Machine Learning–Based Vulnerability Detection in Ethereum Smart Contracts via EVM Bytecode Feature Engineering

Solovev Sergei
Faculty of Computer Science, HSE University
sesesolovev@edu.hse.ru
https://github.com/SergeySolovyev/Machine-Learning-Based-Vulnerability-Detection

February 26, 2026

Abstract

Smart contract vulnerabilities have led to losses exceeding billions of US dollars in the decentralised finance (DeFi) ecosystem. Existing detection tools based on symbolic execution and static analysis, while precise, are computationally expensive and often impractical for large-scale screening. In this work, we propose a lightweight machine learning approach that operates directly on compiled EVM bytecode, requiring neither source code nor contract ABI. We design a feature engineering pipeline that extracts 65 security-oriented numerical features from disassembled bytecode instructions, covering reentrancy patterns, arithmetic overflow indicators, gas-based denial-of-service risks, access control anomalies, and environmental dependencies. Using a dataset of 117,091 real-world Ethereum smart contracts labelled by the Slither static analyser, we evaluate four classifiers—Logistic Regression, Decision Tree, Random Forest, and XGBoost—under stratified 5-fold cross-validation. XGBoost, optimised via Bayesian hyperparameter search (Optuna, 50 trials), achieves an F1-score of 0.947 on cross-validation and 93% accuracy on a held-out validation set, with 0.97 recall for vulnerable contracts and 0.85 recall for safe contracts. We additionally benchmark text-based opcode sequence representations and find that hand-crafted numerical features substantially outperform n-gram vectorisation approaches. All code, trained models, and the feature extraction pipeline are publicly available.

Keywords: smart contract security, vulnerability detection, EVM bytecode analysis, feature engineering, XGBoost, DeFi

1 Introduction

Decentralised Finance (DeFi) has emerged as one of the most significant applications of blockchain technology. Smart contracts—self-executing programs deployed on platforms such as Ethereum—form the backbone of DeFi protocols including decentralised exchanges, lending platforms, and yield aggregators. However, the immutable nature of deployed smart contracts means that vulnerabilities present in the code cannot be patched post-deployment, making pre-deployment security analysis critically important.

Historical exploits have demonstrated the severity of smart contract vulnerabilities. Notable incidents include the DAO hack (2016, $60M), the Parity wallet freeze (2017, $150M), and numerous flash-loan attacks targeting DeFi protocols. The Smart Contract Weakness Classification (SWC) registry catalogues over 37 known vulnerability patterns, ranging from reentrancy (SWC-107) and integer overflow (SWC-101) to denial-of-service via gas limits (SWC-113) and transaction order dependence (SWC-114).

Existing vulnerability detection approaches can be broadly categorised into three families:

1. **Symbolic execution** tools (e.g., Mythril, Manticore) explore execution paths systematically but suffer from path explosion and high computational cost.
2. **Static analysis** frameworks (e.g., Slither, Securify) operate on source code or intermediate representations but require access to Solidity source, which is not always available on-chain.
3. **Machine learning** methods aim to learn vulnerability patterns from data, offering a scalable alternative that can operate on raw bytecode.

A key practical consideration is that only compiled bytecode is universally available on the Ethereum blockchain; source code is published for only a fraction of deployed contracts. This motivates our focus on *bytecode-level* analysis.

In this paper, we make the following contributions:

- We design a comprehensive feature engineering pipeline that extracts 65 security-oriented numerical features from disassembled EVM bytecode, organised into 15 semantic categories (Section 4.3).
- We evaluate four classification algorithms on a large-scale dataset of 117,091 labelled Ethereum contracts, demonstrating that XGBoost with Bayesian hyperparameter optimisation achieves an F1-score of 0.947 (Section 5).
- We compare hand-crafted features against text-based opcode sequence representations and show a substantial performance gap in favour of domain-specific feature engineering (Section 4.6).
- We release our feature extraction code as a reusable, scikit-learn-compatible transformer.

2 Related Work

2.1 Traditional Analysis Tools

Mythril [Mueller, 2018] performs symbolic execution on EVM bytecode to detect a range of SWC-classified vulnerabilities. Slither [Feist et al., 2019] is a Solidity static analysis framework that operates on the abstract syntax tree and can detect over 80 vulnerability patterns. Securify [Tsankov et al., 2018] uses abstract interpretation to verify compliance and violation patterns. Maian [Nikolić et al., 2018] focuses specifically on three categories: suicidal contracts, prodigal contracts, and greedy contracts. While effective, these tools are computationally intensive and produce high false-positive rates in practice.

