Today's Cat Is Tomorrow's Dog: Accounting for Time-Based Changes in the Labels of ML Vulnerability Detection Approaches

Authors:

Ranindya Paramitha, University of Trento (Italy)
Yuan Feng, University of Trento (Italy)
Fabio Massacci, University of Trento (Italy), Vrije Universiteit Amsterdam (The Netherlands)

This work was partly funded by the EU under the H2020 Program AssureMOSS (Grant n. 952647) and the Horizon Europe Program Sec4AI4Sec (Grant n. 101120393), by the Italian Ministry of University and Research (MUR) under the P.N.R.R. – NextGenerationEU grant n. PE00000014 (SERICS subproject COVERT), and by the Dutch Research Council (NWO) under the grant NWA.1215.18.006 (Theseus) and grant KIC1.VE01.20.004 (HEWSTI). This paper reflects only the author's view, and the funders are not responsible for any use that may be made of the information contained therein.

Assurance and certification in secure Multiparty Open Software and Services (AssureMOSS). No single company masters its own national, in-house software. Software is mostly assembled from "the internet" and more than half comes from Open Source Software repositories (some in Europe, most elsewhere). Security & privacy assurance, verification, and certification techniques designed for large, slow, and controlled updates, must now cope with small, continuous changes in weeks, happening in sub-components and decided by third-party developers one did not even know existed. AssureMOSS proposes to switch from process-based to artifact-based security evaluation by supporting all phases of the continuous software lifecycle (Design, Develop, Deploy, Evaluate, and back) and their artifacts (Models, Source code, Container images, Services). The key idea is to support mechanisms for lightweight and scalable screenings applicable automatically to the entire population of software components by Machine intelligent identification of security issues, Sound analysis and verification of changes, and Business insight by risk analysis and security evaluation. This approach supports the fast-paced development of better software with a new notion: continuous (re)certification. The project will generate also benchmark datasets with thousands of vulnerabilities. AssureMOSS: Open Source Software: Designed Everywhere, Secured in Europe. More information at **https://assuremoss.eu**.

Cybersecurity for AI-Augmented Systems (Sec4AI4Sec) . As artificial intelligence (AI) becomes omnipresent, even integrated within secure software development, the safety of digital infrastructures requires new technologies and new methodologies, as highlighted in the EU Strategic Plan 2021-2024. To achieve this goal, the EU-funded Sec4AI4Sec project will develop advanced security-by-design testing and assurance techniques tailored for AI-augmented systems. These systems can democratize security expertise, enabling intelligent, automated secure coding and testing while simultaneously lowering development costs and improving software quality. However, they also introduce unique security challenges, particularly concerning fairness and explainability. Sec4AI4Sec is at the forefront of the move to tackle these challenges with a comprehensive approach, embodying the vision of better security for AI and better AI for security. More information at **https://sec4ai4sec.eu**.

In searCh Of eVidence of stEalth cybeR Threats (COVERT) aims to analyze emerging attack methodologies and develop advanced methods for detecting attacks and identifying guidelines for designing IT systems that ensure reduced vulnerability to new attack categories. The detailed objectives can be divided into four macro categories: (i) Development of advanced tools for analyzing malware and software aimed at identifying vulnerabilities that could be exploited by malware; (ii) Development of tools for analyzing network traffic to identify communications related to ongoing attacks; (iii) Development of machine learning systems that are robust to attacks and through which it is possible to extract knowledge aimed at creating more advanced tools for timely analysis and early identification of attacks; (iv) Analysis of the "human factors" involved in an attack with the development of tools for analyzing and correlating information from OSINT (open sources intelligence) and for the defense and prevention of attacks based on social engineering techniques.

Theseus: Making patching happen project explores ways to improve software patching. The EU cybersecurity framework has evolved into a stricter, harmonized system that imposes requirements on patching. THESEUS has also found that tools like Shodan often misses vulnerabilities. Regulatory deadlines accelerate patching, although coordination issues cause delays. Researchers are developing efficient techniques to detect and mitigate vulnerabilities without performance loss. The reliability of patch-testing tools has proven to be a weak point, requiring improvements to reduce risks. Machine learning is being used to prioritize vulnerabilities more effectively. These insights contribute to better patching by addressing challenges in governance, technology, and operations.

Hybrid Explainable Workflows for Security and Threat Intelligence (HEWSTI). In research into threats to safety and security, people and AI collaborate to obtain actionable intelligence. Their sources and methods often have significant uncertainties and biases. Experts are aware of these limitations, but lack the formal means to handle these uncertainties in their daily work. This project will invent a 'metadata of uncertainty' for threat intelligence (in both machine-readable and also human-interpretable forms) and validate it empirically. Intelligence agencies will then be able to explicitly consider the trade-off between the accuracy, proportionality, privacy, and cost-effectiveness of investigations. This will contribute towards the responsible use of AI to create a safer, more secure society.

Ranindya Paramitha (PhD 2025) is a research fellow at the University of Trento, Italy. She received her PhD from the University of Trento, Italy, in April 2025. Her main research interest is in software security, focusing on empirical analysis of secure software ecosystems, mining software repositories, and how developers can apply security. She is involved in a Horizon Europe Sec4AI4Sec project and has also started to actively serve the research community in several IEEE/ACM International Conferences/Workshops, such as by being a junior PC member (Distinguished Junior PC Award MSR'25) and regular PC member (ICSME'25). Contact her at *ranindya.paramitha@unitn.it.*

Yuan Feng is currently a PhD student at the University of Trento, Italy. Her research interests focus on Machine Learning for Vulnerability Detection (ML4VD) and security in software ecosystems. She is also interested in data quality issues within these domains. Contact her at *yuan.feng@unitn.it.*

Fabio Massacci (PhD 1997) is a professor at the University of Trento, Italy, and Vrije Universiteit Amsterdam, The Netherlands. His research interests include empirical methods for the cybersecurity of sociotechnical systems. For his work on security and trust in sociotechnical systems, he received the Ten-Year Most Influential Paper Award at the 2015 IEEE International Requirements Engineering Conference. He is named co-author of CVSS v4. He leads the Horizon Europe Sec4AI4Sec project and the Dutch National Project HEWSTI. He is the past chair of the Security and Defense Group of the Society for Risk Analysis, and an IEEE CertifAIEd Lead Assessor. Contact him at *fabio.massacci@ieee.org.*

How to cite this paper:

- Paramitha, R., Feng, Y., and Massacci, F. Today's Cat Is Tomorrow's Dog: Accounting for Time-Based Changes in the Labels of ML Vulnerability Detection Approaches. *Proceedings of the ACM on Software Engineering (PACMSE), Issue FSE 2025.* ACM Sigsoft.

Today's Cat Is Tomorrow's Dog: Accounting for Time-Based Changes in the Labels of ML Vulnerability Detection Approaches

RANINDYA PARAMITHA, University of Trento, Italy
YUAN FENG, University of Trento, Italy
FABIO MASSACCI, University of Trento, Italy and Vrije Universiteit Amsterdam, Netherlands

Vulnerability datasets used for ML testing implicitly contain retrospective information. When tested on the field, one can only use the labels available at the time of training and testing (e.g. seen and assumed negatives). As vulnerabilities are discovered across calendar time, labels change and past performance is not necessarily aligned with future performance. Past works only considered the slices of the whole history (e.g. DiverseVUl) or individual differences between releases (e.g. Jimenez et al. ESEC/FSE 2019). Such approaches are either too optimistic in training (e.g. the whole history) or too conservative (e.g. consecutive releases). We propose a method to restructure a dataset into a series of datasets in which both training and testing labels change to account for the knowledge available at the time. If the model is actually learning, it should improve its performance over time as more data becomes available and data becomes more stable, an effect that can be checked with the Mann-Kendall test. We validate our methodology for vulnerability detection with 4 time-based datasets (3 projects from BigVul dataset + Vuldeepecker's NVD) and 5 ML models (Code2Vec, CodeBERT, LineVul, ReGVD, and Vuldeepecker). In contrast to the intuitive expectation (more retrospective information, better performance), the trend results show that performance changes inconsistently across the years, showing that most models are not learning.

