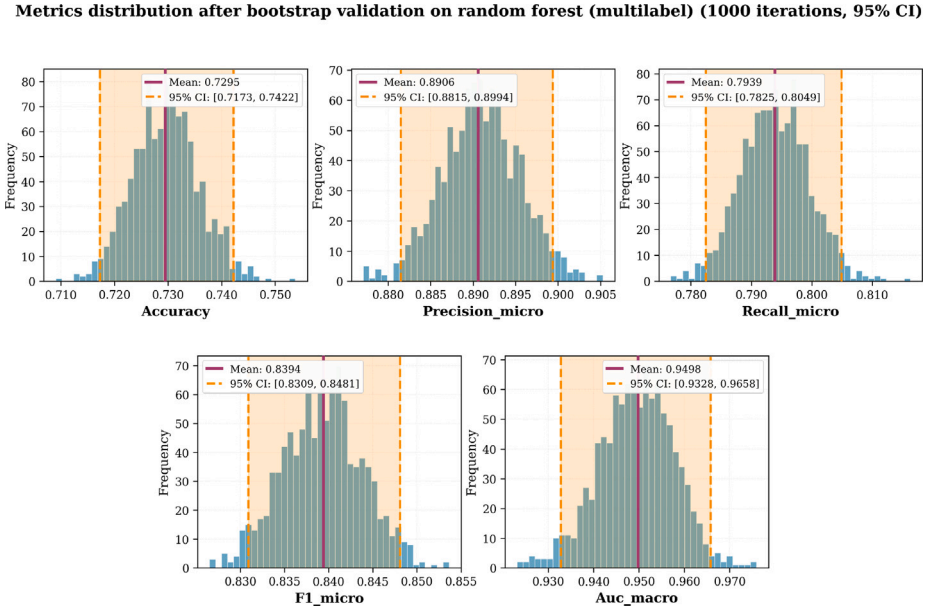


Fig. 5. Metrics distribution after bootstrap validation on random forest for multilabel classification.

Specifically, we employed Latent Dirichlet Allocation (LDA) and Non-Negative Matrix Factorisation (NMF) to introduce a semantic dimension to our analysis.

LDA is a probabilistic model that assumes each document (in this case, each smart contract) is a mixture of topics, with each topic comprising a distribution of words. This model excels at discovering the multi-topic nature of contracts and provides interpretable results through topic probabilities. NMF is a matrix factorisation technique that decomposes the document-term matrix into two non-negative matrices, reflecting document-topic and topic-term associations, and is effective in identifying distinct, non-overlapping components within the data.

Topic distributions can be used as additional features in machine learning classifiers, helping to identify semantic similarities between contracts that may indicate shared vulnerabilities.

7.1. Source code preprocessing

We extract function source code from smart contracts and preprocess it through several steps. The process includes text normalisation by removing whitespace and punctuation, splitting camel case and snake case identifiers, and applying custom tokenisation. This involves filtering tokens by length and excluding those starting with underscores. The preprocessed data was then used to construct a term dictionary and a document-term matrix, capturing the lexical variety and frequency patterns in the smart contract codebase.

7.2. Topic model evaluation

We conducted a grid search to optimise LDA topic modelling using Gensim's LdaMulticore with the following configuration: 100 passes through the corpus, chunk size of 100 documents for batch processing, and asymmetric alpha to allow topics to have different prior weights. The model used 100 iterations for convergence assessment, with random state fixed at 100 for reproducibility. Cross-validation stability was ensured through consistent random seeds across all model configurations. We tested topics from 5 to 45 in 5-step increments and n-grams from 1 to 4, which range was selected to balance granularity with specificity: 15 topics provided a baseline for broad themes, while 45 topics allowed exploration of finer patterns within our 33 distinct vulnerabilities. Our n-gram selection (1 to 4) was based on practical constraints. Topic modelling effectiveness diminishes with higher n-gram sizes due to increased sparsity and reduced generalisability. We chose quadgrams as the upper limit due to our dataset's contextual nature and size—analysing only vulnerable samples' function source code meant larger n-grams would offer minimal benefits while increasing computational costs. We evaluated LDA models using the c_v coherence score, which measures semantic similarity between high-scoring words in topics. By plotting coherence scores against topic counts, we identified optimal configurations balancing detail with interpretability. The model with the highest coherence score was selected for further analysis. For NMF, we employed a parallel evaluation strategy using scikit-learn's TfidfVectorizer, maintaining the same topic and n-gram ranges, and by employing default convergence criteria (tolerance of $1e^{-4}$, maximum 200 iterations). The NMF algorithm employed coordinate descent solver with beta_loss set to "frobenius" for standard squared loss minimisation. We selected the optimal NMF model based on reconstruction error, which measures how accurately the model represents the original data. Lower errors indicated better fit and more precise topic representation. For both approaches, we extracted topic distributions per document to support subsequent analysis. Visualisation of model performance across parameters informed our selection process, optimising for interpretability, coherence, and minimal reconstruction error while maintaining model stability.

Table 5
Selected sample of the extracted topics by the LDA and NMF models.