2.2 Machine Learning Approaches

Several studies have applied ML to smart contract vulnerability detection. Ferreira Torres et al. [2018] benchmarked automated analysis tools and found that individual tools detect at most 10% of vulnerabilities. Wang et al. [2021] proposed ContractWard, which uses bigram features from opcode sequences. Wu et al. [2021] introduced hand-crafted features from contract metadata combined with neural networks. Zhuang et al. [2020] applied graph neural networks to source-code-level contract function graphs. Liu et al. [2021] combined expert features with deep learning for bytecode-level detection.

Our work differs from these approaches in several ways: (i) we operate exclusively on compiled bytecode, not requiring source code; (ii) we engineer a comprehensive set of 65 features that are explicitly mapped to known SWC vulnerability categories; (iii) we use a substantially larger dataset (117K contracts) than most prior studies.

3 Dataset

3.1 Data Source

We use the publicly available dataset of Slither-audited smart contracts from HuggingFace[1], which contains 120,608 real-world Ethereum smart contracts. Each contract includes the Solidity source code, compiled bytecode, contract address, and the full JSON report from the Slither static analyser.

The Slither analyser operates on source code and provides reliable vulnerability labels. We use Slither's output as ground truth for our binary classification task (vulnerable vs. safe), since the analyser has access to type information, function signatures, and control flow that are not available at the bytecode level.

3.2 Preprocessing

We apply the following preprocessing steps:

1. Remove the `0x` prefix from bytecode strings.
2. Remove 3,517 duplicate entries identified by (source_code, bytecode) pairs.
3. Filter out contracts with empty bytecode.
4. Parse Slither JSON reports and map detector outputs to vulnerability labels using a predefined mapping.
5. Create a binary label: `is_vulnerable` $= 1$ if the contract contains at least one vulnerability, 0 otherwise.

After preprocessing, 117,091 contracts remain.

3.3 Data Splitting

We perform a stratified 90/10 train–validation split with random seed 376. Table 1 summarises the resulting class distribution.

Table 1: Dataset split and class distribution.

Split	**Vulnerable (1)**	**Safe (0)**	**Total**
Training	72,196 (68.7%)	32,831 (31.3%)	105,027
Validation	8,022 (68.7%)	3,648 (31.3%)	11,670
Total	80,218	36,479	117,091[2]

The dataset exhibits moderate class imbalance (approximately 2.2:1), which we address through the `scale_pos_weight` hyperparameter during model training.

4 Methodology

4.1 Overview

Our pipeline consists of four stages: (1) bytecode disassembly, (2) feature extraction, (3) model training with hyperparameter optimisation, and (4) threshold tuning for deployment. Figure 2 illustrates the overall architecture.

[1] https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts

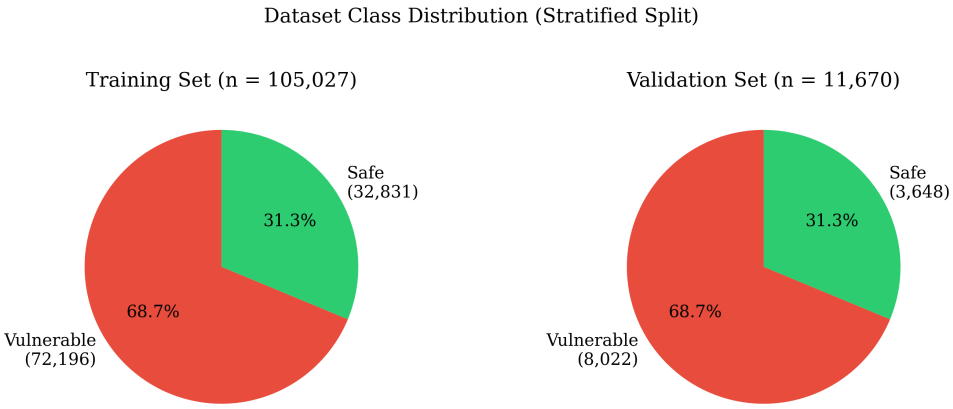

Figure 1: Class distribution in the dataset.

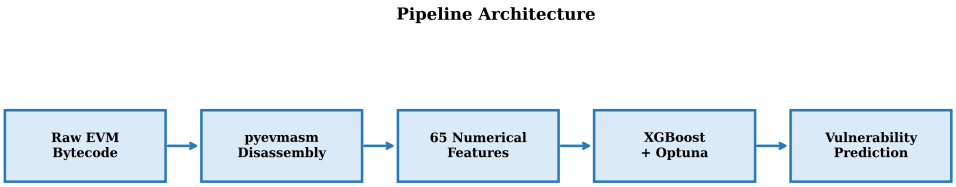

Figure 2: High-level pipeline architecture.

