The Journal of Systems and Software 211 (2024) 112003

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

The impact of hard and easy negative training data on vulnerability prediction performance☆

Fahad Al Debeyan [a,*], Lech Madeyski [b,a], Tracy Hall [a], David Bowes [a]

[a] School of Computing and Communications, Lancaster University, Lancaster, UK
[b] Department of Applied Informatics, Wroclaw University of Science and Technology, Wroclaw, Poland

ARTICLE INFO

Keywords:
Software vulnerability prediction
Vulnerability datasets
Machine learning

ABSTRACT

Vulnerability prediction models have been shown to perform poorly in the real world. We examine how the composition of negative training data influences vulnerability prediction model performance. Inspired by other disciplines (e.g. image processing), we focus on whether distinguishing between negative training data that is 'easy' to recognise from positive data (very different from positive data) and negative training data that is 'hard' to recognise from positive data (very similar to positive data) impacts on vulnerability prediction performance. We use a range of popular machine learning algorithms, including deep learning, to build models based on vulnerability patch data curated by Reis and Abreu, as well as the MSR dataset. Our results suggest that models trained on higher ratios of easy negatives perform better, plateauing at 15 easy negatives per positive instance. We also report that different ML algorithms work better based on the negative sample used. Overall, we found that the negative sampling approach used significantly impacts model performance, potentially leading to overly optimistic results. The ratio of 'easy' versus 'hard' negative training data should be explicitly considered when building vulnerability prediction models for the real world.

1. Introduction

The rapid adoption of technology in all aspects of our lives poses major challenges to privacy and security. The National Vulnerability Database (NVD) defines software vulnerabilities as *a weakness in code that, when exploited, results in a negative impact to confidentiality, integrity, or availability of software systems*.[1] Current techniques to automatically detect software vulnerabilities, such as static and dynamic analysis, produce a high percentage of false positives and false negatives. For example, Pixy (Jovanovic et al., 2006), one of the most cited static analysers for vulnerability prediction, reports 48% false negatives. Therefore, developers are looking for better solutions to automatically detect software vulnerabilities in code (Johnson et al., 2013; Smith et al., 2015).

One possible solution is to use machine learning to predict whether areas of source code are likely to contain software vulnerabilities. Software vulnerability prediction models rely on datasets that contain instances of vulnerable code and other instances of non-vulnerable code to *learn* what is associated with a piece of code being vulnerable. In recent years, several software vulnerability prediction models have been published in the literature that show promising performance levels (Zhou et al., 2019; Li et al., 2018, 2022; Sultana et al., 2021). However, the performance of such models falls dramatically when tested on real-world systems (Chakraborty et al., 2022). Chakraborty et al. (2022) suggest the performance drop is a consequence of training and evaluating vulnerability prediction models on datasets that do not represent real-world scenarios. Such datasets may include duplicates, include synthetic data, or are not labelled correctly (Chakraborty et al., 2022). Previously (Al Debeyan et al., 2022) we re-evaluated several vulnerability prediction models in the literature on a dataset collected from GitHub projects and saw a similar performance drop. When we investigated what the reason might be for such a drop, we found that one main difference in the datasets used was the way the negative sample was collected. While some datasets include entire projects in the negative sample (e.g. Sultana et al. (2021)), other datasets only include the fixed version of vulnerable methods (e.g. Liu et al. (2020), Li et al. (2018)).

Unlike in software defect prediction, where datasets consist of an entire project, vulnerability prediction datasets consist of multiple different projects with a subset of the codebase included in the negative

☆ Editor: Dr. Hongyu Zhang.

* Corresponding author.
E-mail addresses: f.aldebeyan@lancaster.ac.uk (F. Al Debeyan), lech.madeyski@pwr.edu.pl (L. Madeyski), tracy.hall@lancaster.ac.uk (T. Hall), bowesd2@lancaster.ac.uk (D. Bowes).

[1] https://nvd.nist.gov/vuln

https://doi.org/10.1016/j.jss.2024.112003
Received 14 July 2023; Received in revised form 14 December 2023; Accepted 10 February 2024
Available online 20 February 2024

Fig. 1. Image examples of a positive, a hard negative, and an easy negative associated with the word "bus".

sample (Gkortzis et al., 2018; Ponta et al., 2019; Fan et al., 2020). The reason for not including the entire codebase in vulnerability prediction datasets is that, due to the rarity of vulnerabilities, the percentage of vulnerable code in projects can be as low as 0.1% in some projects. The immense imbalance between vulnerable and non-vulnerable code creates a challenge for vulnerability prediction models to learn from the vulnerable sample. To address the imbalance issue, researchers use datasets that exclude huge portions of the negative sample, resulting in a test set that does not accurately represent a real-world scenario where projects that include the entire negative sample are being evaluated.

The assembly of the negative sample and how the diversity of the negative sample impacts predictive performance have been neglected in vulnerability prediction. Other prediction domains have previously explored the negative sample. In particular, the concept of 'hard' and 'easy' negatives is used in computer vision (Xuan et al., 2020). Fig. 1 shows an example of images associated with the word "bus", including a positive, a hard negative and an easy negative.

Applying the same principle of including both hard and easy negatives to software vulnerabilities, we consider vulnerable methods as the positive sample. Hard negatives are methods similar to the positive sample but are still considered negative (not vulnerable). There are likely to be a range of different types of hard negatives in vulnerability data. Currently these types of negative data have not been investigated previously, but we propose that one type of hard negative is the patched versions of vulnerable methods. These patched methods are usually similar to the vulnerable version, except for the code changed to fix the vulnerability. Random methods from the latest version of a project where no known vulnerabilities have been identified are likely to be easy negatives. This easy negative sample would include simple methods such as getters and setters, as well as methods that have minimum similarities to the vulnerable sample. Future work is needed to uncover additional potential types of hard and easy negatives, and to identify ways in which these negatives can be mined.

In this paper, we investigate how the performance of vulnerability prediction models for real-world systems might be impacted by the training data. In particular, we investigate the effect of different approaches to collecting the negative sample in datasets for training and evaluating software vulnerability prediction models. Comparing two or more vulnerability prediction models on test sets that do not reflect the way models are used in practice may not be a fair comparison if the negative sampling strategy has an effect on the performance of a model.

We evaluate different machine learning models that predict software vulnerabilities in datasets that use different approaches to collect negative samples and we measure performance differences based on the negative sampling strategy. To do so, we ran our experiments using three vulnerability prediction models that showed promising results in the literature. The first is by using AST n-grams, taken from our previous work (Al Debeyan et al., 2022). The second model uses code metrics that showed promising performance predicting software vulnerabilities (Morrison et al., 2015; Sultana et al., 2021). he third model is a deep learning model named LineVul (Fu and Tantithamthavorn, 2022). LineVul is a transformer-based vulnerability prediction model that is known for its strong predictive abilities, with an F-measure of 91%. A recent study by Steenhoek et al. (2023) took a fresh look at multiple vulnerability prediction models, including LineVul, using different datasets such as Devign (Zhou et al., 2019) and MSR (Fan et al., 2020). LineVul stood out as the only model that consistently delivered good performance, with an F-measure exceeding 60%, across all the datasets. We also use AutoML (Feurer et al., 2015), a method to automatically search for and select a machine learning model achieving the highest performance measure for a given dataset. For each negative sampling strategy, we apply AutoML to find the machine learning model scoring the highest *Matthews's correlation coefficient (MCC)*, a metric suitable for imbalanced datasets (Yao and Shepperd, 2020), to assess whether the choice of model changes based on the choice of negative sampling strategy. Finally, we measure how a model trained on a dataset that only includes hard negatives performs when evaluated on a dataset that only includes easy negatives and vice versa. Some vulnerability prediction models in the literature (e.g. Li et al. (2018, 2022), Sultana et al. (2021)) are evaluated on datasets that only include one type of negative instances. However, non-vulnerable code in the real world is likely to contain a mixture of easy and hard negatives.

We aim to understand what makes a good negative sample in software vulnerability datasets and we answer the following research questions:

RQ1: Does the strategy of collecting negative samples affect the performance of ML models?

We explore whether optimistic predictive performance is observed in models using specific negative sampling strategies. We assemble four types of datasets, each differing in the way the negative sample is collected. Each dataset is evaluated using a range of popular machine learning classifiers (detailed in Section 3). We report the highest-performing classifiers (Random Forest, Naïve Bayes, Support Vector Machine, and Gradient Boosting). We report two other classifiers which produced lower performances (k-Nearest Neighbour and Logistic Regression) in our reproduction package (Al Debeyan, 2023).

RQ2: Does the strategy of collecting negative samples affect the choice of the ML algorithm and/or hyperparameter tuning?

When researchers present a new vulnerability prediction model, they often compare the performance of their model with other vulnerability prediction models in the literature. In some cases, the models are evaluated on different datasets where the negative sample is collected using different strategies. The answer to RQ2 will assist in determining whether the choice of machine learning algorithm changes depending on the negative sampling strategy. If this is the case, it demonstrates that the superior performance of one vulnerability prediction model over another can be claimed only if both models are evaluated on a test set that is representative of the real world. The way we have addressed RQ2 is by using automated machine learning (AutoML), based on Auto-Sklearn (Feurer et al., 2015), on four datasets that only differ in the negative sampling strategy and by observing whether the model uses the same algorithm and hyperparameter settings or not.