Model	Top words	Semantic interpretation	Key Vulnerabilities
LDA	send, eth, fee, back, balance	Financial transfer operations	Reentrancy, unchecked sends, ETH transfer issues
LDA	price, order, reward, storage, seller	Market and storage logic	State manipulation, price oracle attacks
LDA	spend, nonce, supplyburn, variable, burninstead	Token supply and burn	Front running, access control
LDA	great, botbotuniswap, wallet, uniswaprouter, seconduniswap	Wallet and DEX operations	Front running, reentrancy, price oracle attacks
LDA	trading, open, liquidityeth, trading, uniswaprouter	Trading control and liquidity	MEV attacks, liquidity manipulation
LDA	tokend, nexttokend, previoustokend, ownership, approval	Token ownership and management	Access control
LDA	time, raise, beneficiary, invest, amount	Crowdsale and ICO mechanics	Timestamp dependence, investment fraud patterns
LDA	shareholder, fee, share, feebuy, feesell	Fee distribution	Fee manipulation
LDA	uniswap, great, takefee, trading, open	Uniswap trading and fees	DEX manipulation, fee manipultation
LDA	result, receive, profit, player, winner	Gaming and gambling logic	Randomness manipulation, unfair game mechanics, timestamp dependence
NMF	tokenamount, unavoidable, slippage, ethamount, liquidity	Slippage and liquidity provision	Slippage attacks, reentrancy
NMF	fees, swapping, selltotalfees, buytotalfees, transfer	Buy/sell fee systems	Fee manipulation, reentrancy
NMF	tokenid, preownershipaddr, address, owner, msgsender	ERC721 ownership tracking	Access control, NFT transfer issues
NMF	path, address, amounttoliquify, amounttoswap, amountethliquidity	DEX operations	Path manipulation, reentrancies
NMF	preownershippacked, balance, tokenid, slot, burned	NFT gas optimisation (ERC721A)	Gas griefing, DoS, access control
NMF	spender, approveaddress, allowedmsgsenderspender, value	ERC20 transfer	Access control, reentrancy
NMF	tokenamount, ethamount, addliquidityuint, local, blocktimestamp	Liquidity timing and MEV	MEV exploitation, time dependency, front running, reentrancy
NMF	percent, frequencyinseconds, enabled, set, percentforlpburn	Automatic liquidity pool burn	Front running, DoS
NMF	rewards, valutapproverewards, strategist, vault, keeper	Yield farming vault	Access control, reentrancy, DoS
NMF	newvotes, ncheckpoints, oldvotes, blocknumber, votes	Voting system	DoS, time dependency bugs, access control

7.3. Topics interpretability and semantic relevance for vulnerability detection

To assess the interpretability and validate their relevance for our vulnerability detection task, we manually examined the topics extracted from both LDA and NMF models through semantic analysis. Both LDA and NMF extracted semantically meaningful topics, that represent several design patterns of smart contract functionalities.

Table 5 highlights a selected sample of topics from both the NMF and LDA models, outlining the semantic coherence between discovered topics and vulnerability classes. The correlation between these topics and specific vulnerabilities reflects the implementation of specific design patterns. Topics including financial transfer operations (“send”, “eth”, “fee”, “transfer”, “balance”) directly correlate with reentrancy vulnerabilities, because such patterns often involve external calls without proper state protection, creating preconditions exploited in attacks like the infamous DAO incident.

Market and storage logic topics (“price”, “order”, “storage”) correlate with oracle manipulation vulnerabilities as they include code structures dependent on external stat that can be artificially influenced through loans or coordinated market actions. Moreover, storage logic topics, could include timing mechanisms that could introduce time-dependency and manipulation vulnerability patterns.

Trading and liquidity management topics align with Maximal Extractable Values (MEV) attacks, since they represent the exact patterns that automated extraction bots target for front-running and sandwich attacks.

The gaming logic topic extracted by the LDA model correlate with randomness manipulation. In fact, blockchain gaming patterns are based on predictable data sources on the chain. ERC721 ownership topics correlate with the access control vulnerabilities, as they capture the authentication patterns where msg.sender versus tx.origin confusion commonly occurs. Similarly, ERC20 management logic topics, align with race condition and access control vulnerabilities since they represent the approve/transferFrom mechanism exposed to front-running vulnerabilities.

Automatic liquidity pool burn mechanisms represent systematic token destruction processes designed to reduce circulating supply and increase token value. These contracts include percentage-based burning logic with time-frequency controls. The semantic patterns captured (“frequencyinseconds”, “enabled”, “percentforlpburn”) correlate with front-running vulnerabilities because attackers can monitor pending burn transactions and execute trades before burn events affect token prices. DoS vulnerabilities may emerge when burn mechanisms consume excessive gas or create bottlenecks in contract execution.

Wallet and Decentralised exchange (DEX) operations related topics encompass DEX interactions, market maker protocols, and external wallet integrations. These patterns involve state transitions across multiple contracts and external calls to swap, add liquidity, and manage token reserves. The correlation with front-running, reentrancy, and price oracle attacks stems from the predictable nature of these operations where attackers can observe pending transactions and manipulate prices or exploit state inconsistencies during multi-step processes.

Yield farming vault systems implement investment strategies where users deposit tokens to earn rewards through various DeFi (decentralised finance) protocols. These contracts manage user deposits, calculate rewards, and execute withdrawal mechanisms involving multiple external protocols. The semantic clustering around vault management correlates with access control vulnerabilities through privileged role management, reentrancy risks from external protocol interactions, and DoS vectors through reward calculation processes.

Voting system patterns capture governance mechanisms where token holders participate in protocol decisions through weighted voting based on token holdings or delegation. These systems track vote counts, manage proposal states, and implement time-based voting periods. The correlation with DoS, time dependency, and access control vulnerabilities arises from the computational demands of vote tallying, timestamp-dependent voting windows, and manipulation of voting power through various attack vectors.

7.4. Qualitative analysis — relationships between vulnerabilities and topics

Our dataset encompasses vulnerabilities with varying degrees of semantic detectability through topic modelling. This analysis examines why certain vulnerabilities correlate with extracted topics while others remain challenging to detect through semantic patterns alone. Reentrancy vulnerabilities demonstrate strong correlation with financial transfer topics. These vulnerabilities involve external calls using specific functions (*“call”*, *“send”*, *“transfer”*) and appear within fee management and liquidity patterns. The semantic clustering of terms like “send”, “transfer”, “call”, and “balance” captures the code constructs where reentrancy occurs. Consider the following vulnerable pattern:

Listing 4: Reentrancy vulnerability example

```
function withdraw(uint amount) public {
  require(balances[msg.sender] >= amount);
  // external call
  msg.sender.call{value: amount}("");
  // state change after call
  balances[msg.sender] -= amount;
}
```

The code snippet 4 exhibits the reentrancy vulnerability through a violation of the check-effects interaction pattern: the state update occurs after the external call. The *call* function forwards all available

gas to the recipient, enabling an attacker to recursively invoke *withdraw* before the balance update completes, thereby draining the contract. The topic model identifies this pattern through the co-occurrence of financial terms ("call", "eth", "send") with balance management terms ("balance", "amount", "fee"), which appear in our LDA and NMF financial transfer topics. The topic model captures this vulnerability pattern by identifying the co-occurrence of external value transfers with balance management operations—contracts containing both semantic elements receive high probabilities for reentrancy-related topics, enabling detection without explicit control flow analysis.