4.2 Bytecode Disassembly

We use the `pyevmasm` library (v0.2.3) to disassemble raw EVM bytecode into a sequence of typed instruction objects. Each instruction provides the following attributes: mnemonic (e.g., `CALL`, `SSTORE`), opcode value, program counter (PC), gas fee, number of stack items pushed/popped, operand size, instruction group (e.g., "Environmental Information"), and semantic category.

4.3 Feature Engineering

We design a custom scikit-learn-compatible transformer (`EVMBytecodeFeatureExtractor`) that extracts 65 numerical features from each disassembled contract. The features are organised into 15 semantic categories, each designed to capture indicators of specific SWC vulnerability classes. Table 2 provides a summary.

Table 2: Feature categories with counts and targeted vulnerability patterns.

Category	Count	Targeted Patterns
Basic statistics	2	Contract size and complexity
Block dependency	8	Timestamp dependence, front-running (SWC-116)
Environmental info	4	External data dependencies
External dependencies	6	Balance checks, caller/origin usage (SWC-115)
Calldata operations	5	Input parsing complexity
External calls	5	Reentrancy indicators (SWC-107)
Memory operations	3	Memory access patterns
Stack operations	5	Stack balance and underflow risk
Gas analysis	5	DoS via gas exhaustion (SWC-128)
Arithmetic operations	3	Integer overflow/underflow (SWC-101)
Control flow	4	Branching complexity, obfuscation
Access control	4	Weak authorisation (SWC-115)
Advanced patterns	3	Bad randomness (SWC-120), balance-before-call
Complexity metrics	3	Dangerous operations density, opcode entropy
Composite risk scores	5	Aggregated reentrancy, front-running, DoS, arithmetic, overall risk
Total	**65**	

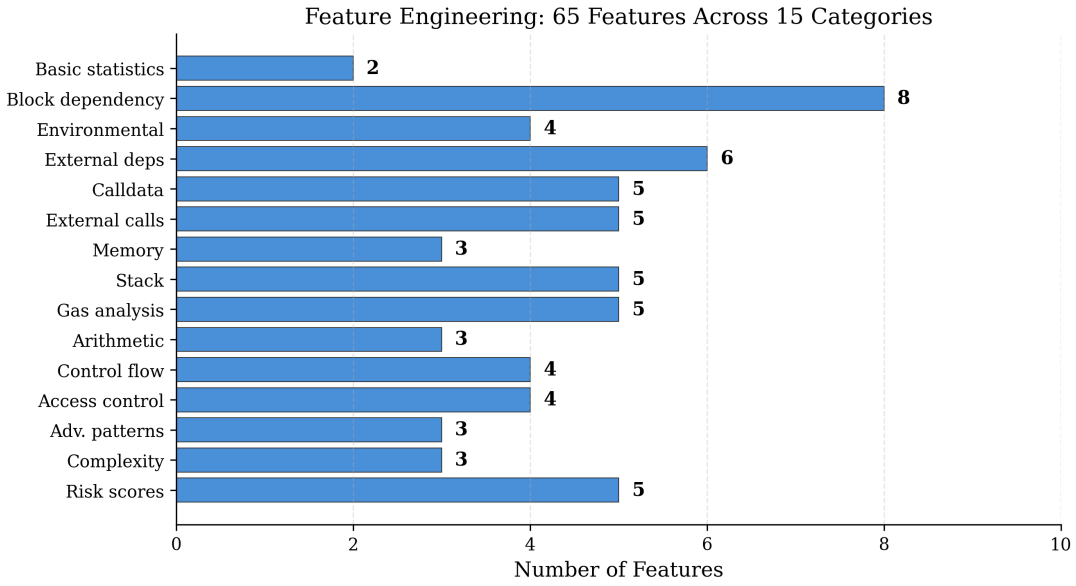

Figure 3: Distribution of feature categories used for bytecode analysis.

Below we describe several key feature families in detail.

4.3.1 Reentrancy Indicators

Reentrancy (SWC-107) is one of the most critical smart contract vulnerabilities. We capture it through multiple features:

- `external_call_count`: total number of `CALL`, `DELEGATECALL`, `STATICCALL`, and `CALLCODE` instructions.

- `potential_reentrancy_pattern`: a binary flag set to 1 when an `SSTORE` instruction occurs within 20 program counter positions before an external call instruction—a heuristic for the classic state-change-before-external-call pattern.
- `reentrancy_risk_score`: a composite score normalised by total instructions, aggregating external call count, call value operations, potential reentrancy patterns, and balance-before-call indicators.