CCS Concepts: • **Software and its engineering** → **Software defect analysis**; *Software libraries and repositories*; **Empirical software validation**; • **Security and privacy** → **Software security engineering**; • **Computing methodologies** → *Cross-validation*; **Supervised learning by classification**.

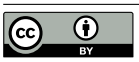
Additional Key Words and Phrases: Dataset tuning, machine learning, perspective, software security

ACM Reference Format:
Ranindya Paramitha, Yuan Feng, and Fabio Massacci. 2025. Today's Cat Is Tomorrow's Dog: Accounting for Time-Based Changes in the Labels of ML Vulnerability Detection Approaches. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE016 (July 2025), 26 pages. https://doi.org/10.1145/3715731

1 Introduction

Using machine learning for (security) bug detection is a recent popular trend in software engineering research [Chakraborty et al. 2021; Marjanov et al. 2022]. Despite the vigorous research, practical deployment is lagging [Arp et al. 2022; Kästner 2022; Lwakatare et al. 2020]. For example, while several open source or commercial tools exist for static security analysis exist (e.g. SonarQuBE, Checkmarx), as well as open ML models for NLP (e.g. [Liu et al. 2019], [Feng et al. 2020], [Hanif and Maffeis 2022]), their transfer to security vulnerability prediction has not been uniformly successful ie. on the reproduction by [Steenhoek et al. 2023], CodeBERT [Feng et al. 2020] and VulbertA [Hanif

Authors' Contact Information: Ranindya Paramitha, University of Trento, Trento, Italy, ranindya.paramitha@unitn.it; Yuan Feng, University of Trento, Trento, Italy, yuan.feng@unitn.it; Fabio Massacci, University of Trento, Trento, Italy and Vrije Universiteit Amsterdam, Amsterdam, Netherlands, fabio.massacci@ieee.org.



ACM 2994-970X/2025/7-ARTFSE016
https://doi.org/10.1145/3715731

and Maffeis 2022] have accuracy lower than 70%. DiverseVul [Chen et al. 2023] reports a dismal F1 of 0.48 for the best LLM Model trained on a large dataset.

One possible explanation is that models are overfitting on different datasets [Chakraborty et al. 2021; Chen et al. 2023] or there are mistakes in the division of training and testing datasets [Arp et al. 2022; Chen et al. 2019]. However, Jimenez and colleagues [Jimenez et al. 2018, 2019] have argued that there is a more fundamental cause that *systematically* creates a gap. Vulnerabilities and bugs have a published date, security-related commits have a commit date, etc. *Before* the date on which the vulnerability was discovered (typically by somebody other than the developer) the 'vulnerable fragment' was *not known to be* vulnerable. The fix, which the ML algorithm will use as an example of not-vulnerable code, *did not even exist.*

Most high-quality datasets can be augmented with the date when the vulnerability was discovered, either by manual labeling (e.g., Devign [Zhou et al. 2019]) or by using other publicly accessible data (e.g., CVEs from NVD [NIST 2024a]) as the ground truth of the labels. For example, BigVul [Fan et al. 2020] uses vulnerability-fixing commits from real-life projects to identify a non-vulnerable fragment (the new version) and a vulnerable fragment (the version before the commit).

Paradoxically, the major problem of all these datasets, whether synthetically created or based on real-life projects, is precisely that they provide at once the complete knowledge of vulnerable and non-vulnerable components at the time of investigation. Therefore, when a dataset is used with ten-fold cross-validation, 80-10-10 split, or other splits, *the ML algorithm benefits from (retrospective) complete information.* The eventually correct labels of some fragments, that in field deployment of the ML model would have only been known in the future (sometimes after years [Hu et al. 2024; Massacci et al. 2011; Nguyen et al. 2016; Ozment and Schechter 2006]), are informing the training.

To address this issue, several studies [Jimenez et al. 2019; Scandariato et al. 2014; Shin et al. 2011] proposed to perform vulnerability prediction based on code metrics, keywords occurrences, function names, and other metrics (but not source code yet) and introduced the idea of restricting the predictor's knowledge to each release: training on the data from release 1 until r and testing on $r + 1$. Yet, they all tested using the labels from the complete information dataset, with the overall information across the years. This does not reflect the deployment scenario on a time t where only a subset of the data and labels are actually known. More recently, studies such as DiverseVul [Chen et al. 2023], proposed to test ML models with an increasing share of the dataset, but they still sample the complete information dataset for training and testing.

We need to extract from the complete information dataset a slice that reflects the evolving knowledge that a model will encounter in time (new, unclassified yet, fragments, changing labels due to late discoveries). Hence, our first question:

RQ1: *How do we extract from a complete-information (retrospective) dataset a timeline of (perspective) datasets corresponding to what would be seen on the field at particular times?*

We propose to use calendar time rather than releases (as in [Jimenez et al. 2019]) or random samples (as in [Chen et al. 2023]) for both correctness[1] and causality[2] considerations to form a timeline in which labels and data points for training and testing are different from the complete information source dataset and capture the partial information available at each time point. We propose to use calendar time instead of releases, also accounting for causality considerations. We train on vulnerability data from time t_0 until observation date t and test on vulnerability data with labels

[1]Figure 1 in our motivating example (§2) shows how one release can have different labels in time, and therefore releases labeled with complete information would use future information. Further, some projects have frequent releases, sometimes even on the same day, and thus no learning could have reasonably taken place (except for fixing releases).

[2]Releases and vulnerability discovery are independent processes collapsing on the times of responsible disclosure. By construction, the vulnerability timestamp in the NVD is *after* (or on the same date) than the time of the new release. The fixing release is not an independent event from the published vulnerability discovery time.

available between the current and the next observation point $t + 1$. *For each time point* in the investigation timeline we generate one training dataset and 2 testing datasets:

(1) *Retrospective Training - Retrospective Testing (R-R)* all available code fragments up to the time of investigation are used for training, testing uses the labels known at the time of investigation.
(2) *Retrospective Training - Perspective Testing (R-P)* test using vulnerable and non-vulnerable code and labels based on the information available in the next time period after the training.

So *R-R* is the performance that researchers would obtain and report on papers by using the dataset available at the time of the investigation. On the other hand, *R-P* is the performance that one would observe by deploying the trained model and testing it on the field after the next time point observation period (e.g., one month, one year).

Given our partitions, a consistent increase in performance (better precision and recall) is what one will expect from a technology to be used in practice as more and more data points are used in the learning process. The existence of such a trend can also be *statistically tested* (with Mann-Kendall) thus showing that a model significantly learns over time.

This finding would have been consistent with DiverseVul experiments [Chen et al. 2023]. Chen et al. Figure 3 shows that increasing the amount of training data from 10% to 90% of the sample improves the F1 score from 0.38 to 0.46. A key observation is that their procedure was a-temporal: the slices are randomly sampled from the *same, complete-information retrospective dataset* and their labels do *not* change from slice to slice (as they may do in reality). Hence, our second research question:

RQ2: *Does ML performance monotonically increase as the available (retrospective) information increases and consolidates?*

We implemented our proposed methodology for the specific case of a vulnerability dataset integrating 4 CVE-based datasets (Linux Kernel, OpenSSL, and Poppler from BigVul [Fan et al. 2020] and NVD dataset by Vuldeepecker [Li et al. 2018]) and a timeline date as input. Given a time-based dataset, we train 5 sota ML-based tools of different types (LineVul [Fu and Tantithamthavorn 2022], CodeBERT [Feng et al. 2020], ReGVD [Nguyen et al. 2022], Code2Vec [Alon et al. 2019], Vuldeepecker [Li et al. 2018]) and tested on vulnerability data using calendar years as period[3].

In sharp contrast to the expectations, our analysis over the years shows that there is no significant trend of ML performance in detecting the vulnerability when tested in the *Retrospective Training - Perspective Testing (R-P)*, which is the one people will experience at a certain point in time. We also show that presenting the ML performance result as (statistically tested) trends provides a more understandable visualization of the ML performance if deployed across the years.