The *"tx.origin"* vulnerability included in our dataset involves using the *"tx.origin"* address (it identifies the original address that sent the transaction) for authorisation, which can be exploited for phishing attacks. Using these Solidity feature for authorisation, creates semantic detectable patterns (e.g. "tx.origin", "owner", "sender" around transaction origin checking). The following example showcases the vulnerability issue:

Listing 5: tx.origin vulnerability example

```
function transferOwnership(address
    newOwner) public {
  // vulnerable: uses tx.origin
  require(tx.origin == owner);
  owner = newOwner;
}
```

The vulnerability 5 arises from the semantic distinction between *tx.origin* and *msg.sender*. While *tx.origin* refers to the externally owned account that initiated the transaction chain, `msg.sender` identifies the immediate caller. An attacker can exploit this distinction by writing a malicious contract that, when invoked by the owner, subsequently calls *transferOwnership*. The authorisation check, passes because `tx.origin` remains the owner despite the malicious contract being the immediate caller. Secure implementations employ `msg.sender` for authorisation, which correctly identifies the immediate calling context. Topic models detect this vulnerability through the presence of "tx", "origin", "owner", and "sender" terms, which appear in our token ownership and management topics. The distinction between secure and vulnerable code manifests in the specific term "tx.origin" versus "msg.sender", both of which the topic model captures through its vocabulary distribution.

Vulnerabilities due to bad randomness generation, which in our dataset are identified as *"weak-prng"* correlate with topics containing "hash", "timestamp", "blocktimestamp", and "block" terms. These exposures, rely on the deterministic nature of blockchain data sources used for pseudorandomness. Gaming and crowdsale topics capture these patterns since they frequently require randomness for fair distribution or outcome determination. The following code snippet demonstrates this vulnerability in a gambling context:

Listing 6: Weak pseudorandom number generation in gambling

```
function playLottery() public payable {
  require(msg.value == 1 ether);
  uint random = uint(keccak256(abi.
      encodePacked(
  block.timestamp, block.difficulty)))
      if (random < 10) { // 10    payable(
      msg.sender).transfer(address(this).
      balance);
  }
}
```

The vulnerability 6 originates from the predictability of the entropy sources: *block.timestamp* can be influenced by miners within certain bounds, while *block.difficulty* adjusts predictably according to the network hash rate. An attacker (possibly with mining capabilities), can manipulate these values to predict or influence the random outcome, thereby subverting the intended fairness of the lottery. Secure implementations employ external oracles (such as Chainlink VRF) that provide verifiable randomness from off-chain sources. The topic model identifies this pattern through terms like "timestamp", "block", "hash", "random", "player", and "winner", which appear in our gaming and gambling logic topics. Contracts employing external oracles produce different topic distributions containing terms like "oracle", "chainlink", or "vrf", enabling the model to distinguish between vulnerable and secure randomness generation approaches through different semantic.

Vulnerabilities involving *"unchecked-transfer"* occur within transfer operations but may require understanding control flow beyond semantic content. These vulnerabilities appear in ERC20 token transfer topics but depend on both missing error handling and semantic patterns, creating ambiguous detection signals. The Example 7 showcases the issue:

Listing 7: Unchecked transfer return value example

```
function distributeTokens(address[] memory
     recipients, uint amount) public {
  for (uint i = 0; i < recipients.length;
      i++) {
  token.transfer(recipients[i], amount);
      // return value not checked
  ownership[recipients[i]] = true;
  }
}
```

Specific ERC20 token implementations return *false* to notify transfer failure rather than reverting execution. When the return value remains unchecked, failed transfers do not halt execution, resulting in incorrect state updates (in this case, granting ownership despite transfer failure). Secure implementations either verify the return value explicitly or employ *SafeERC20* wrapper libraries that revert on failure. The topic model captures terms like "transfer", "token", "ownership", "approval", which appear in our token ownership and management topics. However, detection complexity arises because both vulnerable and secure implementations employ identical vocabulary. The distinction lies in the presence or absence of error checking logic rather than semantically distinct patterns.

Vulnerabilities involving token management such as *"arbitrary-send-erc20"* vulnerability intersects with token transfer topics through ERC20-related terminology. However, detection complexity arises because legitimate token transfers use identical semantic patterns to vulnerable implementations, requiring structural analysis to differentiate secure from insecure code.

However, our dataset also encompasses a subset of vulnerabilities that, given their structural rather than semantic nature, may prove challenging to detect with topic modelling. The vulnerability *"msg-value-loop"* occurs within loop constructs and requires control flow analysis rather than semantic understanding, and it can be difficult for topic models to identify the iterative context that enables this vulnerability. The *"mapping-deletion"* represent another vulnerability that topic features alone may completely miss, since it involves wrong deletion logic, and does not rely on specific semantic patterns.

Vulnerabilities involving the management of ERC20 and ERC721 tokens, may introduce false positives and negatives in our analysis. Topic models identify semantic patterns, but many implementations (such as OpenZeppelin) use established libraries that are secure. The presence of semantic patterns related to these exposures does not necessarily indicate vulnerability, creating semantic ambiguity.

7.5. Results — employing topics as additional features

After extracting topic distributions, we augmented our dataset with K additional features, where K is the number of topics identified. To

Table 6
General performance metrics for random forest with topic modelling (Test/Training).

Model	AUC	Accuracy	Precision	Recall	F1-Score
RF	0.951	0.730 ± 0.00542 / 0.956	0.890 ± 0.00504 / 0.964	0.794 ± 0.00651 / 0.995	0.840 ± 0.00551 / 0.980
RF with NMF	0.986	0.785 ± 0.00484 / 0.974	0.961 ± 0.00344 / 0.988	0.814 ± 0.00624 / 0.989	0.881 ± 0.00410 / 0.988
RF with LDA	0.955	0.789 ± 0.00294 / 0.952	0.903 ± 0.00417 / 0.962	0.802 ± 0.01011 / 0.998	0.849 ± 0.00626 / 0.980

Table 7
Comparison of standard random forest vs. topic modelling enhanced approaches.