4.3.2 Arithmetic Vulnerability Features

Integer overflow/underflow (SWC-101) is detected through:

- `arithmetic_ops_count`: total count of `ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `SDIV`, `SMOD`, `EXP`, and `SIGNEXTEND` instructions.
- `unsafe_arithmetic_pattern`: set to 1 when an arithmetic operation occurs within 5 PC positions before a `JUMPI` (conditional jump), indicating a potential unchecked overflow in Solidity versions prior to 0.8.
- `arithmetic_risk_score`: composite score incorporating arithmetic density and stack underflow risk.

4.3.3 Gas-Based DoS Features

Denial-of-service through gas exhaustion (SWC-128) is characterised by:

- `high_gas_instructions`: count of instructions with gas fee $> 1{,}000$.
- `gas_dos_risk_index`: ratio of high-gas instructions to total instructions.
- `dos_risk_score`: composite score combining gas risk index, control flow complexity, and stack underflow risk.

4.3.4 Opcode Entropy

We compute the Shannon entropy of the opcode mnemonic frequency distribution:

$$H = -\sum_{i=1}^{k} p_i \log p_i \tag{1}$$

where p_i is the relative frequency of the i-th unique mnemonic among k distinct opcodes. High entropy indicates diverse instruction usage (potentially obfuscated code), while low entropy suggests simple, repetitive patterns.

4.3.5 Composite Risk Scores

We aggregate category-level indicators into four composite risk scores plus an overall security risk score:

$$\text{reentrancy\_risk} = \frac{\text{ext\_calls} + \text{call\_value} + \text{reentrancy\_pattern} + \text{balance\_before\_call}}{N} \tag{2}$$

$$\text{frontrunning\_risk} = \frac{\text{block\_dependent} + \text{ext\_dependency\_idx} + \text{bad\_randomness}}{N} \tag{3}$$

$$\text{dos\_risk} = \frac{\text{gas\_dos\_idx} + \text{high\_gas} + \text{cf\_complexity} + \text{stack\_underflow}}{N} \tag{4}$$

$$\text{arithmetic\_risk} = \frac{\text{arith\_ops} + \text{unsafe\_arith} + \text{stack\_underflow}}{N} \tag{5}$$

$$\text{overall\_risk} = \text{mean}(\text{reentrancy}, \text{frontrunning}, \text{dos}, \text{arithmetic}, \text{dangerous\_density}, \text{ext\_dep\_idx}) \tag{6}$$

where N is the total number of instructions.

4.4 Model Selection

We evaluate four classifiers, chosen to represent different model families:

1. **Logistic Regression** (linear baseline): tests whether a linear decision boundary suffices.
2. **Decision Tree**: captures non-linear feature interactions without ensembling.
3. **Random Forest**: reduces variance through bagging of decision trees.
4. **XGBoost**: gradient-boosted trees with L1/L2 regularisation, histogram-based splitting, and native handling of class imbalance.

All models are evaluated using stratified 5-fold cross-validation with F1-score as the primary metric, appropriate for the observed class imbalance.

4.5 Hyperparameter Optimisation

For XGBoost, we perform Bayesian hyperparameter optimisation using Optuna [Akiba et al., 2019] with 50 trials. The search space is defined in Table 3.

Table 3: XGBoost hyperparameter search space.

Parameter	Range	Scale
`max_depth`	[3, 12]	integer
`learning_rate`	[0.01, 0.3]	log-uniform
`n_estimators`	[100, 2000]	integer
`min_child_weight`	[1, 10]	integer
`gamma`	[0.0, 5.0]	uniform
`subsample`	[0.5, 1.0]	uniform
`colsample_bytree`	[0.5, 1.0]	uniform
`reg_alpha`	[0.0, 2.0]	uniform
`reg_lambda`	[0.0, 5.0]	uniform
`scale_pos_weight`	[1, 20]	integer step

The objective function maximises the mean F1-score across 5 stratified folds. The tree method is set to `hist` for computational efficiency on the large dataset.

4.6 Text-Based Opcode Features

As a comparison, we also experiment with representing bytecode as a sequence of opcode mnemonics and applying standard NLP vectorisation methods:

- **CountVectorizer**: bag-of-n-grams with $n \in [1, 3]$ and up to 1,024 features.
- **TF-IDF Vectorizer**: term frequency–inverse document frequency weighting with $n \in [1, 3]$ and up to 1,024 features.

These text representations are fed to Logistic Regression for classification.

5 Results

5.1 Model Comparison

Table 4 presents the 5-fold cross-validation results for all four classifiers using the 65 numerical features.