RQ3: How would a vulnerability prediction model trained on hard negative methods perform in predicting easy negative methods?

There are vulnerability prediction models in the literature that are evaluated on datasets that only include hard negatives (e.g. Li et al. (2022, 2018)). But if such models are to be used in practice, they must also be able to identify easy negatives. The answer to RQ3 will

help establish how a model trained on a dataset that only includes hard negatives responds when faced with easy negatives. To address RQ3, we trained an SVM model with code metrics, an SVM model with AST n-grams, and a deep learning model named LineVul (Fu and Tantithamthavorn, 2022) on datasets that include only hard negatives and evaluated these models on instances of easy negatives to measure if there is any performance difference.

RQ4: How would a vulnerability prediction model trained on easy negative methods perform in predicting hard negative methods?

In software vulnerability prediction, some researchers use the code base of an entire project as the dataset for evaluation (e.g. Sultana et al. (2021), Fu and Tantithamthavorn (2022)). Doing so can result in a negative sample that is almost entirely easy negatives. For example, Sultana et al. (2021) evaluated their proposed model on Apache CXF where there are 45 vulnerability fixes out of 26,366 negative methods. That means that less than 0.2% of the negative sample is hard negatives. The answer to RQ4 will help us understand how a model that is only trained on easy negatives would perform predicting hard negatives. To answer RQ4, we trained a Gradient Boosting model with code metrics, a Random Forest model with AST n-grams, and LineVul (Fu and Tantithamthavorn, 2022) on a dataset that only included easy negatives and evaluated these models on instances of hard negatives to measure the change in performance.

RQ5: What ratio of positives/easy negatives in the training set provides the highest performance on test sets that include entire projects? As described in Section 1, including the entire codebase in a vulnerability prediction dataset introduces a great challenge for vulnerability prediction models to learn from vulnerable instances due to the large imbalance between vulnerable and non-vulnerable instances. To work around this problem, we reduce the size of the negative sample in the training set while keeping all negative instances in the test set. If the negative sampling strategy does have an effect on the performance of the vulnerability prediction model, RQ5 will provide insight into the ratio of positives to easy negatives to include in the training set that will yield the highest MCC when the model is evaluating entire projects, a scenario that is more realistic of how programmers would use vulnerability prediction models in the real world. To answer RQ5, we trained three types of models (AST n-grams, code metrics, and LineVul) on training sets of different ratios of positive to easy negative samples and tested on a test set that includes the full set of easy and hard negatives.

2. Background

2.1. Software vulnerability prediction

Software vulnerabilities have been said to be a subset of software defects (Shin and Williams, 2008). However, Morrison et al. (2015) showed that using the same model features from defect prediction models to predict vulnerabilities does not always achieve the same performance level. Morrison et al. suggest that model features must be revisited to achieve acceptable performance, possibly through security-specific metrics. Shin and Williams (2008) published one of the first papers that uses a logistic regression model to identify code complexity features that differentiate vulnerable functions and faulty functions. Shin and Williams found that vulnerable functions have distinctive characteristics compared to non-vulnerable functions. The distinctive features are particularly related to nested complexity. Shin and Williams also report that predicting vulnerabilities from source code using complexity metrics is a feasible approach with low false positives but still misses many vulnerabilities. Another difference is the way datasets are curated between software defect prediction and software vulnerability prediction.

Although software vulnerability prediction can be considered a subset of software defect prediction, the datasets used for vulnerability prediction are different from the datasets used for defect prediction with respect to the negative sample. In defect prediction, the negative sample usually includes the entire project's code base. However, in vulnerability prediction, due to the low number of vulnerabilities in projects, vulnerability prediction datasets are usually assembled to include vulnerable instances collected from multiple projects. The negative sample in vulnerability prediction datasets is usually either collected from vulnerability fixes (e.g. Li et al. (2018, 2022)), or from a small sample of clean methods (e.g. Chakraborty et al. (2022)). In this paper, we study whether the type of negative samples in vulnerability prediction datasets has an effect on the performance of models.

Many vulnerability prediction models that use code metrics as features rely on conventional machine learning classifiers such as Random Forest, Naïve Bayes, SVM, and others (Shin and Williams, 2008, 2013; Sultana et al., 2021). Conventional machine learning models are suitable when independent variables can be identified as opposed to using deep learning techniques where models do not necessarily rely on predefined independent variables (Janiesch et al., 2021). When the order of neighbouring elements in the data is important, such as in natural language processing, it becomes a challenge to identify meaningful independent variables to be used for conventional machine learning (Janiesch et al., 2021). In our previous work, we proposed using Abstract Syntax Tree (AST) n-grams (i.e. sets of Java AST nodes that define the low-level programming constructs used, as well as their order) as independent variables for a random forest model to predict software vulnerabilities (Al Debeyan et al., 2022). Our approach first transforms ASTs into a sequence of nodes. Then, we extract n-grams for the node sequences and use the n-grams as independent variables. The benefit of using AST n-grams as independent variables is that AST n-grams produce a dataset that contains clearly defined independent variables while holding valuable information about adjacent tokens and their order from ASTs which can be useful in identifying software vulnerabilities that spread across multiple adjacent AST tokens (Al Debeyan et al., 2022). In this paper, we collect static code metrics, which is a common approach followed by other researchers (Shin and Williams, 2013; Sultana et al., 2021), as well as AST n-grams, proposed in our recent paper (Al Debeyan et al., 2022), to produce software vulnerability datasets to answer the research questions posed.

Deep learning has assisted in making significant progress in software vulnerability prediction in recent years. The most recent cutting-edge models, such as SySeVR (Li et al., 2022) and LineVul (Fu and Tantithamthavorn, 2022), have achieved F1 scores of more than 90%. LineVul is a transformer-based function-level vulnerability prediction model with exceptional predictive performance (F-measure of 91% [9]). Steenhoek et al. (2023) have re-evaluated many vulnerability prediction models, including LineVul, using a range of datasets (including Devign (Zhou et al., 2019) and MSR (Fan et al., 2020)). In all datasets utilised by Steenhoek et al. (2023), LineVul was the only model that consistently achieved acceptable performance (> 60% F-measure). In this paper, we include LineVul because of these outstanding results.

2.2. The quality of vulnerability prediction datasets

Chakraborty et al. (2022) evaluated a selection of datasets used in some state-of-the-art software vulnerability prediction models such as VulDeePecker (Li et al., 2018), SySeVR (Li et al., 2022), Russell et al. (2018) and Devign (Zhou et al., 2019). Chakraborty et al. asked "how well do the state-of-the-art DL-based techniques perform in a real-world vulnerability prediction scenario?" and reported that the performance of the state-of-the-art models dropped by more than 50% on average compared to the results reported by the original papers. Chakraborty et al. investigated what causes such a precipitous performance drop and found that existing vulnerability prediction datasets suffer from multiple challenges, such as containing synthetic data, being labelled using a static analyser, or containing a balanced ratio of vulnerable to non-vulnerable instances, which the authors suggest does not represent

```
1  protected void request(String input){
2       String userInput = input;
        .
        .
        .
30
31      httpRequest(userInput);
32
33 }
```

Method 1: Positive Method

```
1  protected void request(String input){
2       String userInput = input;
        .
        .
        .
30      if(validate(userInput)){
31          httpRequest(userInput);
32      }
33 }
```

Method 2: Hard Negative Method

```
1 public Logger getLog() {
2     return LOG;
3 }
```

Method 3: Easy Negative Method

Fig. 2. Example of a vulnerable method (positive), a vulnerability fix (hard negative) and a random clean method (easy negative).

the real world. When a vulnerability prediction model is evaluated on a dataset, it is crucial that the test set represents the real world to have any hope of achieving similar performance results when the model is deployed in real settings. In this paper, we measure the change in the performance of vulnerability prediction models on test sets that follow different negative sampling strategies, including a test set of entire projects, which is a more realistic scenario of how vulnerability prediction models would be used.

2.3. The quality of negative sample

Classifiers are commonly used in machine learning applications to organise data into predefined classes or categories automatically. For example, a classifier might be trained on a dataset of animal images and used to predict which type of animal is in a new image (Valletta et al., 2017). A classifier typically works by learning the characteristics of each class from a training dataset and using that information to make predictions on new data. To demonstrate, for a classifier to learn what represents a cat, the dataset has to contain instances of images that represent a cat (positive sample), as well as images that do not represent a cat (negative sample). The quality of the negative sample has been studied in a variety of machine learning domains such as medicine (Liang et al., 2020), bioinformatics (Cheng et al., 2017), image processing (Robinson et al., 2020; Fan et al., 2022; Xuan et al., 2020) and video moment localisation (Zheng et al., 2022). Liang et al. (2020) evaluated the prediction of drug side effects with a refined negative sample selection strategy that uses the Random Walk with Restart (RWR) algorithm to find drugs with low probabilities of side effects. These drugs are then included in the negative sample. Liang et al. found that the refined strategy produces significantly higher performance in predicting drug side effects. Cheng et al. (2017) studied the effect of selecting high-quality negative samples on the efficiency of predicting protein-RNA interactions and reported an accuracy improvement of up to 204%. Xuan et al. (2020) argue that in computer vision, previous approaches in the literature do not focus on hard negatives, which are negative images that are similar to the positive sample, as they lead to bad local minima early on in training. However, Xuan et al. propose a simple modification to a standard loss function to fix bad optimisation behaviour with hard negative examples and demonstrate that the modification improves the current state-of-the-art results on datasets with high intra-class variance. Similar to the work done in the aforementioned fields, we study the effect of the negative sample in the field of software vulnerability prediction.