Vulnerability	Standard Random forest				Random forest + LDA				Random forest + NMF			
	AUC	Precision	Recall	F1-score	AUC	Precision	Recall	F1-score	AUC	Precision	Recall	F1-score
arbitrary-send-erc20	0.981	0.833 ± 0.059	0.641 ± 0.067	0.725 ± 0.066	0.970	0.920 ± 0.023	0.451 ± 0.044	0.605 ± 0.031	0.979	1.000 ± 0.000	0.451 ± 0.044	0.622 ± 0.039
suicidal	0.656	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.484	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.997	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000
uninitialised-state	0.946	0.843 ± 0.045	0.708 ± 0.044	0.770 ± 0.033	0.954	0.912 ± 0.023	0.687 ± 0.019	0.784 ± 0.018	0.977	1.000 ± 0.012	0.584 ± 0.036	0.738 ± 0.030
arbitrary-send-erc20-permit	1.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000	1.000	1.000 ± 0.000	1.000 ± 0.000	1.000 ± 0.000
controlled-delegatecall	0.997	0.897 ± 0.029	0.788 ± 0.063	0.839 ± 0.047	0.967	0.933 ± 0.022	0.800 ± 0.102	0.862 ± 0.068	1.000	0.960 ± 0.000	0.686 ± 0.072	0.800 ± 0.047
reentrancy-eth	0.994	0.972 ± 0.003	0.920 ± 0.005	0.945 ± 0.004	0.995	0.983 ± 0.007	0.932 ± 0.009	0.957 ± 0.008	0.997	0.992 ± 0.001	0.932 ± 0.010	0.961 ± 0.006
reentrancy-no-eth	0.969	0.850 ± 0.017	0.720 ± 0.018	0.761 ± 0.016	0.971	0.850 ± 0.011	0.720 ± 0.020	0.780 ± 0.016	0.978	0.842 ± 0.029	0.758 ± 0.024	0.798 ± 0.023
tx-origin	0.992	0.902 ± 0.026	0.889 ± 0.030	0.895 ± 0.016	0.991	0.894 ± 0.026	0.884 ± 0.014	0.889 ± 0.016	0.995	0.963 ± 0.004	0.907 ± 0.025	0.934 ± 0.013
unchecked-transfer	0.962	0.750 ± 0.015	0.616 ± 0.031	0.677 ± 0.023	0.964	0.851 ± 0.025	0.639 ± 0.046	0.730 ± 0.039	0.988	0.923 ± 0.013	0.676 ± 0.032	0.781 ± 0.024
weak-prng	0.958	0.852 ± 0.095	0.511 ± 0.018	0.639 ± 0.020	0.969	0.857 ± 0.061	0.444 ± 0.122	0.585 ± 0.120	0.993	0.929 ± 0.079	0.481 ± 0.072	0.634 ± 0.057
unchecked-send	0.872	0.600 ± 0.389	0.111 ± 0.139	0.200 ± 0.267	0.943	1.000 ± 0.000	0.300 ± 0.098	0.462 ± 0.163	0.960	1.000 ± 0.000	0.100 ± 0.000	0.182 ± 0.000
unchecked-lowlevel	0.940	1.000 ± 0.064	0.500 ± 0.115	0.667 ± 0.091	0.931	1.000 ± 0.047	0.667 ± 0.065	0.800 ± 0.041	0.941	1.000 ± 0.000	0.444 ± 0.096	0.615 ± 0.077
msg-value-loop	0.987	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.827	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000	0.802	0.000 ± 0.000	0.000 ± 0.000	0.000 ± 0.000
unprotected-upgrade	0.959	0.909 ± 0.000	0.769 ± 0.192	0.833 ± 0.155	1.000	1.000 ± 0.000	1.000 ± 0.178	1.000 ± 0.135	1.000	1.000 ± 0.000	0.333 ± 0.215	0.500 ± 0.188

evaluate the predictive power of these topic-based features, we set up an experiment combining the topic distributions with traditional software metrics as input features. This approach tested the hypothesis that integrating topic-based features with established metrics would improve the classifiers' performance.

7.5.1. Classification employing topic distributions and metrics as features

We constructed our feature set by combining topic distributions from medium and high-impact vulnerable contracts with their software metrics. Using Non-Negative Matrix Factorisation (NMF), we achieved a reconstruction error of 0.000100 with 45 topics and 4-grams. Latent Dirichlet Allocation (LDA) reached a peak coherence score of 0.61 with 5 topics and 4-grams. For each document, we created a *Topic Distribution* column containing the topic probability vector, which we concatenated with software metrics for classification. Despite 5 topics yielding the highest LDA coherence score, we opted for 20 topics based on four key factors:

Stability Across N-gram Models: The 20-topic configuration maintained high coherence across 1-gram to 3-gram models.

Alignment with Known Vulnerability Types: This choice better aligned with our 33 vulnerability types.

Balance Between Specificity and Generalisability: It captured meaningful patterns while avoiding overfitting to specific phrases.

Enhanced Feature Set: The larger topic count enabled detection of subtle vulnerability-related patterns.

The incorporation of topic distributions from LDA and NMF enhanced the Random Forest model's performance across all metrics (see Table 6). The NMF-based model achieved the highest improvements, increasing precision from 0.890 to 0.961, AUC from 0.951 to 0.986, and recall from 0.794 to 0.814. The LDA-based model showed modest gains, with recall improving from 0.794 to 0.802 and F1-score from 0.839 to 0.849. Both approaches improved overall accuracy from 0.729 to 0.79. While NMF achieved higher overall scores, LDA demonstrated better generalisation potential, showing a smaller accuracy gap between training and testing (0.173 vs 0.189 for NMF). Both topic distribution methods reduced overfitting compared to the baseline model, as evidenced by decreased training-testing performance gaps.