Key observations:

- The substantial improvement from Logistic Regression (0.832) to Decision Tree (0.915) confirms that *non-linear feature interactions* are critical for vulnerability detection.
- Random Forest achieves the highest default F1 (0.946) through variance reduction via bagging.

Table 4: 5-fold cross-validation F1-scores on the training set (105,027 contracts).

Model	**Fold 1**	**Fold 2**	**Fold 3**	**Fold 4**	**Fold 5**
Logistic Regression	0.840	0.829	0.840	0.826	0.826
Decision Tree	0.914	0.917	0.915	0.915	0.914
Random Forest	0.945	0.945	0.946	0.946	0.946
XGBoost (default)	0.935	0.934	0.937	0.937	0.934

Table 5: Summary of cross-validation results: mean F1-score ± standard deviation.

Model	**Mean F1**	**Std**
Logistic Regression	0.832	0.007
Decision Tree	0.915	0.001
Random Forest	**0.946**	0.001
XGBoost (default)	0.935	0.001
XGBoost (Optuna)	**0.947**	—

- XGBoost with default parameters (0.935) slightly underperforms Random Forest, but after Optuna optimisation reaches 0.947, marginally surpassing Random Forest.
- All non-linear models exhibit remarkably low variance across folds ($\sigma \leq 0.001$), indicating stable generalisation.

5.2 Optimised XGBoost: Validation Set Performance

The final XGBoost model, trained with the best hyperparameters from 50 Optuna trials, achieves the following results on the held-out validation set (11,670 contracts):

Table 6: Classification report for the optimised XGBoost model on the validation set.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Safe (0)	0.92	0.85	0.88	3,648
Vulnerable (1)	0.93	0.97	0.95	8,022
Accuracy		—		**0.929**
Macro avg	0.93	0.91	0.92	11,670
Weighted avg	0.93	0.93	0.93	11,670

The model achieves 92.9% overall accuracy with:

- **0.97 recall for vulnerable contracts**: critically important for security applications, ensuring only 3% of truly vulnerable contracts are missed.
- **0.85 recall for safe contracts**: some safe contracts are flagged as false positives, which is acceptable in a screening scenario.
- **Balanced precision** (0.92–0.93 for both classes), indicating that positive predictions are reliable.

The best hyperparameters found by Optuna are: `max_depth` = 11, `learning_rate` = 0.141, `n_estimators` = 1,490, `min_child_weight` = 2, `gamma` = 0.038, `subsample` = 0.834, `colsample_bytree` = 0.502, `reg_alpha` = 0.103, `reg_lambda` = 1.939, `scale_pos_weight` = 4.

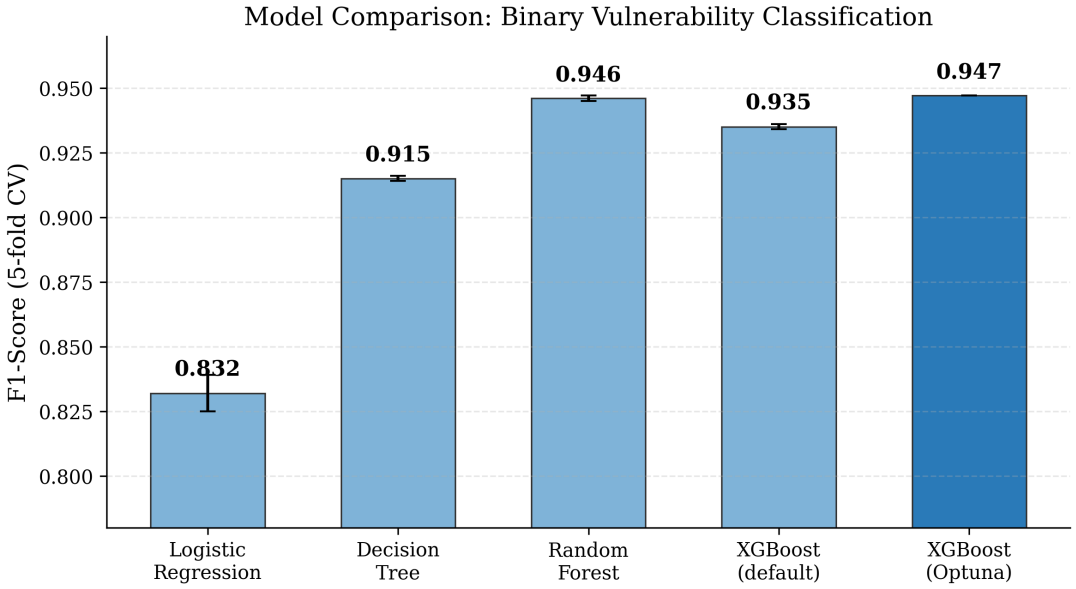

Figure 4: Model comparison on 5-fold cross-validation.