2 Motivating Example

An example of how perspective will be implicitly present in a dataset and how this will change the performance of the ML algorithm is shown in Figure 1. We use five imaginary releases, inspired by the releases used by [Jimenez et al. 2019; Scandariato et al. 2014; Shin et al. 2011]. These releases have some vulnerabilities reported in NVD, portrayed as red arrows in Figure 1. [Jimenez et al. 2019] did 2 experiments, ideal and real. The real experiment uses data from previous (one or three) releases in training, while the ideal experiment uses the complete information data. They claim their labeling (whether a file is vulnerable or not) is based on the reported vulnerabilities *when a version is released*. Therefore, when training on release v2.6.21, V1 would not be present in the training set (File1 and File2, both are known as not vulnerable at that release). Also, in the testing set, File3 will be considered as vulnerable as [Jimenez et al. 2019] always used complete information labeling

[3]Shorter intervals did not have enough changes and therefore the models would have essentially stayed identical.

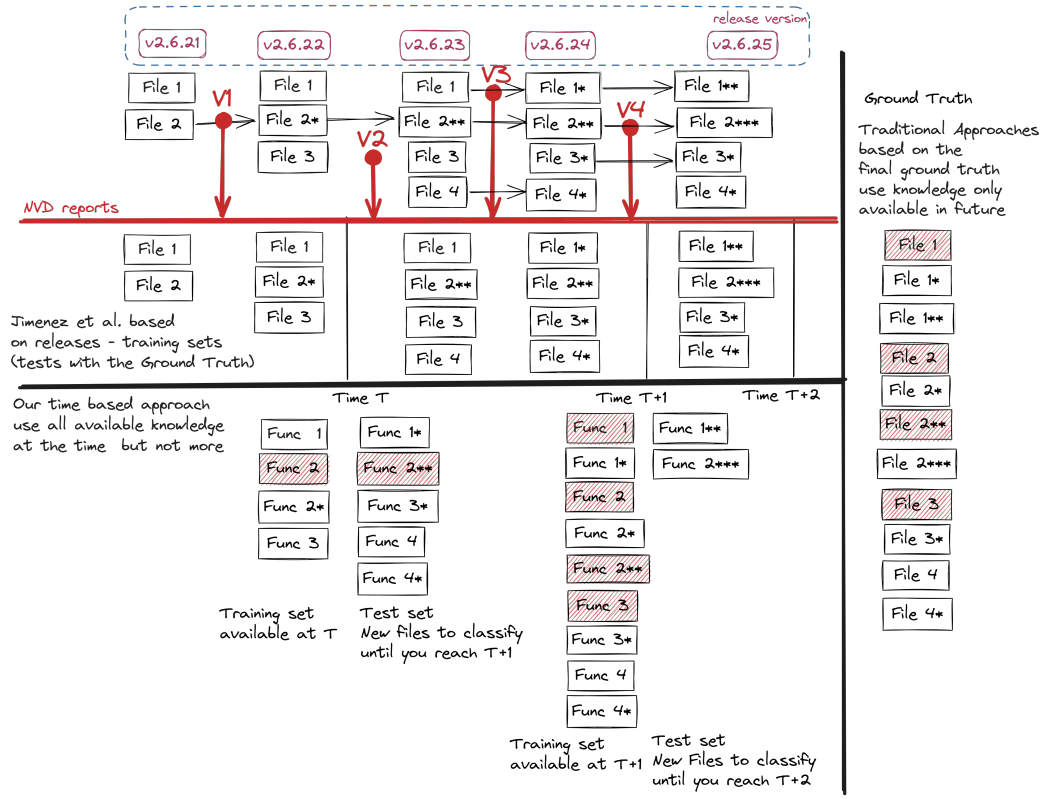


Fig. 1. Motivating example on the benefit of perspective: when you see what you should not see.

to label their testing set. This labeling does not represent the scenario developers will face at that time, because at release v2.6.21 or v2.6.22, File3 was still known as not vulnerable. Moreover, as a patch would most likely happen after a vulnerable version, if we see the training sets, the number of positive data points (vulnerable files) could be very low, for example, File2 and its updates would not be considered as vulnerable in any training. For their ideal experiment, they always train and test using the data with complete information labeling.

Instead of using releases, we propose a methodology that uses *time* as a method to split the dataset. We also use a different set to train and test in our experiments. We train on data and label available until time T and then do 2 tests: retrospective test (using data and label available until time T) and perspective test (test with new data and label from T to $T + 1$).

Our retrospective scenario represents training and testing using only available data until a certain time, while the perspective represents training using available data but testing on *new* data and labels on the next time window. We are *only* using the complete information data to test the last time point (T_N). Different timelines can produce different labels. This reflects what happens at a certain point in time, as vulnerability finding time will affect whether a vulnerability is known.

Additionally, we ran a third experiment with *seen but believed negatives* data/label. We put the discussion on this additional experiment in a separate section (§8).

3 Related Works

3.1 Dataset for Detection ML Training

Early works on bug/defect detection. In their systematic literature review, [Pachouly et al. 2022] mentioned that most of the early research on software defect prediction based on software metrics used NASA, PROMISE [SayyadShirabad and Menzies 2005], and AEEM software defect datasets in their evaluation. The quality of such data was also disputed [Shepperd et al. 2013]. Further, they do not use any issue/bug tracking system as their ground truth. A few datasets, such as Defects4J [Just et al. 2014] and iBugs [Dallmeier and Zimmermann 2007] use bug tracking systems in the version control system to find their label ground truth. According to [Pachouly et al. 2022], also [Yatish et al. 2019] uses JIRA bug tracking systems as their ground truth. These datasets also provide information on when the bug is reported. More recent works [Jimenez et al. 2019; Scandariato et al. 2014; Shin et al. 2011] have broadened the type and scope of datasets used.

Vulnerability detection. [Chen et al. 2023] collected 10 vulnerability detection datasets. Among the 10 datasets, SATE IV Juliet [Okun et al. 2013] and SARD [NIST 2024c] are synthetic and semi-synthetic datasets that cannot fully represent real-world vulnerabilities, and they are used to evaluate several ML models [Li et al. 2018; Mirsky et al. 2023; Russell et al. 2018]. On the other hand, Devign [Zhou et al. 2019] is a manually labeled dataset consisting of 2 C projects (that are publicly available): FFMPeg and Qemu. Though a lack of a clear explanation of why a fragment is considered vulnerable, Devign has also been used by many ML models [Ahmad et al. 2021; Feng et al. 2020; Hanif and Maffeis 2022; Nguyen et al. 2022; Zhou et al. 2019].

To limit manual work, Draper [Russell et al. 2018] and D2A [Zheng et al. 2021] utilize static analyzers to decide whether a code is vulnerable. However, the quality of these labels tends to be low (D2A's authors reported 53% label accuracy. Other works utilizes security issues repository to generate a high-quality-labeled dataset such as ReVeal [Chakraborty et al. 2021], BigVul [Fan et al. 2020], CrossVul [Nikitopoulos et al. 2021], CVEfixes [Bhandari et al. 2021], ProjectKB [Ponta et al. 2019], DiverseVul [Chen et al. 2023], and apart of Vuldeepecker's dataset [Li et al. 2018]. These datasets have higher accuracy labels, but model's evaluations based on them do not take into account the fact that they contain time-based information: some vulnerabilities are known (published) at a certain point in time. A summary of datasets and ML models is given in Table 1.

Commit classification. There are several manually labeled datasets on commit classification: [Ghadhab et al. 2021] has created a manually labeled dataset of 1793 commits; [dos Santos and Figueiredo 2020] includes 3 manually classified datasets [Levin and Yehudai 2017], [Mauczka et al. 2015], [Safdari 2018]. Other works tried to identify vulnerability-contributing commits (VCC) or security-relevant commits. [Perl et al. 2015] by taking available CVEs with links to commits and tracing back the introducing commit (640 VCCs from 718 CVEs). A similar work is from [Le et al. 2021], which used NVD to get 1,229 unique VCCs and manually curated some fixing commits (VulasDB [Ponta et al. 2019]). Moreover, TreeVUL [Pan et al. 2023] categorized vulnerable types referring to the CWE tree and provided a fine-grained vulnerable type dataset. These datasets, which are based on an issue tracker, actually have a time component: the fix commits are only known when the vulnerability (with the fix) is published. Evaluating ML models using this kind of dataset makes it possible to exploit the time variable to avoid the benefit of hindsight.