Table 6 compares the standard Random Forest model with LDA and NMF-enhanced versions, showing improved accuracy and F1-scores across several vulnerabilities. Table 7 details performance by vulnerability type. The NMF model significantly improved detection of "*unchecked-transfer*" vulnerabilities (F1-score: 0.781 vs. 0.677 baseline), while LDA showed moderate improvement (F1-score: 0.730). Topic modelling enhanced detection rates for complex vulnerabilities while maintaining performance on well-detected cases. Both *"reentrancy"* types showed improvements, confirming that our semantic clustering capture the code constructs, as formulated in our qualitative analysis: *reentrancy-eth* increased from 0.945 (RF) to 0.957 (LDA) and 0.961 (NMF), while "*"reentrancy-no-eth"* improved from 0.761 (RF) to 0.780 (LDA) and 0.798 (NMF). However, rare vulnerabilities like *"suicidal"* and *"msg-value-loop"* remained undetected across all models. Some vulnerabilities showed mixed results: *"arbitrary-send-erc20"* decreased from 0.725 (RF) to 0.605 (LDA) and 0.622 (NMF), while *"unprotected-upgrade"* improved with LDA (F1-score: 1.000) but declined with NMF (0.500). The *"weak-prng"* vulnerability showed slight decreases (0.639 RF, 0.585 LDA, 0.634 NMF). Despite theoretical expectations that timestamp and randomness-related semantic patterns would aid detection, these terms appear frequently across many contract types, creating semantic noise rather than discriminative signals. This demonstrates that semantic ubiquity can reduce topic modelling effectiveness even when conceptual correlations exist.

These results indicate that smart contract security requires a combined approach. Topic modelling proves effective for vulnerabilities like *"reentrancy"* and *"unchecked-transfer"*, while traditional methods better detect others like *"arbitrary-send-erc20"*. In practice, teams should balance improved detection rates with results interpretability, potentially combining machine learning detection with targeted manual reviews. However, the results also reveal that our approach struggles with rare vulnerabilities lacking sufficient training examples or those requiring dynamic analysis for detection. The observed train-test performance gaps primarily reflect the inherent challenges of learning from extreme class imbalance rather than conventional overfitting. Our dataset contains 33 vulnerability types with highly skewed distributions, creating a fundamental data scarcity problem where powerful algorithms can memorise specific patterns of rare classes during training but struggle to generalise the same patterns to unseen data. We implemented several mitigation strategies including balanced class weighting, constrained hyperparameters, and stratified cross-validation. However, the performance gap persists because regularisation cannot create information that does not exist in the training data. The stratified cross-validation results provide the most reliable performance indicators, as they average across multiple train-test partitions, with our bootstrap validation confirming result stability within the statistical constraints imposed by data availability.

Answer to RQ3: The integration of topic distributions derived from LDA and NMF as additional features in our dataset showed modest improvements in multi-label classifier performance for vulnerability detection. The Random Forest classifier, trained with NMF-based topic distributions, improved the score for all metrics. The methodology showed particular efficacy in enhancing the detection of common vulnerabilities but demonstrated limited improvement for rare vulnerability types. These insights suggest that topic distributions can contribute to more robust vulnerability detection models, though the overall improvement may not be uniform across all aspects of the classification task, considering the possible noise introduced by topics distribution on vulnerabilities that heavily rely on metrics, and the highly skewed distributions of the 33 vulnerabilities encompassed within our dataset.

8. Threats to validity

Our study faces several validity threats, which we address through mitigation strategies.

Internal validity. We minimised feature selection bias by employing Adaptive LASSO, which improves feature consistency in high-dimensional datasets. To reduce model selection bias, we evaluated multiple classifiers representing different modelling paradigms; this design choice is introduced in the contributions section and discussed here for completeness.

External validity. While our dataset is substantial (74,225 contracts), it may not capture all possible vulnerability types or contract design patterns. To reduce over-representation of specific coding practices, we drew contracts from diverse sources (SmartBugs Curated, Smart Sanctuary, Smart Corpus). Nonetheless, results may not generalise to all deployment contexts.

Construct validity. Our reliance on Slither for vulnerability detection introduces several methodological constraints. Static analysis tools operate on source code without contract execution, limiting detection of runtime-dependent vulnerabilities. Comparative benchmarking (Durieux et al., 2020) demonstrates that Slither's detection accuracy varies substantially across vulnerability categories, with strong performance for reentrancy detection (88% on annotated vulnerable contracts) but poor performance for arithmetic vulnerabilities (0% detection). Precise false positive and false negative rates for Slither across our 33 vulnerability types remain unquantified in existing literature, representing a limitation in our ground truth validation. Alternative tools such as Mythril, Securify, and Oyente offer complementary detection capabilities. Durieux et al. (2020) found that combining multiple tools detected 42% of known vulnerabilities compared to 17% for Slither alone, suggesting that our Slither-only labelling may underrepresent certain vulnerability types whilst potentially overrepresenting others. Cross-validation with alternative tools on a sample of contracts could strengthen label confidence but was beyond this study's scope due to tool version incompatibility and computational constraints. Our filtering strategy affects dataset composition. By retaining only medium and high-severity vulnerabilities, we excluded low-severity issues representing code quality concerns rather than exploitable vulnerabilities. Whilst this aligns with security auditing priorities, it introduces selection bias: our models cannot identify low-severity issues that might become exploitable through contract interactions or specific deployment contexts. The severity classifications themselves reflect Slither's assessment framework, which may not align with all real-world exploitation scenarios. A subset of our dataset includes manually verified labels (SmartBugs Curated), providing partial validation of Slither's classifications for common vulnerability patterns. However, the majority of our 74,225 contracts rely solely on Slither's automated classification. Topic modelling applies a bag-of-words assumption to source code, which does not fully capture program structure. Despite this limitation, our empirical results show measurable improvements, suggesting that topics capture meaningful patterns even within this constraint.

Adversarial risk. In practice, attackers may attempt to evade detection through obfuscation or proxy patterns. Solidity's use of fixed keywords limits such attacks, but shadowing remains a potential risk for metric and topic-based detection. Nevertheless, more sophisticated altering techniques, such as using proxy patterns or indirect function calls, could circumvent both metric-based and topic-based detection. These approaches might manipulate structural metrics by distributing functionality across multiple contracts or alter semantic patterns through indirection layers that shadow vulnerability-related code constructs from static analysis.

Model bias and representativeness. We acknowledge that model bias due to over-representation of certain contract types or coding styles in our training data could affect performance on contracts with different architectural patterns. This bias could manifest in reduced detection accuracy for contracts employing novel design patterns, alternative development frameworks, or domain-specific implementations that deviate from mainstream coding conventions. Such representational limitations underscore the importance of continuous dataset expansion and validation across diverse contract architectures to ensure robust generalisation of our vulnerability detection approach.