5.3 Text-Based Features: Comparison

Table 7 compares the numerical feature approach against text-based opcode sequence representations.

Table 7: Comparison of numerical features vs. text-based opcode representations (3-fold CV accuracy).

Feature Type	Model	Accuracy
65 numerical features	XGBoost (Optuna)	**0.929**
65 numerical features	Random Forest	0.946*
65 numerical features	Logistic Regression	0.832*
CountVectorizer (1,024, 1–3-grams)	Logistic Regression	0.865
TF-IDF (1,024, 1–3-grams)	Logistic Regression	0.795

*F1-score (5-fold CV); accuracy metrics differ slightly.

The results clearly show that hand-crafted numerical features, which encode domain knowledge about EVM semantics and known vulnerability patterns, substantially outperform generic text-based representations. The CountVectorizer approach achieves 86.5% accuracy—competitive with Logistic Regression on numerical features (83.2% F1)—but fails to capture the security-relevant structural information that domain-specific features provide.

5.4 Comparison with Traditional Tools

In preliminary experiments on a smaller dataset of 4,341 contracts labelled with Mythril vulnerability reports, we observe:

- **Mythril baseline** (binary prediction from scan results): 66% accuracy.
- **Our XGBoost model** on the same subset: substantially higher accuracy.

This suggests that our ML approach can complement traditional symbolic execution tools, offering faster inference with competitive or superior accuracy for binary vulnerability screening.

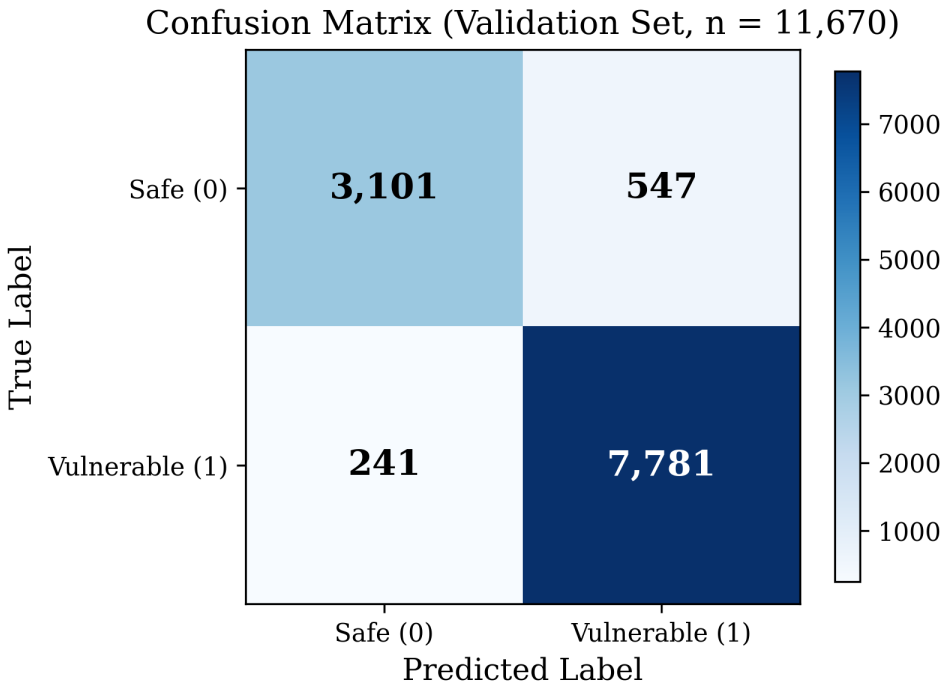

Figure 5: Confusion matrix for the optimised XGBoost model on the validation set.

5.5 Feature Importance

Analysis of XGBoost feature importances reveals the following top predictive features:

1. `pops` (stack pops)—contracts with different stack usage profiles exhibit distinct vulnerability patterns.
2. `block_dependent_count`—dependence on block variables is a strong vulnerability indicator.
3. `environmental_ratio`—the proportion of environment-querying instructions correlates with vulnerability.
4. `external_call_count`—the number of external calls is directly related to reentrancy and other inter-contract risks.
5. `total_gas_cost`—total gas cost reflects contract complexity; simpler contracts tend to be more vulnerable in our dataset.

An important empirical finding from exploratory data analysis on the 4,341-contract subset is that *safe contracts tend to be larger, more complex, and more expensive in gas*, while *vulnerable contracts tend to be smaller and simpler*. This counter-intuitive observation may reflect the fact that well-engineered contracts include additional safety checks and guards that increase bytecode size.