3.2 Models for Vulnerability Detection

Learn Features. [Li et al. 2018] proposed Vuldeepecker which could find vulnerabilities that hadn't been reported in NVD. They built code gadgets and transformed them into vectors. The problem for Vuldeepecker is that it does not perform well in real-world situations [Chakraborty

Table 1. Used datasets in the SOTA of ML-based vulnerability detection.

Different ML models on vulnerability detection have been evaluated in the state of the art with different datasets. Some of them are time-based (labeling based on CVEs or issue trackers), and some are not. In our validation, we use the time-based dataset with CVE, and for our preliminary validation, we choose NVD, the smaller dataset. In bold are the dataset and models we use in our evaluation.

Dataset	Description	Labeling	Has CVE?	Used by ML	Architecture
NVD [NIST 2024a]	Fragments extracted from vulnerable software lined to the NVD Dataset managed by NIST.	CVE-based	✓	**Vuldeepecker [Li et al. 2018]**	**RNN, BLSTM**
SARD [NIST 2024c]	Artificial dataset of test programs with documented weaknesses.	Semi-synthetic	✗	SySeVr [Li et al. 2022a]	RNN
BigVul [Fan et al. 2020]	A large C/C++ vulnerability dataset from FOSS Github projects.	CVE-based	✓	**LineVul [Fu and Tantithamthavorn 2022]**	**Transformers**
Devign [Zhou et al. 2019]	Manually labeled dataset of vulnerable and non-vulnerable codes from 2 large-scale open-source C projects: FFMPeg and Qemu.	Manual	✗	**CodeBERT [Feng et al. 2020]**	**Transformers**
				VulBERTa [Hanif and Maffeis 2022]	Transformers
				PLBART [Ahmad et al. 2021]	Transformers
				Code2Vec [Alon et al. 2019]	**MLP, AST**
				ReGVD [Nguyen et al. 2022]	**GNN, token**
				Devign [Zhou et al. 2019]	GNN, property graph
Chromium+Debian [Chakraborty et al. 2021]	A dataset of Chromium and Debian codes labeled by issue tracker.	Issue-based	✗	ReVeal [Chakraborty et al. 2021]	GNN, property graph
DiverseVul [Chen et al. 2023]	A C/C++ dataset from security issues websites, vulnerability-fixing commits, and source codes.	Issue-based	Some	Some LLM4Security such as LLMVulExp [Mao et al. 2024]	LLM

et al. 2021]. And [Li et al. 2022a] further proposed SySeVR to extract more related syntax and semantic vulnerability candidates automatically while reported in [Chakraborty et al. 2021], the model's accuracy dropped by 73 percent when using real-world datasets.[Chakraborty et al. 2021] made a high-quality dataset with less duplication, more real data, and fewer irrelevant features in their model named ReVeal.

Other works are also engaged in learning more features. Devign [Zhou et al. 2019] used the Conv module to help extract features for the GNN model in the model's classification process. Code2vec [Alon et al. 2019] used a code vector to predict semantic attributes. SvulD[Ni et al. 2023] embedded subtle semantic information into the model which distinguishes and representative semantics are used. VulBG [Yuan et al. 2023] considers functions' features and behaviors to use in other functions' vulnerable detection. [Li et al. 2024] considered context information.

Text Classification. After RoBERTa [Liu et al. 2019] and related pre-trained models significantly improved on natural language processing problems, CodeBERT [Feng et al. 2020], VulBertA [Hanif and Maffeis 2022] and PLBert [Ahmad et al. 2021] were proposed to work in source code's vulnerability detection. CodeBERT was the first to program language (PL) with natural language (NL) knowledge, which trained with a hybrid objective function. [Nguyen et al. 2022] proposed ReGVD which used raw source code as tokens to build graphs. Moreover, LineVul [Fu and Tantithamthavorn 2022] adopts Transformer techniques that can work on line-level predictions, other than on a function or file level. The results with DiverseVul [Chen et al. 2023] show that LLMs can only be competitive with GNNs for very large datasets. Interestingly, LLMs require a large amount of training data to surpass ReVeal. When trained solely on CVEFixes data, a much smaller training set, there is no clear advantage of LLMs over GNN- based ReVeal model and ReVeal is even better than 6 LLMs (out of 10).

3.3 Model's Evaluation

For the model's evaluation, previous works focused on two factors: metrics and operations [Chakraborty et al. 2021]. Some other dimensions are considered based on the design of models. [Li et al. 2019] proposed to do quantitative evaluations between different factors with two datasets.

Table 2. Comparison Between Previous Works

Study	Systems	Evaluation	Results	Labels
[Shin et al. 2011]	Mozilla Firefox, Red Hat Enterprise, Linux kernel	Code metrics, Sample until each release	Precision .03 - .05, Recall .87 - .90 & .79 - .85	Until Release, Complete Info
[Scandariato et al. 2014]	20 Android apps	Keywords, Sample until each release	Precision .90 & .86, Recall .77	Until Release, Complete Info
[Jimenez et al. 2016]	Linux Kernel	Code metrics and function names/imports, Sample until each release	Precision .65 & .76, Recall .22-.64 & .16-.48	Until Release, Complete Info
[Jimenez et al. 2019]	Linux Kernel, OpenSSL, Wireshark	Code metrics and function names/imports, Sample until each release	Precision .45-.83, Recall .36-.77	Until Release, Complete Info
[Chen et al. 2023]	DiverseVul, Devign, ReVeal, BigVul, CrossVul, CVEFixes	Source code functions, Random sample of complete info	F1 .09-.47 Precision .12-.52, Recall .05-.44	Complete Info
This paper	Linux Kernel, OpenSSL, Poppler, NVD Vuldeep.	Source code functions, Sample until each time	Recall 0.0-1.0, FPR 0.1-.91	Retrospective (Until Time t) & Perspective (Until Time $t+1$)

Besides, CodeBERT [Feng et al. 2020] was first evaluated by NL-PL tasks based on the model's specific structure. It was evaluated separately on both the NL side and the PL side.

Moreover, Devign [Zhou et al. 2019] proposed baseline methods for the model's performance evaluation, including Metrics + Xgboost, 3-layer BiLSTM, 3-layer BiLSTM + Att, and CNN. This evaluation procedure was also adopted by ReGVD [Nguyen et al. 2022]. However, none of the previous evaluations actually take into account the time variable, which can exist in a vulnerability detection dataset as vulnerability is known at a certain point in time.

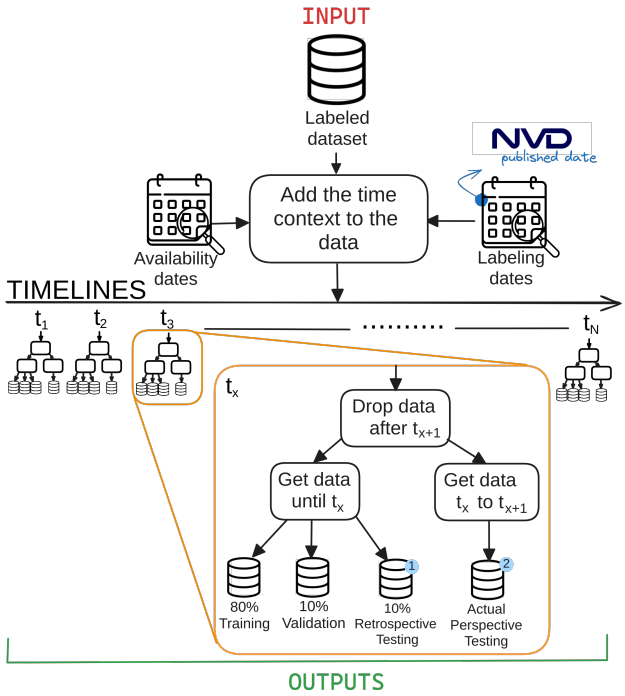
In Table 2, we list some previous studies that worked on the evaluation method: release-based validation and cross-validation [Jimenez et al. 2016; Scandariato et al. 2014; Shin et al. 2011]. They used metrics and keywords extracted from files as the prediction. The most complete work is by Jimenez and colleagues 2019 who considered all different code metrics used by previous work. evaluated the model with consideration of different versions of packages that previous works did not count into.