Semantic noise from topic modelling. Our approach faces inherent limitations arising from semantic noise in topic modelling applications to source code. Both LDA and NMF techniques assume bag-of-words representations that may be fundamentally misaligned with program semantics, where token sequences are governed by syntax rather than semantic patterns. LDA's probabilistic assumption that documents represent mixtures of topics may not adequately capture the deterministic nature of code structures, while NMF's non-negative matrix factorisation approach, though effective at identifying distinct patterns, can struggle with the hierarchical and syntactic relationships inherent in programming languages. Our empirical results demonstrate that while NMF-derived features provided greater performance improvements than LDA across most metrics, both approaches showed differential benefits across vulnerability types. The varying effectiveness suggests that the semantic representations extracted by these models may sometimes introduce confounding signals rather than meaningful abstractions, particularly for vulnerabilities that depend primarily on structural properties rather than lexical patterns captured through bag-of-words assumptions.

Limitations with rare vulnerabilities. Our approach fails for several rare but potentially critical vulnerabilities. Vulnerabilities with fewer than 50 training examples (suicidal: 0 F1-score, msg-value-loop: 0 F1-score, unchecked-send: F1-score 0.2) show near-zero detection rates across all configurations, including topic-enhanced models. This represents a fundamental limitation of supervised learning approaches when confronted with extreme class imbalance rather than a hyperparameter tuning issue. The inherent data scarcity problem means that no amount of regularisation can create information that does not exist in the training data. Whilst class weighting and balanced sampling partially address common vulnerabilities, they cannot overcome the statistical challenge of learning reliable decision boundaries from fewer than 50 positive examples distributed across 10 cross-validation folds. This limitation affects the external validity of our findings, as the model cannot generalise to rare vulnerability patterns not adequately represented in training data.

Conclusion validity. We mitigated class imbalance with SMOTE (binary classification) and class weighting (multi-label classification). We also used multiple performance metrics and bootstrap validation to test stability and robustness. Results were consistent across these checks, supporting the reliability of our findings.

9. Future work

Our study establishes a foundation for combining software metrics with topic modelling in smart contract vulnerability detection, revealing several directions for future research.

Cross-platform evaluation and generalisability. While our methodology demonstrates effectiveness within the Ethereum ecosystem, validation across alternative blockchain platforms represents a critical research direction. Future work should evaluate the approach on other smart contract based platforms such as Solana, Cardano, and Hyperledger platforms, addressing the challenges of platform-specific programming languages and tooling ecosystems. Such cross-platform studies would provide empirical evidence about the transferability of our metrics-topic modelling framework and clarify performance variations across different blockchain environments.

Advanced representation learning. The limitations observed with traditional topic modelling suggest potential benefits from more sophisticated semantic analysis techniques. Transformer-based approaches such as CodeBERT and GraphCodeBERT offer promising alternatives that could capture both structural and semantic properties of code in a more integrated manner. These models may address some of the noise issues we identified with LDA and NMF while providing more nuanced understanding of code semantics within syntactic constraints.

Rare Vulnerability Detection Our findings reveal fundamental challenges in detecting rare vulnerability classes using traditional supervised learning approaches. Several critical vulnerabilities (suicidal, msg-value-loop, mapping-deletion) showed zero detection rates despite various mitigation strategies. Future research should explore alternative paradigms specifically designed for extreme class imbalance scenarios.

Focal loss and cost-sensitive learning. Focal loss (Lin et al., 2017) addresses class imbalance by down-weighting well-classified examples and focusing learning on hard cases. Unlike standard cross-entropy with class weights, focal loss dynamically adjusts the loss contribution based on prediction confidence, potentially improving rare vulnerability detection. Implementation would require extending our Random Forest classifier to support custom loss functions or adopting gradient boosting frameworks that natively support focal loss.

Few-shot learning approaches. Few-shot learning techniques, particularly prototypical networks and matching networks, are designed to learn from limited examples by leveraging meta-learning across related tasks. Applied to vulnerability detection, this could involve training on common vulnerability types and adapting to rare ones through metric learning. Siamese networks could learn similarity metrics between code representations, enabling classification of rare vulnerabilities by comparing them to learned prototypes rather than requiring extensive training examples.

Anomaly detection frameworks. Reframing rare vulnerability detection as an anomaly detection problem offers a complementary approach. One-class SVM, isolation forests, or autoencoder-based methods could model the distribution of secure code, flagging deviations as potentially vulnerable. This approach shifts from learning vulnerability patterns (which requires many examples) to learning normality patterns (which can leverage the abundance of non-vulnerable code). Hybrid systems combining supervised classification for common vulnerabilities with anomaly detection for rare cases represent a promising research direction.

Synthetic data augmentation. Program transformation techniques could generate synthetic vulnerable contracts by systematically introducing known vulnerability patterns into secure code. Whilst requiring careful validation to avoid introducing artifacts, this approach could provide additional training examples for underrepresented vulnerability classes. Combining this with adversarial training might improve model robustness.

Integration with Deep Learning Approaches Whilst computational constraints precluded direct comparison with transformer-based models such as CodeBERT or GraphCodeBERT in this study, future work should investigate hybrid architectures combining our interpretable metric-based approach with deep learning semantic representations. Such integration could address complementary weaknesses: transformer models capture fine-grained semantic patterns but lack architectural context, whilst our metrics provide structural indicators but may miss subtle implementation details. Potential integration strategies include: (1) using CodeBERT embeddings as additional features alongside metrics and topic distributions, (2) employing attention mechanisms to weight metric importance based on learned code representations, or (3) developing multi-task learning frameworks where structural metrics guide transformer attention towards security-relevant code regions. These approaches would require careful evaluation on representative dataset subsets to balance computational feasibility with methodological rigour.

Complete baseline comparisons. While computational constraints limited our comparison scope, future work should evaluate our approach against state of the art deep learning models for smart contract vulnerability detection. Such comparisons should include graph neural networks, sequence models, and ensemble methods, potentially using representative dataset subsets to balance computational feasibility with methodological rigour.

Enhanced future engineering. The interaction between structural metrics and semantic features warrants further investigation. Future studies could explore class-aware gating mechanisms, cost-sensitive training approaches, and alternative code representation techniques that better capture vulnerability-inducing patterns while minimising semantic noise.