Fig. 2 shows a vulnerable method on the top left, a hard negative method on the top right, which is a patched version of the vulnerable method, and an easy negative method on the bottom. We can see that the patched version of the vulnerable method matches the vulnerable version in 31 of 33 lines of code, being harder to differentiate than an easy negative method.

2.4. Automated machine learning

Automated Machine Learning (AutoML) has recently received significant attention, and its state-of-the-art is (to a large extent) captured in recent surveys by He et al. (2021) and Talbi (2020). AutoML is the process of automating multiple machine learning tasks, such as preprocessing, model selection, and hyperparameter optimisation (Feurer et al., 2015). Therefore, it can be used to find an optimised tuned machine learning algorithm, decreasing the expertise required by its user. Given a dataset, an AutoML system can recommend a pipeline (sequence of tasks) to solve a machine-learning problem. Using AutoML enables us to establish whether the machine learning algorithm and hyperparameters vary according to different negative sampling strategies. We take advantage of a Python package named Auto-Sklearn that implements the principles of AutoML in a Python environment.

3. Methodology

This section describes the methodology to answer the research questions posed in three parts. First, we describe how we gathered eight different datasets (four AST n-grams datasets and four code metrics datasets) to answer RQ1 (see Section 3.1). The second part concerns applying AutoML to our datasets to answer RQ2 (see Section 3.2). Finally, in Section 3.3, we describe the methodology of how we apply appropriate splits to the datasets to answer RQ3 and RQ4.

3.1. Dataset gathering

To gather real-world samples of software vulnerabilities along with hard negatives and easy negatives, we took advantage of a vulnerability patch database published by Reis and Abreu (2021). The vulnerability patch database contains a list of software vulnerability patches on GitHub that include the CVE ID,[2] the project GitHub link, and the patch commit ID for various programming languages. Due to the maturity of Java code metrics tools compared to other languages, as well as the compatibility of the AST parser we used, the first step we took was to filter the vulnerability patches to include only those related to the Java programming language. After that, we used the commit ID from the project GitHub repository to identify the files that were changed to patch the vulnerability. From the changed files, we collect the changed Java methods. The version of the method before the commit is considered vulnerable, whereas the version after the commit is considered fixed or hard negative. We were able to collect 7165 vulnerable methods. To collect easy negative samples, we consider the latest version of each of the projects included in the dataset as the

[2] Common Vulnerabilities and Exposures unique identifiers for publicly known vulnerabilities in publicly released software packages. https://cve.mitre.org/.

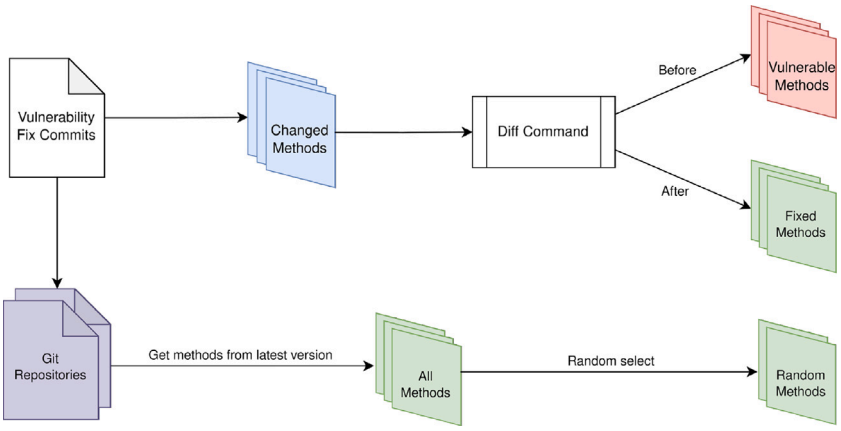

Fig. 3. The process of collecting vulnerable and non-vulnerable methods from Git fix commits.

Table 1
Distribution of vulnerable, hard negative and easy negative methods for the datasets used in the paper.

Dataset	Vulnerable	Hard negative	Easy negative
N-grams	6515	6467	3,236,762
Metrics	2836	2827	4,771,803
MSR	10,900	10,900	2,706,845

version that contains no known vulnerabilities. We randomly collected a sample of these versions and considered them easy negatives. Fig. 3 shows the full process of extracting vulnerable methods, vulnerability fixes, and random clean methods. Section 4 explains how we extract AST n-grams and software code metrics from all the methods collected.

To evaluate LineVul, we use the MSR (Fan et al., 2020) dataset used in LineVul's original paper (Fu and Tantithamthavorn, 2022). The MSR dataset is one of the largest vulnerability datasets. The dataset is collected from 348 open-source Github projects, which includes 91 different CWEs from 2002 to 2019, 188,636 C/C++ functions with a ratio of vulnerability functions of 5.7% (i.e., 10,900 vulnerable functions), and 5,060,449 LOC with a ratio of vulnerable lines of 0.88% (i.e., 44,603 vulnerable lines). While the MSR contains vulnerability fixes, LineVul's original evaluation does not include vulnerability fixes in the training and testing sets. To collect the clean methods from the latest versions of Git repositories, we follow the same approach in Fig. 3.

Table 1 shows the distribution of vulnerable, hard negative and easy negative methods for each dataset after removing duplicates. It is worth noting that the slight difference in the number of vulnerable and hard negative methods is due to multiple vulnerable methods having the same fixed version.

To study the effect of the negative sampling strategy on the performance of vulnerability prediction models, we collected four types of datasets, each with a different negative sampling strategy:

Only Hard Negatives: In addition to positive (vulnerable) methods, only includes hard negative methods (vulnerability fixes). The ratio of positive to hard negative to easy negative instances is 1:1:0.

One-to-One: A dataset that, in addition to positive (vulnerable) methods, includes one hard negative, as well as one easy negative (random clean) method for every positive (vulnerable) method. The ratio of positive to hard negative to easy negative instances is 1:1:1.

One-to-Five: A dataset that, in addition to positive (vulnerable) methods, includes one hard negative, as well as five easy negative methods for every positive method. The ratio of positive to hard negative to easy negative instances is 1:1:5. We specifically chose to include 5 easy negatives for every positive instance because we examined all projects included in the datasets and found the project with the lowest ratio of positive to easy negative instances was Jenkins[3] with a ratio of 1:5.

Only Easy Negatives: In addition to positive (vulnerable) methods, includes five easy negative methods for each positive method. The ratio of positive to hard negative to easy negative instances is 1:0:5.

We chose these four types of negative sampling strategies based on their use in the literature (Zhou et al., 2019; Li et al., 2018, 2022; Chakraborty et al., 2022). The NVD dataset (used in VulDeePecker (Li et al., 2018) and SySeVR (Li et al., 2022)) only includes hard negatives. ReVeal (Chakraborty et al., 2022) includes hard negatives and a subset of easy negatives (similar to One-to-One or One-to-Five) while Devign (Zhou et al., 2019) and Sultana et al. (2021) use datasets that include a subset of easy negatives (similar to Only Easy Negatives). The four types examine the models' behaviour across different ratios of hard and easy negatives. We publish all datasets used in this paper in our reproduction package (Al Debeyan, 2023) as wider adoption of reproducible research would be beneficial for empirical software engineering research (Madeyski and Kitchenham, 2017).

3.1.1. AST n-grams

We followed the process in our previous work (Al Debeyan et al., 2022) to convert Java methods into AST n-grams. For each Java method, we first parse the corresponding AST. After that, we traverse through the AST nodes using a depth-first search to get an AST node sequence. Finally, we transform the AST node sequence to n-grams where $0<n\leq 3$. We eliminate instances where the AST does not change between the vulnerable method and its fix for two reasons. The first reason is to remove vulnerable methods where the change does not affect the behaviour of the method, like adding comments or changing variable names. The second reason is that we would eliminate instances where they were labelled non-vulnerable but were later found to be vulnerable. We also remove duplicates as they can produce overly optimistic performance measures (Chakraborty et al., 2022). The total number of vulnerable methods in the AST n-grams datasets after removing duplicates and conflicts was 6515.