The closest papers to our work are the studies by Jimenez et al [Jimenez et al. 2019] and Chen et al.[Chen et al. 2023], we list different parts of methodology in Table 3. Firstly, we work on a time-based dataset considering the CVE published date while DiverseVUl sample the overall dataset so it may contain vulnerability information that will be available in the future, while Jimenez et al. work on the released date of the repository (which does not account for the close, induced release of the CVE entry). The major difference is that [Chen et al. 2023; Jimenez et al. 2019] *always* test on the complete information data, when training both with the ideal or (so-called) real datasets. These data and labeling do not represent what developers will face in the real-life scenario, as they are not available yet. In contrast, we use data and labels available *at each timeline* to compare between our retrospective (testing using data and label available at time t) and perspective (testing on data and label available between t and $t+1$). After experimenting with our datasets, we can see clearly how model performance would be influenced by the perspective data.

Main Gap: Several datasets for ML evaluation have a time variable, but the current evaluation of ML models in SOTA often fails to take this variable into account.

Table 3. Comparison between Jimenez et al. 2019, Chen et al 2023 and this Paper

	[Chen et al. 2023]	[Jimenez et al. 2019]	This Paper
Methodology			
Training Set	Random SubSample	Sample until Release	Sample until Time
Training Labels	Complete Info	Known at Release	Known at Time
Testing Set	Sample of Complete Info	Sample until Next Release	Sample until Next Time
Testing Label	Complete Info	Complete Info	Known at Next Time
Results	Result as a trend (line chart)	Global results (boxplots)	Result as a trend, (line charts)
Evaluation			
Systems(Datasets)	DiverseVUl, Devign, ReVeal, BigVul, CrossVul, CVEFixes	Linux Kernel, OpenSSL, Wireshark	Linux, OpenSSL, Poppler (from BigVul [Fan et al. 2020]), NVD Vuldeepecker
Models	1 GNN, 10 BERTs	Random Forests	2 BERTs, 1 GNN, 1 BLSTM, 1 MLP



Given a time-based dataset, our proposed methodology adds time context to the data from a given source and then produces a timeline of datasets with perspective views. Time context can be instantiated with vulnerability publishing date, such as NVD, bug issue report date, etc. For each time point in the timeline, it produces one training set (possibly also one validation set) and 2 testing sets. It is important to note that we do not only split the data points by time, but also use only available labels at every point in time.

Fig. 2. Our proposed methodology to eliminate retrospectives.

4 Methodology

To answer RQ1, we propose a methodology to produce a timeline of datasets from a time-based labeled dataset, given a specific timeline of dates. The conceptual flow of the approach is depicted in Figure 2, and the inputs/outputs are summarized in Table 4.

We elaborate on the steps of the methodology as follows (pseudocode is shown in Algorithm 1).

(1) *Add time context to data.* In this step, the data is mapped with the inputted labeling and availability dates. If the inputted labeling and availability dates come from another dataset, then

Table 4. Input and Output of Our Methodology

Object	Description	Instantiation
Input		
Labeled dataset	A dataset to train an ML-based vulnerability detection tool/ model with CVE information. These CVEs will then be used to determine whether the vulnerability is known or in perspective at the time of observation.	BigVul [Fan et al. 2020], Vuldeepecker's [Li et al. 2018]
Availability dates	A set of dates that are pre-selected to simulate the information availability at a certain point in time for the training of the ML model	Published dates of the codes or commits.
Labeling dates	A set of dates that are pre-selected to simulate the label availability at a certain point in time for the training of the ML model	Published dates of CVEs in the NVD.
Timeline dates	A set of dates to simulate when we are observing the ecosystem.	2010, 2011, 2012, ...
Testing delta	(Optional) a delta in months that we use to simulate the information availability at a testing time (which intuitively should be later than training time).	6 months/ 12 months
Output (N time points)		
For each time point in a timeline,		
Training set	X% of the relabeled dataset to train the ML model	80% + 10% validation set
2 testing sets $\times N$	*R-R* test set, *R-P* test set to test the ML model	2 test sets $\times N$
2 testing results $\times N$	The result of testing the ML model on the 2 test sets $\times N$	Precision, Recall (TPR), FPR, ... $\times$ 2 test sets $\times N$

the dataset is fetched before the mapping (e.g., a record labeled 1 now becomes labeled 0 at time t).

(2) *Produce a timeline of datasets to test ML's performance at each time point in a timeline.* We take a timeline date and the testing delta and use them as our reference. **For each time point in the timeline,** we produce a training dataset and a set of testing datasets from the input dataset based on different possible assumptions.
   Retrospective Training - Retrospective Testing (R-R): If the data point's labeling date is later than the chosen timeline, we drop positive and negative data points from the database. This rule assumes that before the timeline, the point does not exist.
   Retrospective Training - Perspective Testing (R-P): We keep records with a labeling date later than the chosen timeline but still inside the testing delta.

(3) ***For each time point in the inputted timeline****, train the ML on the training set.* We train the ML on a subset of the relabeled dataset containing X% (predefined or provided as input) of data points from the complete info dataset. This subset can be divided into training and validation datasets.

(4) ***For each time point in the inputted timeline****, test the ML model on 2 different scenarios* mentioned in Step 2: *R-R* and *R-P*.

We can then plot the metrics from the tested model, both for *R-R* case and *R-P* case, to see if there is a certain trend or difference. For this comparison, any metrics would do because basic metrics (TP, FP, TN, and FN) are not independent: at least their sum must be equal to the total number of samples. In the general case, there are more complex dependencies due to the chosen ML algorithm. Therefore, if one aggregates these basic metrics to form other metrics (such as precision, recall, and F1) and one metric fluctuates, the other metrics would also fluctuate in balance.

Algorithm 1: Produce a timeline of datasets.

```
input  : Labeled dataset (original_dataset)
input  : Labeling dates (label_dates)
input  : Availability dates (availability_dates)
input  : Timeline dates (timeline_dates)
input  : (Optional) Testing delta (delta in months, default=12)
input  : (Optional) Percentage (percentage)
output : A Timeline of datasets (datasets)

// Add time context to data.
dataset ← mapDataWithTime(original_dataset, label_dates, availability_dates)

// The function to get 3 testing datasets.
Function relabel(dataset, timeline_date):
    Perspective_test ← [ ]
    for rec|rec ∈ dataset do
        if rec.label_date > timeline_date + delta then
            dataset.dropData(rec)
        else if rec.label_date > timeline_date then
            // assumption == R-P
            Perspective_test.append(rec)
            // For Retrospective: Both label and record unknown: we drop the record
            dataset.dropData(rec)
    return dataset, Perspective_test

// Produce a timeline of datasets.
datasets ← {}
for tdate|tdate ∈ timeline_dates do
    dataset, Perspective_test ← relabel(dataset, tdate)
    datasets[tdate]["perspective"] ← Perspective_test
    // Split into training and testing.
    training, validation, testing_dataset ← splitTrainTest(dataset, percentage)
    datasets[tdate]["training"] ← training
    datasets[tdate]["validation"] ← validation
    datasets[tdate]["retrospective"] ← retrospective_test
return datasets
```

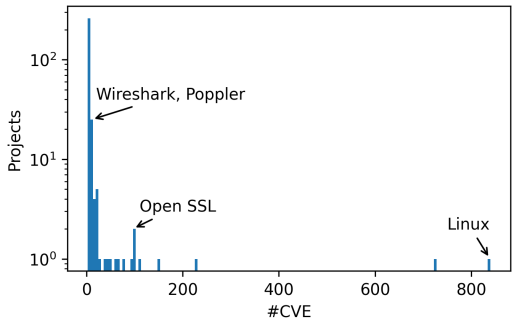
5 Dataset and Model Selection

5.1 Dataset

We wanted to replicate the study by [Jimenez et al. 2019] (the study with a similar idea to ours) using our methodology. However, their dataset does not include the publication date and/or CVE ID of the vulnerability, which makes our methodology inapplicable. We also tried to use VulDat7 [Jimenez et al. 2018], which is the framework used by [Jimenez et al. 2019] to generate their dataset, but the framework is outdated and cannot be used. Therefore, we decided to instead use the BigVul dataset [Fan et al. 2020], which covers real vulnerabilities until 2019. Figure 3 shows the distribution of the number of CVEs (#CVE) by project. It also shows the position of the 3 projects used by [Jimenez et al. 2019]: `linux`, `wireshark`, and `openSSL` in the distribution.