Vulnerability specific feature engineering. Future research could explore ablation studies to isolate individual metric contributions and quantify their isolated predictive power across different vulnerability classes. This empirical validation could inform the development of vulnerability-specific models that optimise feature sets for particular security issues. Reentrancy vulnerabilities might benefit from models emphasising coupling and function call metrics, while access control issues could prioritise modifier-related features. Such specialised approaches could potentially improve detection accuracy compared to general-purpose classification while providing clearer insights into the structural patterns that predispose contracts to specific exposure types. Moreover, investigating class-aware feature selection techniques could help identify optimal metric combinations for rare vulnerability detection, addressing current limitations in handling underrepresented vulnerability classes through targeted feature engineering strategies.

Cross-Platform Evaluation and Generalisability Whilst our methodology demonstrates effectiveness within the Ethereum ecosystem, validation across alternative blockchain platforms represents an important research direction. Future work should evaluate the approach on other smart contract platforms such as Solana (Rust-based), Cardano (Plutus/Haskell), and Hyperledger Fabric (Go/Java), addressing the challenges of platform-specific programming languages and tooling ecosystems. Each platform presents distinct characteristics: Solana's architecture differs fundamentally from Ethereum's, Cardano employs functional programming paradigms, and Hyperledger targets permissioned networks with different security assumptions. Adapting our metrics-topic modelling framework would require platform-specific metric extractors, vulnerability taxonomies aligned with each platform's attack surface, and validation that topic modelling effectively captures semantic patterns in non-Solidity languages.

Such cross-platform studies would provide empirical evidence about the transferability of combining structural metrics with semantic analysis for vulnerability detection, clarifying whether performance variations reflect fundamental differences in programming paradigms, platform architectures, or developer practices across blockchain ecosystems.

These research directions reflect the multifaceted nature of smart contract security challenges, where no single approach provides complete coverage across all vulnerability types and deployment contexts.

The immutable nature of smart contracts increases the consequences of security failures, making robust vulnerability detection systems essential for blockchain ecosystem stability. While our study demonstrates the value of combining structural and semantic analysis, significant gaps remain in detecting rare vulnerabilities and ensuring cross-platform applicability.

10. Conclusions

10.1. Practical implementation considerations

The deployment of our vulnerability detection methodology in real-world development environments involves several practical considerations that warrant discussion:

Computational requirements: Our approach combines static analysis with topic modelling, both of which present manageable computational demands for typical smart contract auditing scenarios. Static analysis using tools like Slither operates efficiently on individual contracts, while topic modelling represents a one-time training overhead that can be amortised across multiple auditing sessions. For typical decentralised applications (dApps) projects involving hundreds of contracts, computational costs remain within practical bounds for most development teams.

Integration challenges: The primary barriers to adoption relate to dependency management rather than computational constraints. Smart contract analysis requires access to all external dependencies and library imports, which may not always be available in development environments. Version compatibility presents another challenge, as emerging Solidity versions may introduce incompatibilities with existing analysis tools or require model retraining to handle new vulnerability patterns.

Adoption barriers: The primary barriers to adoption relate to dependency management rather than computational constraints. Smart contract analysis requires access to all external dependencies and library imports, which may not always be available in development environments. Version compatibility presents another challenge, as emerging Solidity versions may introduce incompatibilities with existing analysis tools or require model retraining to handle new vulnerability patterns.

Workflow integration: The methodology can be integrated into continuous integration pipelines as an automated pre-screening step, flagging contracts that warrant manual security review. Development teams should balance automated detection capabilities with expert audit processes, using our approach to prioritise security attention rather than replace human analysis.

10.2. Conclusion

Our study demonstrates that combining software metrics with topic modelling can provide measurable improvements for smart contract vulnerability detection, while revealing important limitations of current approaches. Metrics like Cyclomatic Complexity, Nesting Depth, and Function Calls emerge as significant vulnerability predictors, and our Random Forest classifier achieved strong performance for common vulnerability types in both binary and multi-label classification tasks. The integration of topic modelling, particularly through Non-Negative Matrix Factorisation, enhanced classification performance for certain vulnerability classes, improving F1-scores from 0.839 to 0.881.

However, our results also highlight significant challenges in vulnerability detection. The approach struggles with rare vulnerabilities that lack sufficient training examples, with several exposure patterns (e.g., suicidal, msg-value-loop) showing zero detection rates across all models. The inherent class imbalance in vulnerability datasets creates fundamental data scarcity problems that cannot be resolved through regularisation alone. Topic modelling provides differential benefits across vulnerability types, suggesting that semantic features complement structural metrics for some vulnerabilities while offering limited value for others.

The significant gaps identified in our study point to multiple critical research directions, including the need for specialised approaches to rare vulnerability detection, cross-platform validation, and integration with more sophisticated code representation techniques. Future work should also focus on exploring advanced topic modelling techniques to minimise semantic noise, and exploring techniques such as anomaly detection, class-aware gating mechanism, and cost-sensitive training approaches.

CRediT authorship contribution statement

Giacomo Ibba: Conceptualisation, Methodology, Software, Investigation, Data curation, Formal analysis, Writing – original draft. **Rumyana Neykova:** Conceptualisation, Methodology, Data curation, Formal analysis, Writing – original draft, Writing – review & editing, Supervision. **Marco Ortu:** Conceptualisation, Methodology, Data curation, Formal analysis, Writing – original draft, Writing – review & editing, Supervision. **Roberto Tonelli:** Writing – review & editing. **Steve Counsell:** Methodology, Writing – review & editing. **Giuseppe Destefanis:** Conceptualisation, Methodology, Data curation, Formal analysis, Writing – original draft, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare no competing financial interests or personal relationships that could have influenced the work reported in this paper.

Data availability

Data and scripts are shared within the replication package.

References

Atzei, N., Bartoletti, M., & Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). In *Principles of security and trust: 6th international conference, POST 2017, held as part of the European joint conferences on theory and practice of software, ETAPS 2017, uppsala, Sweden, April 22-29, 2017, proceedings 6* (pp. 164–186). Springer.