3.1.2. Code metrics

To gather code metrics for our Java dataset, we rely on the Understand[4] tool to collect 26 method-level metrics. The 26 code metrics are listed and defined in the online appendix included in our reproduction package (Al Debeyan, 2023). Similarly to what we did with AST n-grams, we removed conflicting instances and duplicates where the code metrics do not change between a vulnerable method and its fix. The total number of vulnerable methods in the code metrics datasets

[3] https://www.jenkins.io/
[4] https://www.scitools.com/features/

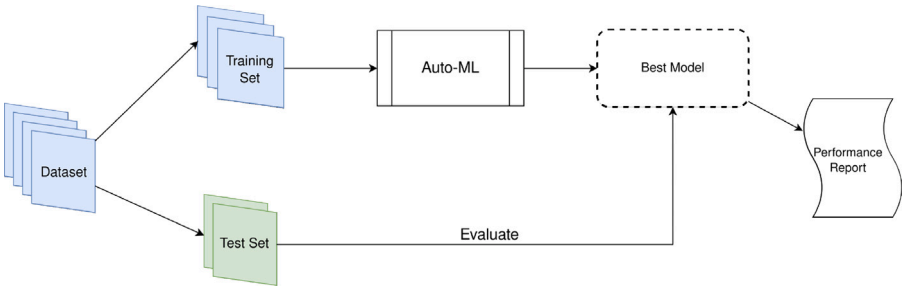

Fig. 4. The process of AutoML to automatically find the ML-tuned model with highest performance.

after removing duplicates and conflicts was 2836. There were fewer vulnerable methods in the code metrics datasets than in the AST n-grams because there were instances where the AST changed between a vulnerable method and its fix while there was no change in the metrics.

3.2. Automated machine learning

It is common practice in the field of vulnerability prediction to compare the performance of a proposed machine learning model with other state-of-the-art models in the literature (Zhou et al., 2019; Li et al., 2018, 2022; Chakraborty et al., 2022). However, models in the literature do not always follow the same negative sampling strategy. This section describes how we use AutoML to investigate whether the choice of machine learning algorithm remains the same throughout the four types of datasets we gather discussed in Section 3. If the machine learning algorithm changes depending on the negative sampling strategy, we can conclude that a comparison between models in the literature may not be reliable unless the negative sampling strategy for all models is the same.

Fig. 4 shows the methodology of how we used AutoML to find the model with the highest performance for each dataset. In the first step, we split the dataset into a training set, which is given to AutoML as input, and a test set to evaluate the model produced by AutoML. Following other researchers (Shu et al., 2022), we split the dataset 85:15 where 85% of the dataset is the training set, and 15% is the test set. Next, we run AutoML using the training set performing a 5-fold cross-validation on the training set to automatically find the machine learning model with the highest MCC. AutoML performs cross-validation on 16 different machine learning classifiers with hyperparameter tuning and returns the model that achieves the highest performance. The list of machine learning classifiers and their hyperparameters supported by AutoML are listed in the online appendix included in our reproduction package (Al Debeyan, 2023). We then test the model produced on our test set and report the performance of that model. It is important to note that the test set is never seen by AutoML during the comparison process. We isolate the test set to increase the generalising capability of the models.

3.3. Cross-evaluation

Part of our work in this paper is to investigate how a model that is trained on one type of negative sample behaves when tested on another type of negative sample (RQ3 and RQ4). To perform such experiments, we split the samples in our dataset into three different types: positive sample, hard negative sample, and easy negative sample as described in Section 3.1. After that, we split the positive sample 85:15, where 85% of the positive sample is used for training and 15% for testing. Then, we train two different models, one that is trained on the hard negative sample along with 85% of the positive sample, and another that is trained on the easy negative sample along with 85% of the positive sample. Finally, we test the first model on easy negative samples along with 15% of the positive sample, and we test the second model on hard negative samples along with 15% of the positive sample. The choice of the machine learning algorithm and hyperparameter tuning for each model is based on AutoML's highest MCC model for the matching training dataset. Fig. 5 shows the cross-evaluation process.

To answer RQ5, we collect all vulnerable methods in every project included in the dataset along with all hard negative (vulnerability fix) methods as well as all easy negative methods. Collecting every method would ensure we include the entire set of methods for each of the three types. Typically, in a real-world scenario where a vulnerability prediction model would be used to examine an entire project for vulnerabilities, the model would have to analyse every method in the project where each method would fall under one of the three types (vulnerable, hard or easy negative).

After collecting the real-world dataset, we split the dataset 85% for training and 15% for testing. The test set remains the same for all models while we run AutoML using different ratios of positives to easy negatives to asses which ratio results in the best performance on the real-world test set. We also run AutoML with the same ratios but without hard negatives to measure the effect of adding hard negatives in the training set. We include ratios starting with a training set with no easy negatives (1:0) and increasing the ratio by one up to 20 easy negatives for every vulnerable method (1:20). We implement the process for AST N-grams and software code metrics resulting in 84 training sets and two real-world test sets. With every training set, we run AutoML on the training set, performing a 5-fold cross-validation and finding the algorithm and hyperparameters that achieve the highest MCC. Using the trained model selected by AutoML, we reevaluate the model on the real-world test set and report the Precision, Recall, F-measure and MCC.

3.4. Performance measures

We compare performance levels using two measures, F-measure and Matthews correlation coefficient (MCC). We report the F-measure due to its prominence in other vulnerability prediction models in the literature. However, Yao and Shepperd (2020) suggest not using F-measure when evaluating models on imbalanced datasets. Yao et al. showed that when dealing with imbalanced datasets, MCC is suitable for handling such cases. Since all our datasets are imbalanced, we include MCC in our evaluation. We also include Precision, Recall and AUC in our reproduction package (Al Debeyan, 2023).

Precision, Recall, F-measure and MCC rely on true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN).

Precision, Recall and F-measure are defined as:

$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN} \quad F_1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

Matthews correlation coefficient is defined as:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FN)(TP + FP)(TN + FN)(TN + FP)}}$$

3.5. Statistical analysis

To measure the statistical significance of the difference in performance between models, we calculate the effect size using Cliff's δ (Cliff, 1993). Cliff's δ is non-parametric and suggested by Kitchenham et al.

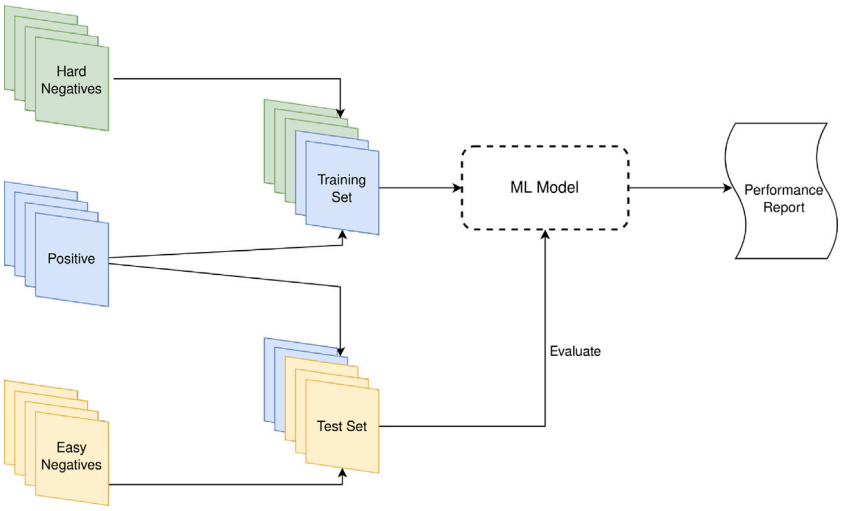

Fig. 5. The process of training a model using one negative sampling strategy and testing on the other.

Table 2
Performance results in both Python and R (in brackets) for 8 datasets.

Metric	Dataset	Random forest	Naïve Bayes	SVM	Gradient boosting
F-measure	N-grams Only Hard	0.73 (0.68)	0.59 (0.61)	0.56 (0.58)	0.61 (0.56)
	N-grams 1–1	0.70 (0.63)	0.32 (0.32)	0.09 (0.04)	0.26 (0.28)
	N-grams 1–5	0.70 (0.57)	0.34 (0.33)	0.10 (0.04)	0.21 (0.24)
	N-grams Only Easy	0.82 (0.79)	0.49 (0.47)	0.59 (0.56)	0.57 (0.53)
	Metrics Only Hard	0.46 (0.47)	0.29 (0.58)	0.67 (0.49)	0.51 (0.46)
	Metrics 1–1	0.38 (0.33)	0.15 (0.43)	0.00 (NaN)	0.19 (0.20)
	Metrics 1–5	0.31 (0.23)	0.00 (0.34)	0.00 (NaN)	0.11 (0.06)
	Metrics Only Easy	0.71 (0.68)	0.01 (0.43)	0.01 (0.55)	0.63 (0.61)
MCC	N-grams Only Hard	0.46 (0.36)	−0.03 (0.01)	−0.03 (−0.01)	0.13 (0.07)
	N-grams 1–1	0.57 (0.47)	0.11 (0.11)	0.01 (0.00)	0.18 (0.16)
	N-grams 1–5	0.66 (0.54)	0.22 (0.21)	0.12 (0.05)	0.23 (0.23)
	N-grams Only Easy	0.80 (0.76)	0.38 (0.36)	0.56 (0.53)	0.54 (0.49)
	Metrics Only Hard	−0.08 (−0.08)	0.00 (0.01)	0.00 (0.00)	0.00 (0.01)
	Metrics 1–1	0.10 (0.09)	0.00 (0.18)	−0.02 (0.00)	0.06 (0.09)
	Metrics 1–5	0.22 (0.17)	0.02 (0.25)	0.00 (0.00)	0.10 (0.08)
	Metrics Only Easy	0.66 (0.64)	0.05 (0.39)	0.04 (0.52)	0.59 (0.57)

for small sample sizes (Kitchenham et al., 2017). The effect size $|\delta| > 0.428$ is considered large, $0.276 \leq |\delta| < 0.428$ is medium, $0.112 \leq |\delta| < 0.276$ is small, and $|\delta| < 0.112$ is negligible.