In BigVul, `wireshark` has 10 CVEs, but all of them are published in the same year. Therefore, `wireshark` is not a good example to show how our methodology shows the performance difference between the realistic and the ideal world, as we do not have the data to test in the realistic setting (assuming we train on y and test on $y + 1$). We then decided to choose another project from the same position in the distribution (representing projects with low CVE count): `poppler`, whose number of CVEs is in the same group as `wireshark` but its CVEs are published in different years spanning from 2009 until 2018.

Additionally, we reviewed 10 different papers from the SOTA on ML-based vulnerability detection evaluated by [Steenhoek et al. 2023]. We then clustered them based on the datasets they used in



The 3 projects chosen by [Jimenez et al. 2019] cover the ecosystem well, as `linux` can represent the projects with high number of CVE, `openSSL` for projects with medium number, and `wireshark` for the low number. As `wireshark`'s CVEs are published in the same year, we take another project from the same position `poppler` to represent the projects with a low number of CVEs

Fig. 3. Distribution of CVE count (#CVE) in BigVul Dataset

Table 1. This table has covered different ML architectures, ie. RNN, MLP, GNN, and transformers. We then decide to also evaluate our methodology on the NVD dataset by Vuldeepecker [Li et al. 2018] which can be found in Github [Li et al. 2022b] as it is lightweight and suitable for a preliminary evaluation. We chose this dataset as it uses real-world vulnerabilities (CVE-based) like BigVul dataset [Fan et al. 2020], and is not manually labeled like Devign's dataset [Zhou et al. 2019]. From the NVD dataset, we filtered to get only the records with CVEs. Vuldeepecker provided 2 datasets: CWE-119 and CWE-399, which we combined to get more records. The merged dataset contains 628 records with 68 CVEs in total.

5.2 Models

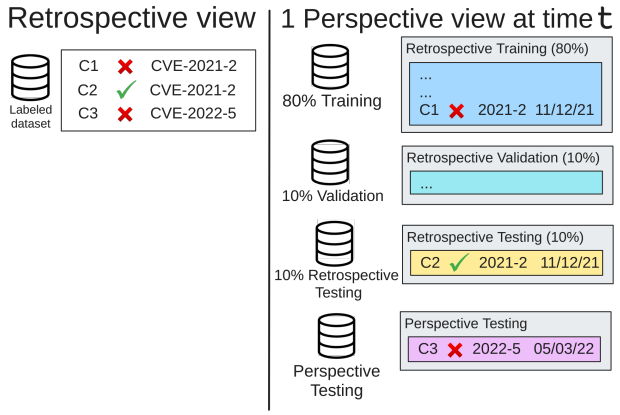
In Table 1, we reported a selection of historical works to illustrate the evolution of the field. Each line differs from the previous ones in either using a different representation or dataset. In recent years, we have seen a consolidation of some preferred methods (BERT and LLMs in general) and a stronger attention to the development of datasets [Risse and Böhme 2024], which have evolved over the years. Despite the different representations adopted, the input code is often flattened into a vector when serving as input for a neural network. Models also evolved: after the first BERT-based detection model [Hanif and Maffeis 2022] or variants tested different languages [De Sousa and Hasselbring 2021] used previous models as a pre-trained model [Mamede et al. 2023].

We then picked five ML models (Vuldeepecker, ReGVD, CodeBERT, Code2Vec and LineVul) with different ML architectures out of other models in Table 1 for our evaluation. We picked them to see how different ML architectures (BLSTM, GNN, MLP, and Transformers) react to the elimination of retrospectives that we did. We ran the replication package by [Steenhoek et al. 2023] for all models except Vuldeepecker, for which we ran an implementation in Python available on Github [Johnb110 2022].

6 Implementation

To see the methodology in action, we implement a Python script for a specific use case: *vulnerability detection dataset*. Our Python implementation needs several inputs that are mapped to the inputs in our methodology:

- **Labeled dataset**: refer to Subsection 5.1 of dataset selection.
- **Labeling dates**: for the code labeling dates (vulnerable/ not vulnerable), we use NVD [NIST 2024a]'s published date. For any vulnerability in the input dataset with an unknown published date, we fetch it directly through NVD's APIs [NIST 2024b].



One record on a dataset is a code fragment with a CVE label. The code availability dates are not available, so we assume that the code is available before the code is labeled by a CVE (vulnerable or not vulnerable). The date on the record is the date when a CVE is published for a certain code. The negative records also have a reference to a CVE because they are extracted from the same commit as the vulnerable code mentioned in the CVE. From one input dataset, we produce a timeline of datasets, of which for each time point, we produce 1 training (and 1 validation) set and 2 testing sets. (retrospectiveand Perspective).

Fig. 4. Example instantiation of the proposed methodology.

- **Availability dates**: as the original dataset we use does not have information about the code availability dates, we assume that the code availability is less than or equal to the labeling dates (NVD published date). This assumption holds as the code has to be available before a CVE can be found. For this implementation, we use labeling dates as availability dates. We acknowledge this as a possible threat to validity in Section 9.
- **Testing delta:** we use 12 months as we assume that the machine learning trained in y_0 will be used (tested) to identify vulnerability in the next year (y_1).[4]
- **Split train-test percentage**: this input adds flexibility for the user of the script to define how to split the dataset into training and testing, and also possibly validation datasets. The default is 80% training, 10% validation, and 10% testing.

The script is implemented with the following steps:

(1) *If the input dataset is not completely CVE-based*: Take only the CVE-based part of the dataset to be processed.
(2) Remove unavailable information from the data and relabel: **for each time point in the timeline**, we use the assumption rules mentioned in Section 4 to assess 3 kinds of performances. So for each record in the dataset
   P1 *Retrospective Training - Retrospective Testing (R-R).* We check on the NVD when the CVE is published. If the published date of the CVE is later than the timeline date, we drop the records. This simulates the training at a certain point in time by using only records with known labels at that time.
   P2 *Retrospective Training - Perspective Testing (R-P).* We keep only records in the testing delta after the chosen timeline. This dataset simulates the real-life testing as a model that we trained on period t would most likely be used in production to detect vulnerabilities during period $t + 1$.

   The example of the input retrospective dataset and one of the produced perspective dataset (for time t) is shown in Figure 4.

[4]A shorter timeline, e.g., every quarter or even every month, is an interesting option, but its prediction will oscillate too strongly as not many code fragments are committed in our dataset. A larger dataset with daily or monthly updates would be needed, and we plan to design it for future work.

Table 5. Produced datasets for training and testing

The datasets grow more and more because the code availability and the CVEs found are also getting more and more throughout the years. However, the proportion of positive and negative data points remains stable.

Dataset	Metric	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	2018	2019
Linux	Training	-	-	-	9	7668	13274	17234	20871	27548	32159	35315	38014
	Validation	-	-	-	2	853	1476	1916	2319	3062	3574	3925	4225
	Retrospective Testing	-	-	-	2	948	1640	2128	2578	3402	3971	4361	4694
	Perspective Testing	-	-	-	9456	6921	4888	4490	8244	5692	3897	3332	-
OpenSSL	Training	-	-	-	-	-	28	247	780	1163	1308	1340	1505
	Validation	-	-	-	-	-	4	28	88	131	147	149	168
	Retrospective Testing	-	-	-	-	-	4	31	98	145	163	166	187
	Perspective Testing	-	-	-	-	-	270	660	473	179	37	205	-
Poppler	Training	-	475	578	578	578	814	844	844	844	872	873	-
	Validating	-	53	66	66	66	91	95	95	95	98	98	-
	Retrospective Testing	-	60	73	73	73	102	105	105	105	109	109	-
	Perspective Testing	-	129	-	-	290	37	-	-	35	1	-	-
NVD Vuldeep.	Training	146	249	300	416	663	941	1175	1691	2163	2192	-	-
	Validating	18	31	37	52	82	118	146	211	271	274	-	-
	Retrospective Testing	19	32	39	53	84	118	148	212	271	276	-	-
	Perspective Testing	129	64	145	308	348	292	645	591	37	-	-	-

Table 6. Dataset generation across the years

The mean derivative shows the increase of data points every year, which reaches 48.6%. The *Never seen* columns show how the test in the Perspective dataset contains way more data points compared to the retrospective dataset (*Seen and known* columns).