6 Discussion

Practical deployment. The trained model processes a single contract in milliseconds, making it suitable for real-time screening of newly deployed contracts. We envision its use as a first-pass filter: contracts flagged as vulnerable can be forwarded to more expensive symbolic execution tools (Mythril, Slither) for detailed analysis.

Threshold selection. In deployment, the choice of classification threshold depends on the operational priority:

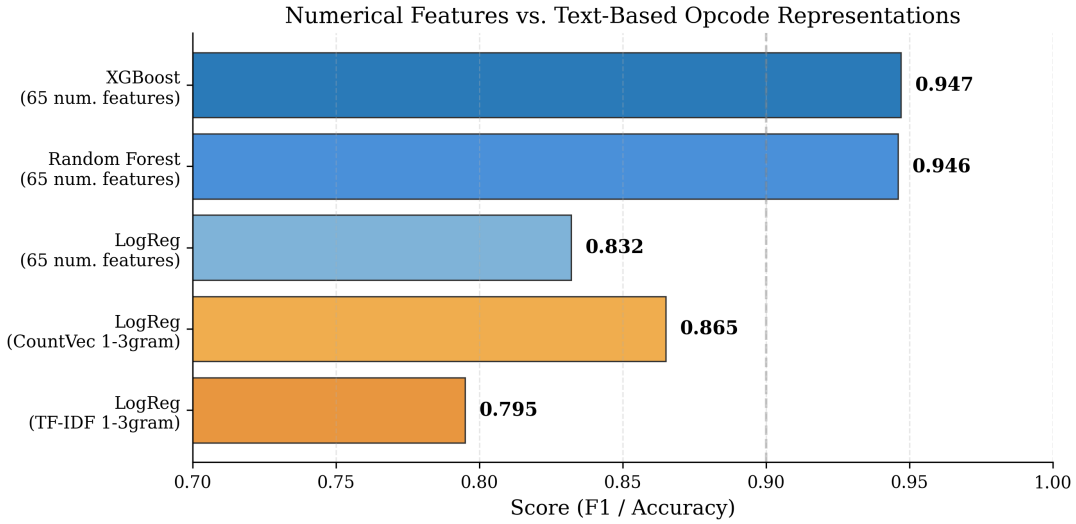

Figure 6: Numerical features vs. text-based representations.

- *High-recall mode* (threshold ≈ 0.3): minimises missed vulnerabilities at the cost of more false positives. Suitable for automated scanning pipelines.
- *High-precision mode* (threshold ≈ 0.7): reduces false positives, ensuring that flagged contracts are very likely vulnerable. Suitable for resource-constrained manual audit workflows.
- *Balanced mode* (threshold ≈ 0.5): the default setting, achieving the best F1-score.

Limitations. Several limitations should be noted:

1. **Label quality**: ground-truth labels are derived from Slither, which itself has false positives and false negatives. Our model's ceiling is bounded by label accuracy.
2. **Binary classification**: the current model predicts only the presence of *any* vulnerability, not the specific vulnerability type. Multi-label classification is a natural extension.
3. **Feature heuristics**: PC-proximity heuristics (e.g., "SSTORE within 20 positions of CALL") are approximations that do not account for jump destinations. More sophisticated control-flow graph analysis could improve pattern detection.
4. **Evolving attack surface**: new vulnerability patterns may emerge that are not captured by the current feature set. Periodic retraining on updated datasets is necessary.
5. **Solidity version effects**: the `unsafe_arithmetic_pattern` feature is most relevant for Solidity < 0.8, which introduced built-in overflow checks.

Comparison with deep learning. While deep learning approaches (CNNs, RNNs, GNNs) have been explored for smart contract analysis, our results demonstrate that a well-engineered feature set combined with gradient boosting achieves competitive performance with several advantages: interpretability, low computational cost, and no need for GPU infrastructure.

7 Conclusion

We have presented a machine learning pipeline for detecting vulnerabilities in Ethereum smart contracts using 65 hand-crafted features extracted from EVM bytecode. Our approach requires neither source code nor contract ABI, operating exclusively on the compiled bytecode available on-chain.