4. Evaluation and analysis

In this paper, we investigate the effect of the negative sampling strategy on the performance of software vulnerability prediction models to answer the following research questions:

RQ1: Does the strategy of collecting negative samples affect the performance of ML models?

Table 2 shows the performance of four machine learning classifiers based on the datasets collected. We chose to evaluate a Random Forest (RF) model, a Naïve Bayes model, a Support Vector Machines (SVM) model and a Gradient Boosting Machine (GBM) model. We decided to choose these models due to their wide use in machine learning and specifically vulnerability prediction (Chakraborty et al., 2022; Sultana et al., 2021). To answer RQ1, we are not looking for the highest-performing model but rather studying the difference in model performance based on the negative sampling strategy. For that reason, we used the default hyperparameters for each model in the Python package Scikit-learn[5] and the default hyperparameters in the R package mlr3 (Lang et al., 2019). The default hyperparameters between the two packages differ which can result in a difference in the performance of the same machine learning algorithm.

The results in Table 2 show that for F-measure, the model performs best 4 out of 8 times when we include easy negatives and exclude hard negatives from the dataset using Scikit-learn and 4 out of 8 times when using mlr3. However, as we describe in Section 3.4, MCC provides a better comparison measurement due to the nature of the datasets having different vulnerable to non-vulnerable ratios. For MCC, the model performs best for all models in both scikit-learn and mlr3 when we include only easy negatives in the dataset. Also, in most cases, the model performs worst in MCC when the dataset only includes hard negatives without including easy negatives. These results are in line with our expectation as it would be harder for a model to distinguish between a positive (vulnerable) method and a hard negative (vulnerability fix) since the two methods would have a lot of similarities, with the exception of a small change to fix the vulnerability (as illustrated in Fig. 2).

Table 3 shows the effect size and statistical significance of the results in Table 2. To calculate the effect size, we define the baseline as the dataset that includes only hard negatives and compare the results of the remainder of the datasets to the baseline. For each of the machine learning models, we can see that there exists at least one dataset that has a large effect size ($|\delta| \geq 0.428$) with a p-value < 0.05. The results from Tables 2 and 3 lead to the finding labelled Finding 1. Other performance measures such as precision, recall and area under the curve (AUC) are reported in our reproduction package (Al Debeyan, 2023).

[5] https://scikit-learn.org/stable/

Table 3
Effect size (Cliff's δ) and p-value for the difference in performance compared to the baseline datasets (N-grams only hard and metrics only hard).

Metric	Dataset	Random forest	Naïve Bayes	SVM	Gradient boosting
F-measure	N-grams 1–1	Lg**	Lg**	Lg**	Lg**
	N-grams 1–5	Lg**	Lg**	Lg**	Lg**
	N-grams Only Easy	Lg**	Lg**	Lg**	Lg**
	Metrics 1–1	Lg**	Sm***	Lg**	Lg**
	Metrics 1–5	Lg**	Md***	Lg**	Lg**
	Metrics Only Easy	Lg**	Ng***	Lg**	Lg**
MCC	N-grams 1–1	Lg**	Lg**	Lg*	Lg**
	N-grams 1–5	Lg**	Lg**	Lg**	Lg**
	N-grams Only Easy	Lg**	Lg**	Lg**	Lg**
	Metrics 1–1	Lg**	Sm***	Ng***	Lg**
	Metrics 1–5	Lg**	Md***	Sm***	Lg**
	Metrics Only Easy	Lg**	Lg*	Lg**	Lg**

Effect Size: Large(Lg) $|\delta| \geq 0.428$, Medium (Md) $0.276 \leq |\delta| < 0.428$, Small (Sm) $0.112 \leq |\delta| < 0.276$, Negligible (Ng) $|\delta| < 0.112$
* Statistical Significance: $p < 0.05$
** Statistical Significance: $p < 0.01$
*** Statistical Significance: $p \geq 0.05$

Table 4
Highest MCC machine learning algorithm per dataset.

Dataset	Model	F-measure	MCC
N-grams Only Hard	SVM	0.69	0.50
N-grams 1–1	K-Nearest Neighbour	0.70	0.59
N-grams 1–5	K-Nearest Neighbour	0.69	0.67
N-grams Only Easy	Random Forest	0.82	0.80
Metrics Only Hard	SVM	0.39	0.13
Metrics 1–1	Extra Trees	0.62	0.39
Metrics 1–5	Gradient Boosting	0.56	0.51
Metrics Only Easy	Gradient Boosting	0.81	0.77

Finding 1: The negative sampling strategy significantly affects the performance of machine learning models that perform software vulnerability prediction. A model that performs well on one negative sampling strategy may not achieve the same performance results when evaluated on datasets that use other negative sampling strategies.

RQ2: Does the strategy of collecting negative samples affect the choice of the ML model and/or hyperparameter tuning?

To answer RQ2, using the same eight datasets gathered to answer RQ1, we used the Python package Auto-sklearn.[6] Auto-sklearn is a Python implementation of AutoML. For each dataset, we ran Auto-sklearn for 10 h, performing cross-validation and comparing MCC to find the model with the highest performance. We set the memory limit to 4 GB and used 10 threads. We ran all Auto-ML experiments on The High-End Computing facility at Lancaster University. Table 4 shows the results of AutoML on the eight datasets gathered.

For AST n-grams datasets, we see that an SVM model scored the highest MCC for the dataset that includes only hard negatives, while a random forest model scored the best MCC on the dataset with only easy negatives. For AST n-gram datasets that included a mixture of easy and hard negatives (1–1 and 1–5), K-Nearest Neighbour models scored the best MCC. For code metric datasets, an SVM model was selected as the highest performing for the dataset including only hard negatives. For the dataset where there is 1 easy negative and 1 hard negative for every positive instance, extra trees was the selected model, while a Gradient Boosting model scored the highest MCC for the dataset including 1 hard negative and 5 easy negatives as well as the dataset that only includes easy negatives. These results lead to the finding labelled Finding 2.

[6] https://automl.github.io/auto-sklearn/master/

Finding 2: The model and/or hyperparameter tuning for software vulnerability prediction with the highest performance changes depending on the negative sampling strategy. Therefore, we can confirm that the superior performance of one vulnerability prediction model over another can be claimed only in a scenario that matches the ratio of easy/hard negatives in the test set.

RQ3: How would a vulnerability prediction model trained on hard negative methods perform in predicting easy negative methods?
RQ4: How would a vulnerability prediction model trained on easy negative methods perform in predicting hard negative methods?

We address RQ3 and RQ4 together because they both assess the performance change when the training set and the evaluation set follow different negative sampling strategies. From Tables 2 and 4, we can see that the more easy negatives are added to the data set, the better a model performs in predicting software vulnerabilities. Some models in the literature use datasets that only include hard negatives (vulnerability fixes) in their datasets (Li et al., 2018, 2022). However, when such models are used in practice, they need to successfully identify easy negatives as well, which are examples on which the model was not trained. It is important to study how a model performs when trained on hard negatives and evaluated on easy negatives and vice versa, how a model performs when trained on easy negatives and evaluated on the hard ones.

To answer RQ3 and RQ4, we separated instances in AST n-grams and code metrics representations into three different types: positive, hard negative, and easy negative instances. We followed the process described in Section 3.3 to create two training sets and two test sets for each representation. After that, we use the model selected by AutoML depending on the negative sampling strategy. For LineVul, we train the model on a training set that includes one type of negative instances and evaluate the model on a test set that includes the other type of negative instances. We reproduced the results of LineVul on the original MSR dataset to ensure LineVul is set up correctly. Table 5 shows the *precision*, *recall*, and *F-measure* reported by LineVul compared to our reproduction results. Our reproduction results are within a 2% difference compared to the original paper in general.

Table 6 shows the results when the models are trained on hard negatives and evaluated on easy negatives.

For AST n-grams, the SVM model was tuned based on the AutoML results for the dataset containing only hard negatives. The F-measure dropped from 0.76 in Table 4 to 0.07. MCC also dropped from 0.52 to 0.03. For code metrics, the SVM model was also tuned based on the

Table 5
LineVul reproduction on the original dataset (MSR).

Original results			Reproduced results		
Precision	Recall	F1	Precision	Recall	F1
0.97	0.86	0.91	0.96	0.88	0.92

Table 6
F-measure and MCC when model is trained on hard negatives and tested on easy negatives.

Dataset	Model	F-measure	MCC
N-grams	SVM	0.07	0.03
Metrics	SVM	0.05	−0.05
MSR	LineVul	0.01	0.02

Table 7
F-measure and MCC when model is trained on easy negatives and tested on hard negatives.

Dataset	Model	F-measure	MCC
N-grams	Random Forest	0.25	0.07
Metrics	Gradient Boosting	0.14	−0.06
MSR	LineVul	0.68	0.22

AutoML results for the dataset containing only hard negatives. The F-measure for the code metrics dataset dropped from 0.39 to 0.05. MCC also dropped from 0.13 to −0.05. For LineVul, F-measure dropped from 0.92 to 0.01, while MCC dropped from 0.92 to 0.02. These results lead to the finding labelled Finding 3.