Aufiero, S., Ibba, G., Bartolucci, S., Destefanis, G., Neykova, R., & Ortu, M. (2024). Dapps ecosystems: Mapping the network structure of smart contract interactions. *EPJ Data Science*, *13*(1), 60.

Boser, B. E., Guyon, I. M., & Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on computational learning theory* (pp. 144–152).

Breiman, L. (2001). Random forests. *Machine Learning*, *45*, 5–32.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, *16*, 321–357.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493.

Cramer, J. S. (2002). The origins of logistic regression.

Davison, A. C., & Hinkley, D. V. (1997). *Bootstrap methods and their application: No. 1*, Cambridge University Press.

Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., & Hierons, R. (2018). Smart contracts vulnerabilities: a call for blockchain software engineering? In *2018 international workshop on blockchain oriented software engineering* (pp. 19–25). IEEE.

Durieux, T., Ferreira, J. F., Abreu, R., & Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 ethereum smart contracts. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering* (pp. 530–541).

Efron, B. (1992). Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics: methodology and distribution* (pp. 569–593). Springer.

Feist, J., Grieco, G., & Groce, A. (2019). Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd international workshop on emerging trends in software engineering for blockchain* (pp. 8–15). IEEE.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Friedman, J. H. (2001). Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, 1189–1232.

Han, D., Li, Q., Zhang, L., & Xu, T. (2022). A smart contract vulnerability detection model based on graph neural networks. In *2022 4th international conference on frontiers technology of information and computer* (pp. 834–837).

Hao, X., Ren, W., Zheng, W., & Zhu, T. (2020). SCScan: A SVM-based scanning system for vulnerabilities in blockchain smart contracts. In *2020 IEEE 19th international conference on trust, security and privacy in computing and communications* (pp. 1598–1605). ISSN: 2324-9013.

Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., & Rosu, G. (2018). KEVM: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st computer security foundations symposium* (pp. 204–217). ISSN: 2374-8303.

Ibba, G., Aufiero, S., Bartolucci, S., Neykova, R., Ortu, M., Tonelli, R., & Destefanis, G. (2024). Mindthedapp: a toolchain for complex network-driven structural analysis of ethereum-based decentralised applications. *IEEE Access*.

Ibba, G., Aufiero, S., Neykova, R., Bartolucci, S., Ortu, M., Tonelli, R., & Destefanis, G. (2024). A curated solidity smart contracts repository of metrics and vulnerability. In *Proceedings of the 20th international conference on predictive models and data analytics in software engineering* (pp. 32–41).

Ibba, G. F., Destefanis, G., & Neykova, R. (2024). A curated solidity smart contracts repository of metrics and vulnerabilities.

Jiang, B., Liu, Y., & Chan, W. (2018). ContractFuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM international conference on automated software engineering* (pp. 259–269). ISSN: 2643-1572.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision* (pp. 2980–2988).

Lou, Y., Zhang, Y., & Chen, S. (2020). Ponzi contracts detection based on improved convolutional neural network. In *2020 IEEE international conference on services computing* (pp. 353–360). ISSN: 2474-2473.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., & Hobor, A. (2016). Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (pp. 254–269). New York, NY, USA: Association for Computing Machinery.

Medeiros, N., Ivaki, N., Costa, P., & Vieira, M. (2017). Software metrics as indicators of security vulnerabilities. In *2017 IEEE 28th international symposium on software reliability engineering* (pp. 216–227). ISSN: 2332-6549.

Medeiros, N., Ivaki, N., Costa, P., & Vieira, M. (2020). Vulnerable code detection using software metrics and machine learning. *IEEE Access*, *8*, 219174–219198.

Okutan, A., & Yıldız, O. T. (2014). Software defect prediction using Bayesian networks. *Empirical Software Engineering*, *19*, 154–181.

Ortu, M., Vacca, S., Destefanis, G., & Conversano, C. (2022). Cryptocurrency ecosystems and social media environments: An empirical analysis through hawkes models and natural language processing. *Machine Learning with Applications*, *7*, 100229.

Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Fahl, S., & Acar, Y. (2015). VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (pp. 426–437). New York, NY, USA: Association for Computing Machinery.

Pinna, A., Ibba, S., Baralla, G., Tonelli, R., & Marchesi, M. (2019). A massive analysis of ethereum smart contracts empirical study and code metrics. *IEEE Access*, *7*, 78194–78213.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536.

Shepperd, M., Guo, Y., Li, N., Arzoky, M., Capiluppi, A., Counsell, S., Destefanis, G., Swift, S., Tucker, A., & Yousefi, L. (2019). The prevalence of errors in machine learning experiments. In *Intelligent data engineering and automated learning–IDEAL 2019: 20th international conference, manchester, UK, November 14–16, 2019, proceedings, part i 20* (pp. 102–109). Springer.

Singh, P. D., & Chug, A. (2017). Software defect prediction analysis using machine learning algorithms. In *2017 7th international conference on cloud computing, data science & engineering-confluence* (pp. 775–781). IEEE.

Singh, Y., Kaur, A., & Malhotra, R. (2010). Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality Journal*, *18*, 3–35.

Song, J., He, H., Lv, Z., Su, C., Xu, G., & Wang, W. (2019). An efficient vulnerability detection model for ethereum smart contracts. In *Network and system security: 13th international conference, NSS 2019, sapporo, Japan, December 15–18, 2019, proceedings* (pp. 433–442). Berlin, Heidelberg: Springer-Verlag.

Tonelli, R., Pierro, G. A., Ortu, M., & Destefanis, G. (2023). Smart contracts software metrics: A first study. *PLoS One*, *18*(4), Article e0281043.

Wang, W., Song, J., Xu, G., Li, Y., Wang, H., & Su, C. (2021). ContractWard: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, *8*(2), 1133–1144.

Zhang, H., Zhang, X., & Gu, M. (2007a). Predicting defective software components from code complexity measures. In *13th Pacific rim international symposium on dependable computing* (pp. 93–96). IEEE.

Zhang, H., Zhang, X., & Gu, M. (2007b). Predicting defective software components from code complexity measures. In *13th Pacific rim international symposium on dependable computing* (pp. 93–96).

Zheng, Z., Xie, S., Dai, H.-N., Chen, X., & Wang, H. (2018). Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, *14*(4), 352–375.