Dataset	Metric	Fraction with Vulnerabilities: Seen and known Positive Training t	Seen and known Positive Validation t	Seen and known Positive Present Test t	Seen but Believed Negative at that time	Never seen Positive, tested $t+1$	Never seen Negative, tested $t+1$
Linux	Mean Drv.	28.9%	28.7%	28.4%	75.9%	75.9%	49.2%
	Relative %						
	Mean	3.2%	0.4%	0.4%	1.1%	4.2%	95.8%
	St.Dev	0.2%	0.0%	0.0%	0.9%	0.9%	0.9%
OpenSSL	Mean Drv	50.3%	48.1%	54.5%	31.8%	31.8%	64.9%
	Relative %						
	Mean	8.2%	1.1%	1.2%	33.5%	10.6%	89.4%
	St.Dev	2.6%	0.7%	0.7%	33.4%	6.1%	6.1%
Poppler	Mean Drv	7.3%	2.8%	6.5%	27.8%	27.8%	31.7%
	Relative %						
	Mean	2.8%	0.4%	0.4%	0.3%	0.3%	52.8%
	St.Dev	0.2%	0.0%	0.0%	0.5%	0.5%	50.2%
NVD Vuldeep.	Mean Drv	38.5%	39.5%	37.6%	92.6%	92.6%	81.5%
	Relative %						
	Mean	27.3%	3.3%	3.5%	14.2%	14.2%	60.3%
	St.Dev	4.1%	0.6%	0.4%	8.2%	8.2%	10.2%

(3) Split train-test: based on the input, we split the datasets into training and testing (also validation if necessary). For the second assumption, we merge the testing dataset from the splitting with the added negatives.
(4) Return the outputs: 1 training set (can be divided into training and validation), and 2 different testing sets: *retrospective* and *Perspective*.
(5) Train and test (2 times) the ML models.

7 Evaluation

This section presents our methodology evaluation with the security vulnerability detection use case and how it answers our research questions.

7.1 Dataset Generation

We first generated 2 datasets for each timeline and for each project using our Python script that implemented our methodology. The generated datasets are shown in Table 5. From the generated datasets across the years, we can have several observations as portrayed in Table 6.

Even with the data point increase, the relative mean and standard deviation show that the percentage of positive (vulnerable) data points is more or less the same yearly. The *Never seen Positive/Negative, tested next year* columns have the data from the next year to simulate testing a previously trained ML model in the next year's data. On average, every year we test 6.2%-14.9% of the total population of negatives that have never been seen. The mean derivative of the yearly amount of data points shows that it increases 31.7%-81.2% every year. With complete information, we test with only 0.2%-4.2% of negatives every year, while if we consider the information available at any given time, we test with 52.8%-95.8% of the negatives that have never been seen.

Finding #1: If you do not account for perspective the ML model will be trained on data points that were either not correctly classified (up to 1/3 of the vulnerabilities seen in the previous period) or not even available (up to 81.5% wrt those seen the previous period).

7.2 Models Evaluation

After getting the modified datasets, we then applied 5 different ML-based vulnerability-finding tools on both the original dataset and modified datasets and observed the change in their performance. These 5 tools are chosen among different ML tools with different architectures from Table 1: Vuldeepecker (RNN/BLSTM), Code2Vec (MLP/AST), ReGVD (GNN, token), CodeBERT and LineVul (Transformers). In Table 7, we list all models' differences between precision (left) and recall (right) results with the Perspective and retrospective datasets. For precision, some models work better in the retrospective, and some other work better with Perspective. While for recall, the models' performances with retrospective datasets are on average higher than with Perspective datasets except for CodeBERT in `linux` dataset, Vuldeepecker in `NVD Vuldeepecker`, and LineVul/Code2Vec (slightly) in `openssl dataset`.

We tested whether such a difference between the MLs' performance tested in *R-P* and *R-R* is significant with the Wilcoxon signed rank test, with effect size estimator for correlated samples by Vargha and Delaney (A) [Ruscio and Gera 2013]. We believe the A value has here a natural Software Engineering interpretation: the probability that by taking at random a sample with the "benefit of hindsight", its ML performance will be higher than the sample performance in reality. Since we are testing 5 different tools, according to Bonferroni correction, the p_{value} of a test needs to be $< 0.05/5$ *or* 0.01 to be considered as significant. The test on recall results in a statistically significant difference for CodeBERT ($T = 42.0, p_{value} = 0.002, A = 0.75$), LineVul ($T = 68.0, p_{value} = 0.006, A = 0.71$), and ReGVD ($T = 15.0, p_{value} = 0.004, A = 0.7$). While for precision, all the tests return insignificant ($0.015 < p_{value} < 0.55$) after the Bonferroni correction.

To see if we can reach better performance by giving more perspective to the model, we portray the precision (left) and recall (right) of the models (tested with Perspective dataset) by time in Figure 5. We show line charts to show the trends and boxplots as has been done in [Jimenez et al. 2019]. In `linux` dataset, most models, except Vuldeepecker, have on average high precision and recall. However, there are no increasing trends as one could hypothesize with more and more retrospective data. Vuldeepecker's recall even goes up and down, showing no trend. Interestingly, in the `OpenSSL` dataset, the figure changes. LineVul, ReGVD, and CodeBERT still have high precision, but their recall drops with more and more retrospectives, while Code2Vec and Vuldeepecker show no trend by going up and down.

Table 7. Models evaluation: difference in the precision and recall statistic.

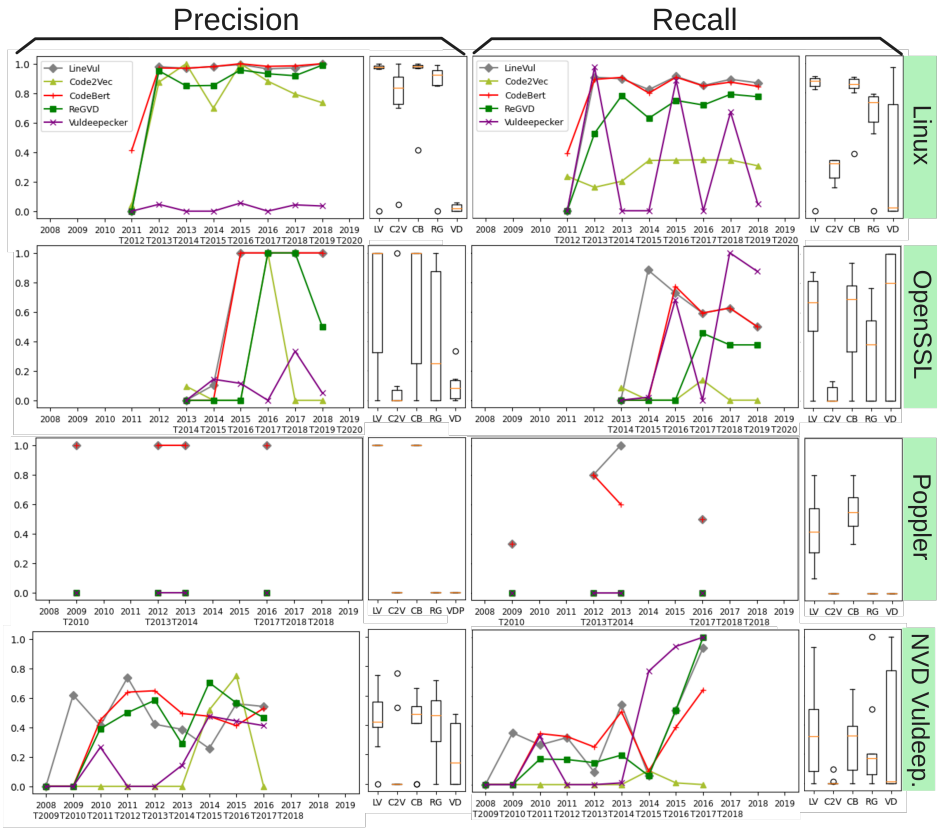
RF: Random Forest from [Jimenez et al. 2019], LineVul, C2V: Code2Vec, CB: CodeBERT, RG: ReGVD, VD: Vuldeepecker. The more positive the gap, the better the performance is. The recall on average is better when evaluated using the retrospective dataset, except for CodeBERT in `linux` dataset, LineVul and Code2Vec (slightly) in `openssl` dataset, and Vuldeepecker in `NVD Vuldeepecker` dataset. This shows that the ML models' Perspective performance (*R-P*) can be not as good as we got from testing using the same year data with the training (*R-R*).