> **Finding 3:** The performance of a vulnerability prediction model trained on hard negatives drops when evaluated on easy negatives. This means that a model that is only trained on hard negatives may not be suitable for use in practice.

Table 7 shows the results when models are trained on easy negatives and evaluated on hard negatives. For AST n-grams, a random forest model was tuned based on the results of AutoML for the dataset containing only easy negatives. The F-measure dropped from 0.82 to 0.25, and MCC dropped from 0.79 to 0.03. For code metrics, a Gradient Boosting model was tuned based on the results of AutoML for the dataset containing only easy negatives. The F-measure dropped from 0.81 to 0.14, and MCC dropped from 0.77 to −0.06. For LineVul, F-measure dropped from 0.92 to 0.68, while MCC dropped from 0.92 to 0.22. These results lead to the finding labelled Finding 4.

> **Finding 4:** The performance of a vulnerability prediction model trained on easy negatives drops when evaluated on hard negatives. This means that a model that is only trained on easy negatives may not be suitable to use in scenarios where all methods are prone to vulnerabilities (vulnerable or hard negatives).

RQ5: What ratio of positives/easy negatives in the training set provides the highest performance on test sets that include entire projects? To answer RQ5, using the dataset collected that includes methods from entire projects, we used Auto-sklearn on training sets with different positive to easy negative ratios. We ran Auto-sklearn for 10 h on each model, setting the memory limit to 20 GB and using 5 threads.

Fig. 6 shows the precision and recall of the six types of models (AST N-grams with hard negatives, AST N-grams without hard negatives, Metrics with hard negatives, Metrics without hard negatives, LineVul with hard negatives, and LineVul without hard negatives) evaluated on three test sets that represent entire projects. For precision, we see that for all six types of models, precision is at its lowest when the training set only includes hard negatives. Precision increases as we add more easy negatives in the training set. On the other hand, recall tends to be more stable throughout different easy negative ratios with a slight decrease with the exception of training sets with no easy negatives where the three models (N-grams, metrics and LineVul) have completely different behaviour.

Fig. 7 shows the F-measure and MCC of the six types of models. In all six types of models, we see a trend where both F-measure and MCC increase as the positive to easy negative ratio decreases reaching a plateau when the easy negative ratio is 1:15. For AST N-grams and LineVul, we see models trained on datasets that include hard negatives have better F-measure and MCC compared to models trained on datasets without hard negatives. On the other hand, for code metrics, while all models perform poorly in terms of F-measure and MCC, models trained on datasets without hard negatives perform slightly better in both F-measure and MCC. it is worth noting that as we decrease the ratio of positives to easy negatives from 1:1 to 1:20, we create a larger imbalance ratio between the positive and the negative samples. We have incorporated the oversampling technique known as SMOTE as well as Random Over-Sampling (ROS) on the N-grams and metrics models on all training sets twice, once while including hard negatives and once without hard negatives. While SMOTE is a well-known oversampling technique, Yang et al. (2023) report that ROS results in the best increase in performance for vulnerability prediction models. Our findings indicate that neither SMOTE nor ROS had a positive impact on the models' performance when compared to models without oversampling. These results have been included in our online appendix included in our reproduction package (Al Debeyan, 2023). The results in Figs. 6 and 7 lead to Finding 5 and Finding 6.

> **Finding 5:** As the ratio of positive to easy negatives decreases in the training set, the precision of a vulnerability prediction model increases, resulting in an overall increase in F-measure and MCC reaching a plateau at the ratio 1:15. This means that models perform best assessing entire projects when the number of easy negatives in the training set is 15 times the number of vulnerable instances.

> **Finding 6:** Although the code metrics used in this paper reached acceptable performance levels on some vulnerability prediction datasets (see Table 4), these metrics models performed poorly when evaluated on a test set that represents entire projects. This means that models should be evaluated on test sets representing entire projects before deciding whether they are suitable to be used by developers.

5. Discussion

Our goal was to answer the five research questions posed in Section 1 related to the effects of different strategies to collect negative samples in datasets used to train and evaluate software vulnerability prediction models.

Our results suggest that the strategy of selecting a negative sample strongly affects the performance of ML models (RQ1 and Finding 1), as well as the choice of the highest performing ML models and their hyperparameters whose values control the learning process (RQ2 and Finding 2).

Despite the effect that the negative sampling strategy seems to have on performance, to the best of our knowledge, the effect of the

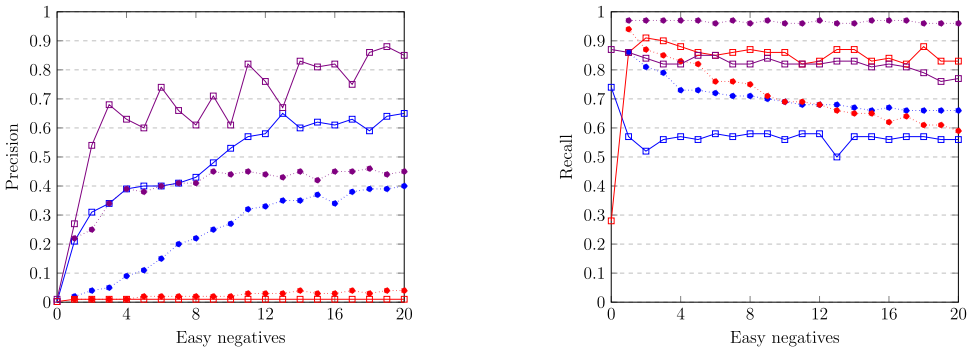

Fig. 6. Precision and recall of AST N-grams and code metrics models trained on datasets with different numbers of easy negatives for every positive instance.

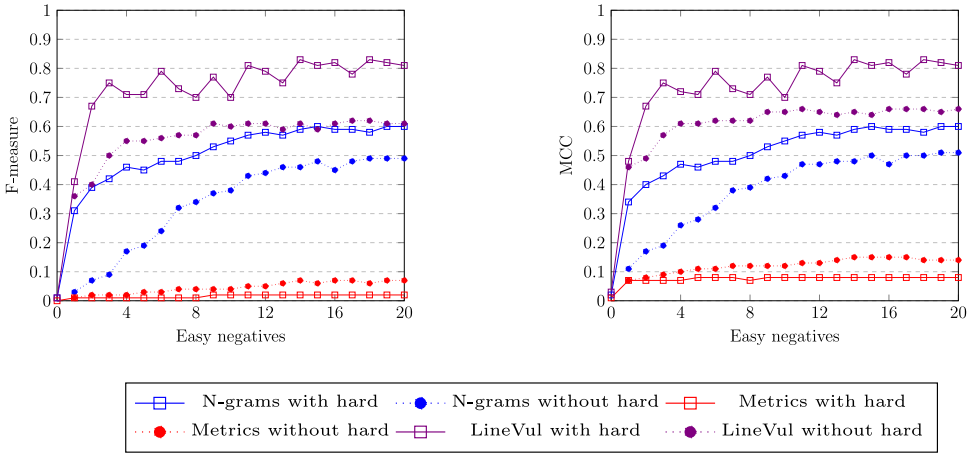

Fig. 7. F-measure and MCC of AST N-grams and code metrics models trained on datasets with different numbers of easy negatives for every positive instance.

negative sampling strategy has not been studied previously. Therefore, to analyse the effect even further, in subsequent research questions, we analysed how vulnerability prediction models trained on hard negative methods perform in predicting easy negative methods (RQ3), as well as the opposite, i.e. how vulnerability prediction models trained on easy negative methods perform in predicting hard negative methods (RQ4). This analysis enabled a better understanding of the impact that different strategies have on the performance of the models, especially in settings that are different from the ones in which the models were trained. We also try to explain the observation by Chakraborty et al. (2022) that the performance of state-of-the-art models drops in a real-world vulnerability prediction scenario. We found (see Findings 3 and 4) that the performance of vulnerability prediction models trained on one kind of negative sample (easy or hard negatives) drops when evaluated on another kind of negative sample (hard or easy negatives, respectively). The drop in performance is also highlighted by Chakraborty et al. when pre-trained models were evaluated on a new dataset. The pre-trained models were trained on datasets with negative samples that contained different distributions of hard vs. easy negatives compared to the evaluation dataset. Further research is needed to fully explain the phenomena observed by Chakraborty et al. (2022).

Knowing the influence of the negative sampling technique on vulnerability prediction model performance (Finding 1), we evaluated alternative ratios of easy negatives in the training set that provide the best performance in a more realistic scenario of assessing entire projects (RQ5). We discovered that, up to a point, the more easy negative samples in the training set, the better the model performs on a test set that represents entire projects. However, the performance advantage reaches a point where increasing the size of the easy negative sample yields no performance gain (Finding 5). We also demonstrated the importance of evaluating models on test sets that include entire projects' code base to ensure the models' adequacy for use in real-world scenarios (Finding 6).

Summary of Findings 1-6: The findings of our study show that the strategy of selecting a negative sample has non-negligible effects not only on the performance of machine learning models but also on the selection of the models (including their hyperparameters) and external validity of the vulnerability prediction models, i.e., how they perform in different (e.g., real-world) settings and applications.