Evaluated on 117,091 real-world contracts, our optimised XGBoost classifier achieves 0.947 F1-score on cross-validation and 92.9% accuracy on held-out data, with 0.97 recall for vulnerable contracts. We show that domain-specific numerical features substantially outperform text-based

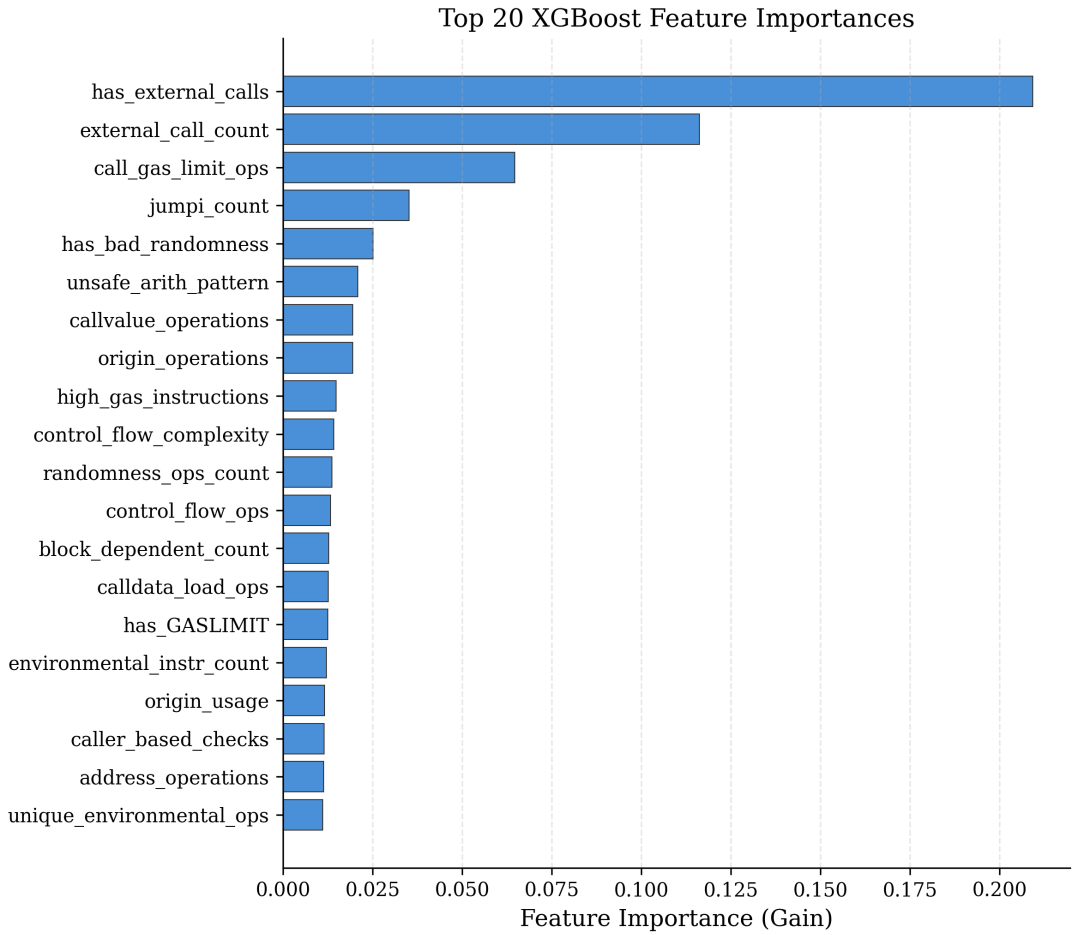

Figure 7: Top feature importances from the optimised XGBoost model.

opcode sequence representations, confirming the value of incorporating EVM semantics into the feature engineering process.

Future work includes extending to multi-label classification (predicting specific vulnerability types), incorporating control-flow graph analysis for more precise pattern detection, and benchmarking against deep learning baselines on the same dataset. All code and models are available at https://github.com/[REDACTED].

Acknowledgements

Data Availability

The smart contract dataset is publicly available at https://huggingface.co/datasets/mwritescode/slither-audited-smart-contracts. Code and trained models will be published upon acceptance.

References

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2623–2631. ACM, 2019. doi: 10.1145/3292500.3330701.

Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019. doi: 10.1109/WETSEB.2019.00008.

Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018.

Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. *arXiv preprint arXiv:2106.09282*, 2021.

Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. In *9th Annual HITB Security Conference*, 2018. Mythril: https://github.com/Consensys/mythril.

Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pages 653–663. ACM, 2018. doi: 10.1145/3274694.3274743.

Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 67–82. ACM, 2018. doi: 10.1145/3243734.3243780.

Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering*, 8(2):1133–1144, 2021. doi: 10.1109/TNSE.2020.2968505.

Huaiguang Wu, Zhipeng Zhang, Shengli Wang, Yingze Lei, Bo Shi, Chao Guo, and Yiqun Li. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. *arXiv preprint arXiv:2106.09440*, 2021.

Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3283–3290, 2020.