Implication of Findings 1-6: A vulnerability prediction model's superior performance in one dataset does not necessarily translate into a similar advantage in real-world scenarios. Vulnerability prediction models should only be recommended for use in practice if they perform well on a test set replicating the ratio of easy to hard negative instances found in real-world scenarios.

We suggest that different negative sampling strategies can serve different purposes depending on what researchers are trying to achieve. For example, when a vulnerability prediction model is trained on a dataset that only includes easy negatives, i.e., random methods from projects, the model performs well predicting vulnerability-prone methods without differentiating between the vulnerable or the fixed version of the method. This kind of prediction can be useful for developers to narrow down the search space and manually examine the method to establish whether it contains the necessary checks to prevent a vulnerability. On the other hand, vulnerability prediction models that

are trained on a dataset which only includes hard negatives, i.e. the fixed version of vulnerable methods, are useful for developers when they already know a method is vulnerability-prone and want to automatically assess whether the method includes the necessary checks to prevent the vulnerability. We suggest that fellow researchers evaluate vulnerability prediction models on a test set that includes entire projects while training the model on training sets that include different ratios of easy/hard negatives to maximise the performance of the model in a real-world setting.

6. Related work

To our knowledge, measuring the effect of the strategy to collect the negative sample on vulnerability prediction models has not been previously studied or discussed in the literature. Even in the more mature field of defect prediction, we could not find papers that discuss the impact of the negative sample. The reason is that in defect prediction, researchers mainly evaluate models on an entire project due to the larger sample size of defects compared to vulnerabilities. For that reason, defect prediction datasets would include both hard and easy negatives by default.

Chakraborty et al. studied the characteristics of software vulnerability datasets used in the literature for vulnerability prediction (Chakraborty et al., 2022). Chakraborty et al. discussed the issue of existing datasets being balanced where only vulnerability fixes are included (hard negatives). They mitigate the balanced dataset problem by adding clean methods that were not related to vulnerabilities (easy negatives) by including all unchanged methods in a fix commit. We suggest that doing so is a better approach than only including vulnerability fixes. However, methods that are not changed in one fix commit can be labelled as vulnerable in later fix commits, and that would create conflicts in the dataset. We suggest that sampling the easy negatives from the projects' latest versions where there are no known vulnerabilities eliminates conflicting instances in the dataset. Moreover, Chakraborty et al. re-evaluated multiple vulnerability prediction models in the literature on their dataset which fixes multiple issues with current datasets such as containing synthetic data, being balanced or labelled using a static analyser. However, the re-evaluation did not address to what extent each of the issues raised contributes to the decrease in performance.

Croft et al. (2023) studied data quality in vulnerability prediction datasets. Croft et al. examined four state-of-the-art vulnerability prediction datasets and inspected the labelling accuracy, uniqueness of instances, consistency of labels and completeness of code in each dataset. Croft et al. report that 20%–71% of labels were inaccurate in real-world datasets, and 17%–99% of data points were duplicated. Although the work done by Croft et al. is essential to the quality of vulnerability datasets, they only address the quality of the positive sample. Our work complements the work of Croft et al.

Garg et al. (2022) proposed a new approach to train vulnerability prediction models using *Realistic Training Data Settings*. In contrast to *Clean Training Data Settings*, where the component's labelling information (vulnerable/non-vulnerable) is always available regardless of time, Realistic Training Data Settings require only vulnerability labels that are available at training time to be used for training the prediction models. For instance, in Realistic Training Data Settings, at a given time t, only the vulnerabilities known at time t should be available for training. Although the model presented by Garg et al. showed promising MCC levels, the model only works on a file level rather than a method level leaving security engineers with a larger search area to further inspect.

VulDeePecker is one of the state-of-the-art vulnerability prediction models (Li et al., 2018). The model achieves promising prediction performance, reaching an F-measure of 86.6% and 95% predicting buffer error vulnerabilities and memory management vulnerabilities, respectively. However, as Chakraborty et al. report, the datasets used to evaluate the model contain up to 80% of duplicate instances (Chakraborty et al., 2022). Such duplicates create bias in model evaluation since the model is tested on instances it has already been trained on. Moreover, after we examined the datasets, we found that the datasets only include vulnerability fixes (hard negatives) which makes the evaluation results not generalisable to real-world scenarios. When VulDeePecker was re-evaluated on a dataset more representative of a real-world scenario, the model plummeted in performance to an F-measure of 12% (Chakraborty et al., 2022). Other state-of-the-art models in the literature, such as SySeVR (Li et al., 2022), Russell et al. (2018), and Zhou et al. (2019) have all suffered a similar drop in performance when evaluated on a dataset more representative of real-world scenarios.

Sultana et al. (2021) presented a vulnerability prediction model that uses code metrics to predict software vulnerabilities in Java projects. The model was evaluated on Tomcat,[7] Apache CXF[8] and Stanford SecuriBench[9] achieving an F-measure between 75% and 85%. However, looking at the datasets used for evaluation, the percentage of hard negatives is between 0.17% and 1.43%. As the results in this paper suggest, these low percentages of hard negatives can over-optimise the performance results of vulnerability prediction models compared to the results they may achieve when deployed and used to assess entire projects.

Yang et al. (2023) assessed the impact of data sampling for the data imbalance problem in deep learning-based vulnerability prediction models. While Yang et al. found that oversampling techniques, specifically random oversampling, improve the performance of vulnerability prediction models, our experiments with oversampling showed no advantage over using original datasets.

7. Threats to validity

7.1. Internal validity

Multi-function Vulnerabilities At times, a software vulnerability can affect multiple methods instead of just one. Some of these methods may not be vulnerable when used individually. Since we consider all changed methods in a fix commit to be vulnerable, such methods would be labelled as vulnerable in the process. One way to overcome multi-function vulnerabilities is to change the level of granularity to the level of a file instead of a method. We suggest that using method-level granularity has more benefits than the drawback of multi-method vulnerabilities.

Refactoring Non-vulnerable Methods We gathered vulnerable methods for our dataset by tracing changed methods from GitHub commits that fixed vulnerabilities. However, during a vulnerability fix, developers may change the structure of methods by adding comments or changing the names of variables. Such methods should not be labelled as vulnerable as they are not related to the vulnerability the commit fixes. To eliminate such methods, we only include the changed methods that differ in their ASTs before and after the fix. Because the way we parse the AST disregards variable names and comments, such methods not related to the vulnerability are minimised in our dataset.

Contaminated Easy Negatives To gather the easy negative sample, we consider the methods from the latest version of each project in the dataset as easy negatives since the latest version does not contain any known vulnerabilities. When doing so, vulnerability fixes (hard negatives) would automatically be included in the sample as well, which contaminates the easy negative sample. However, when we manually checked the percentage of hard negatives to the overall number of methods in the latest version of each project, we found that the number is less than 2% and, therefore, would not have a great effect on the results. Furthermore, in this paper, we focus on one type of hard

[7] https://tomcat.apache.org/
[8] https://cxf.apache.org/
[9] https://suif.stanford.edu/~livshits/securibench/stats.html

negatives (vulnerability fixes). We realise that there can be other types of methods that can be classified as hard negatives. Future work should focus on techniques to identify and extract other types of hard negative methods.

7.2. External validity

Randomness of AutoML The AutoML model we used relies on random model selection and model configurations over a specified period of time. The random element of AutoML means that for every run, AutoML has a different order for model selection and tuning. For that reason, we chose to run AutoML for 10 h on each dataset using 10 threads, which is 50 times the resources set as the default. Moreover, the purpose of the AutoML experiments was not to find the absolute best tuning for each dataset but rather to show that the choice of model and tuning changes depending on the strategy of the negative sampling used.

8. Conclusion

We evaluated the effect of the negative sampling strategy in datasets on the performance of vulnerability prediction models. We gathered four types of datasets using two different code representations (code metrics and AST), making a total of eight datasets that differ in the way the negative sample is collected. We then evaluated four machine learning classifiers on the datasets to measure the difference in performance based on the negative sampling strategy. We found that the negative sampling strategy seems to affect the performance of a machine learning model predicting software vulnerabilities. We also ran AutoML on all eight datasets to examine if the highest-performing machine learning algorithm changes based on the negative sampling strategy. We found that the choice of the machine learning algorithm changes based on the negative sampling strategy used in a dataset. We also showed that a model that is trained on hard negatives does not perform as well predicting easy negatives and vice versa. Finally, we showed that, up to a certain level, models perform better in a real-world setting when they are trained on datasets that include higher ratios of easy negatives. Our findings suggest that researchers should evaluate vulnerability prediction models on test sets that represent entire projects to decide the fitness of the model to be used in practice.

CRediT authorship contribution statement

Fahad Al Debeyan: Conceptualization, Data curation, Methodology, Resources, Software, Writing – original draft, Writing – review & editing. **Lech Madeyski:** Data curation, Methodology, Supervision, Validation, Writing – review & editing. **Tracy Hall:** Conceptualization, Methodology, Supervision, Writing – review & editing. **David Bowes:** Methodology, Writing – review & editing.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Lech Madeyski reports financial support was provided by Engineering and Physical Sciences Research Council (EPSRC).

Data availability

I have shared the link to the replication package that includes the data.

Acknowledgements

Lech Madeyski worked on the paper during his research internship at Lancaster University at the invitation of Prof. Tracy Hall. Lech Madeyski was partially funded by an Engineering and Physical Sciences Research Council (EPSRC), UK grant EP/S005730/1. Calculations in R have been carried out using resources provided by Wroclaw Centre for Networking and Supercomputing (http://wcss.pl), grant No. 578.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.jss.2024.112003.

References

Al Debeyan, F., 2023. Reproduction package for the paper "The Impact of Hard and Easy Negative Training Data on Vulnerability Prediction Performance". Zenodo, http://dx.doi.org/10.5281/zenodo.8426023.

Al Debeyan, F., Hall, T., Bowes, D., 2022. Improving the performance of code vulnerability prediction using abstract syntax tree information. In: Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering. In: PROMISE 2022, Association for Computing Machinery, New York, NY, USA, pp. 2–11.

Chakraborty, S., Krishna, R., Ding, Y., Ray, B., 2022. Deep learning based vulnerability detection: Are we there yet? IEEE Trans. Softw. Eng. 48 (9), 3280–3296.

Cheng, Z., Huang, K., Wang, Y., Liu, H., Guan, J., Zhou, S., 2017. Selecting high-quality negative samples for effectively predicting protein-RNA interactions. BMC Syst. Biol. 11 (2), 9.

Cliff, N., 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol. Bull. 114 (3), 494.

Croft, R., Babar, M.A., Kholoosi, M.M., 2023. Data quality for software vulnerability datasets. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 121–133.

Fan, J., Li, Y., Wang, S., Nguyen, T.N., 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories. MSR '20, Association for Computing Machinery, New York, NY, USA, pp. 508–512. http://dx.doi.org/10.1145/3379597.3387501.

Fan, Z., Wei, Z., Li, Z., Wang, S., Huang, X.-J., Fan, J., 2022. Negative sample is negative in its own way: Tailoring negative sentences for image-text retrieval. In: Findings of the Association for Computational Linguistics. NAACL 2022, pp. 2667–2678.

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M., Hutter, F., 2015. Efficient and robust automated machine learning. In: Advances in Neural Information Processing Systems, vol. 28, pp. 2962–2970.

Fu, M., Tantithamthavorn, C., 2022. Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 608–620.

Garg, A., Degiovanni, R., Jimenez, M., Cordy, M., Papadakis, M., Le Traon, Y., 2022. Learning from what we know: How to perform vulnerability prediction using noisy historical data. Empir. Softw. Eng. 27 (7).

Gkortzis, A., Mitropoulos, D., Spinellis, D., 2018. Vulinoss: a dataset of security vulnerabilities in open-source systems. In: Proceedings of the 15th International Conference on Mining Software Repositories. pp. 18–21.

He, X., Zhao, K., Chu, X., 2021. AutoML: A survey of the state-of-the-art. Knowl.-Based Syst. 212, 106622.

Janiesch, C., Zschech, P., Heinrich, K., 2021. Machine learning and deep learning. Electron. Mark. 31 (3), 685–695.

Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R., 2013. Why don't software developers use static analysis tools to find bugs? In: 2013 35th Intern Confer on Software Engineering. ICSE, IEEE, pp. 672–681.

Jovanovic, N., Kruegel, C., Kirda, E., 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In: 2006 IEEE Symposium on Security and Privacy. S&P'06, IEEE, p. 6.

Kitchenham, B., Madeyski, L., Budgen, D., Keung, J., Brereton, P., Charters, S., Gibbs, S., Pohthong, A., 2017. Robust statistical methods for empirical software engineering. Empir. Softw. Eng. 22, 579–630.

Lang, M., Binder, M., Richter, J., Schratz, P., Pfisterer, F., Coors, S., Au, Q., Casalicchio, G., Kotthoff, L., Bischl, B., 2019. mlr3: A modern object-oriented machine learning framework in R. J. Open Source Softw.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2022. SySeVR: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secure Comput. 19 (4), http://dx.doi.org/10.1109/TDSC.2021.3051525.

Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y., 2018. VulDeePecker: A deep learning-based system for vulnerability detection. In: Network and Distributed Systems Security (NDSS) Symposium. San Diego, CA, USA.

Liang, H., Chen, L., Zhao, X., Zhang, X., 2020. Prediction of drug side effects with a refined negative sample selection strategy. Comput. Math. Methods Med. 2020.

Liu, S., Lin, G., Han, Q.-L., Wen, S., Zhang, J., Xiang, Y., 2020. DeepBalance: Deep-learning and fuzzy oversampling for vulnerability detection. IEEE Trans. Fuzzy Syst. 28 (7), 1329–1343.

Madeyski, L., Kitchenham, B., 2017. Would wider adoption of reproducible research be beneficial for empirical software engineering research? J. Intell. Fuzzy Systems 32 (2), 1509–1521. http://dx.doi.org/10.3233/JIFS-169146.

Morrison, P., Herzig, K., Murphy, B., Williams, L., 2015. Challenges with applying vulnerability prediction models. In: Proceedings of the 2015 Symposium and Bootcamp on the Science of Security. HotSoS '15, Association for Computing Machinery, New York, NY, USA.

Ponta, S.E., Plate, H., Sabetta, A., Bezzi, M., Dangremont, C., 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories. MSR, IEEE, pp. 383–387.

Reis, S., Abreu, R., 2021. A ground-truth dataset of real security patches. CoRR, abs/2110.09635.

Robinson, J., Chuang, C.-Y., Sra, S., Jegelka, S., 2020. Contrastive learning with hard negative samples. arXiv preprint arXiv:2010.04592.

Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., McConley, M., 2018. Automated vulnerability detection in source code using deep representation learning. In: 2018 17th IEEE International Conference on Machine Learning and Applications. ICMLA, IEEE, pp. 757–762.

Shin, Y., Williams, L., 2008. An empirical model to predict security vulnerabilities using code complexity metrics. In: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '08, Association for Computing Machinery, New York, NY, USA, pp. 315–317.

Shin, Y., Williams, L., 2013. Can traditional fault prediction models be used for vulnerability prediction? Empir. Softw. Eng. 18 (1), 25–59.

Shu, R., Xia, T., Williams, L., Menzies, T., 2022. Dazzle: using optimized generative adversarial networks to address security data class imbalance issue. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 144–155.

Smith, J., Johnson, B., Murphy-Hill, E., Chu, B., Lipford, H.R., 2015. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 248–259.

Steenhoek, B., Rahman, M.M., Jiles, R., Le, W., 2023. An empirical study of deep learning models for vulnerability detection. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2237–2248.

Sultana, K.Z., Anu, V., Chong, T.-Y., 2021. Using software metrics for predicting vulnerable classes and methods in java projects: A machine learning approach. J. Softw. Evol. Process 33 (3), e2303, e2303 smr.2303.

Talbi, E.-G., 2020. Optimization of deep neural networks: a survey and unified taxonomy. working paper or preprint, URL https://hal.inria.fr/hal-02570804.

Valletta, J.J., Torney, C., Kings, M., Thornton, A., Madden, J., 2017. Applications of machine learning in animal behaviour studies. Anim. Behav. 124, 203–220.

Xuan, H., Stylianou, A., Liu, X., Pless, R., 2020. Hard negative examples are hard, but useful. In: European Conference on Computer Vision. Springer, pp. 126–142.

Yang, X., Wang, S., Li, Y., Wang, S., 2023. Does data sampling improve deep learning-based vulnerability detection? yeas! and nays! In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 2287–2298.

Yao, J., Shepperd, M., 2020. Assessing software defection prediction performance: Why using the matthews correlation coefficient matters. In: Proceedings of the Evaluation and Assessment in Software Engineering. pp. 120–129.

Zheng, M., Huang, Y., Chen, Q., Liu, Y., 2022. Weakly supervised video moment localization with contrastive negative sample mining. In: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 1. p. 3.

Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Adv. Neural Inf. Process. Syst. 32.

Fahad Debeyan: Fahad, currently pursuing a Ph.D. in computer science at Lancaster University, specialises in the field of software engineering. His primary research focus centres on code analysis, with a strong emphasis on identifying and predicting software vulnerabilities. His particular interest lies in leveraging machine learning techniques for the prediction of these vulnerabilities.

Lech Madeyski: Is an Associate Professor and Deputy Head of the Department of Applied Informatics at Wroclaw University of Science and Technology. Dr. Madeyski is interested in industry-relevant research, focused on finding novel solutions to real problems within software engineering and empirical evaluation of the proposed approaches via statistical methods. His research focuses on empirical software engineering, artificial intelligence and machine learning in software engineering (e.g., software defect/vulnerability prediction, code smell detection), reproducible research, and robust statistical methods.

Tracy Hall: Is a Professor in Software Engineering at Lancaster University, UK. Professor Hall's main research interests are in the analysis of code and the detection, prediction and repair of defects and vulnerabilities in code. She is particularly interested in automatic approaches to defect and vulnerability repair. Her interests also include software testing and the human factors in relation to the developers producing code. In particular, the errors developers make that result in faults and vulnerabilities in code.

David Bowes: Previously a Senior Lecturer in the School of Computing and Communications at Lancaster University. Dr Bowes researches Software Engineering techniques for real world industrial application.