Precision Gap between Perspective and Retrospective Testing

Metric	RF	LV	C2V	CB	RG	VD
`Linux`						
Mean	-43.5%	+1.3%	-0.9%	+7.3%	+1.0%	-0.8%
St.Dev.	+31.2%	+2.5%	+4.4%	+14.0%	+3.3%	+2.9%
`OpenSSL`						
Mean	-8.72%	-10.5%	-1.2%	0.0%	+7.2%	+1.2%
St.Dev.	-25.4%	+39.6%	+58.4%	0.0%	+24.4%	+13.4%
`Poppler`						
Mean	N/A	0.0%	0.0%	0.0%	0.0%	-2.3%
St.Dev.	N/A	0.0%	0.0%	0.0%	0.0%	+4.6%
`NVD Vuldeepecker`						
Mean	N/A	-35.9%	-19.8%	-10.5%	-12.7%	+0.7%
St.Dev.	N/A	+16.5%	+36.2%	18.1%	+16.4%	+9.5%

Recall Gap between Perspective and Retrospective Testing

Metric	RF	LV	C2V	CB	RG	VD
`Linux`						
Mean	-73.5%	-1.4%	-3.6%	+5.2%	-5.0%	-14.9%
St.Dev.	+18.1%	+6.8%	+13.3%	+14.7%	+5.4%	+52.2%
`OpenSSL`						
Mean	-37.0%	+0.6%	+0.5%	-10.1%	-4.5%	-47.1%
St.Dev.	+9.1%	+30.9%	+9.8%	+19.1%	+11.3%	+49.3%
`Poppler`						
Mean	N/A	-27.9%	0.0%	-29.6%	0.0%	-12.5%
St.Dev.	N/A	+40.2%	0.0%	+15.7%	0.0%	+25.0%
`NVD Vuldeepecker`						
Mean	N/A	-47.7%	-3.5%	-28.5%	-33.6%	+33.2%
St.Dev.	N/A	+29.3%	+5.4%	-21.0%	+34.8%	+43.0%



In `linux` dataset, the performance of the ML models (except Vuldeepecker) seems to have a trend: it gets better with more retrospective data. However, this trend does not happen in the other 3 datasets.

Fig. 5. Evolution of Precision and Recall of 5 ML models with more and more retrospectives.

While LineVul seems to work well on `linux` dataset, it performs worse in the `NVD Vuldeepecker` dataset (lower right) with lower precision and recall (average 34.01% st.dev. 27.46%). Not only LineVul, ReGVD, and CodeBERT also have on average lower precision and recall in this dataset

Table 8. Mann-Kendall Trend Test on the ML Performance Tested on Perspective Data

We did the Mann-Kendall test for each model in the 4 datasets, therefore, p_{value} needs to be less than 0.0125 for the test to be significant. There is only one significant increasing trend: Vuldeepecker in their own dataset (NVD Vuldeepecker).

Precision

Model	Dataset	Trend	MK	p
LineVul	Linux	No trend	12	0.17
	OpenSSL	No trend	9	0.07
	Poppler	No trend	0	1.00
	NVD Vuldeep.	No trend	2	0.92
Code2Vec	Linux	No trend	1	1.00
	OpenSSL	No trend	-3	0.65
	Poppler	No trend	0	1.00
	NVD Vuldeep.	No trend	11	0.15
CodeBERT	Linux	No trend	21	0.01
	OpenSSL	No trend	8	0.11
	Poppler	No trend	0	1.00
	NVD Vuldeep.	No trend	11	0.29
ReGVD	Linux	No trend	14	0.11
	OpenSSL	No trend	7	0.22
	Poppler	No trend	0	1.00
	NVD Vuldeep.	No trend	17	0.09
Vuldeepecker	Linux	No trend	4	0.69
	OpenSSL	No trend	2	0.85
	Poppler	No trend	0	1.00
	NVD Vuldeep.	No trend	18	0.06

Recall

Dataset	Model	Trend	MK	p
LineVul	Linux	No trend	2	0.90
	OpenSSL	No trend	-3	0.71
	Poppler	No trend	2	0.73
	NVD Vuldeep.	No trend	14	0.18
Code2Vec	Linux	No trend	14	0.11
	OpenSSL	No trend	-3	0.65
	Poppler	No trend	0	1.0
	NVD Vuldeep.	No trend	9	0.25
CodeBERT	Linux	No trend	2	0.90
	OpenSSL	No trend	4	0.57
	Poppler	No trend	0	1.00
	NVD Vuldeep.	No trend	19	0.06
ReGVD	Linux	No trend	16	0.06
	OpenSSL	No trend	7	0.22
	Poppler	No trend	0	1.00
	NVD Vuldeep.	No trend	21	0.04
Vuldeepecker	Linux	No trend	2	0.89
	OpenSSL	No trend	8	0.18
	Poppler	No trend	0	1.00
	NVD Vuldeep.	Increasing	24	0.01

These three models have an average low recall until 2014 but then their recall increases significantly in 2015 and 2016. Most likely at that time, they are trained with enough retrospective information to identify vulnerability in the next year. Vuldeepecker has a dual behavior from the paper to the field setting. It also has an increasing recall trend from 2013 to 2016. Unlike Vuldeepecker, Code2Vec keeps failing to identify vulnerability as it did in the *R-R* case.

We tested if the models exhibited a positive (increasing) trend of performance across the years (Perspective test set) with the Mann-Kendall trend test. We ran the Mann-Kendall test for each model in the 4 datasets. According to Bonferroni correction, the p_{value} of a test needs to be $< 0.05/4$ *or* 0.0125 to be considered as significant. As shown in Table 8, none of the tests of the precision returns significant, which means we cannot conclude any increasing (or decreasing) trend in the precision of the models when tested in unknown future data, even with more and more retrospective. For the recall, only Vuldeepecker on its own dataset (NVD Vuldeepecker) returns a significant increasing trend.

Finding #2: Observation over the years shows that there is no consistent ML performance of the prediction when tested in the *R-P* case.

8 Additional Extensions

We also ran additional experiments with *seen but believed negatives*. These data points represent the codes available at a certain time of observation t and *considered as* not vulnerable but are actually found vulnerable in the future. These data points are negatives at time t, but then become positive at $t + 1$. Assuming that the data (the code) exists beforehand, we add these positive points as negatives to our retrospective test set. The result of this additional experiment shows that the recall does not change for most models and datasets, but the precision dropped. This happens because the models are still classifying most of the *believed negative* data as vulnerable. This high False Positive at a certain time t would be considered as bad, but they are actually classifying the code correctly (as vulnerable/ positive), just the label was still different at time t.

9 Threat to Validity and Future Works

9.1 Validation Case

For our preliminary validation, we chose vulnerability detection as a case study. We chose this case study as vulnerability detection has been a popular field, but the ML model's performance still needs to be improved to find vulnerabilities in a real-world setting [Chakraborty et al. 2021]. However, we believe that our methodology can be applied to any ML model evaluation with a time-based dataset, and we plan to do more evaluations with other cases in the future.

9.2 Validation Dataset

We only applied our methodology to 4 datasets (3 extracted from BigVul [Fan et al. 2020] and 1 from Vuldeepecker [Li et al. 2018]) in our validation.

Quality. There is a possibility that the chosen datasets may have some wrong labels [Croft et al. 2023]. Obviously, the quality does indeed affect the ML algorithm's performance. Our focus is to show that no matter the dataset or model, if one still tests on the entire dataset (retrospectively), one will get a better result than one would obtain in the field. The wrong labels would therefore not impact the phenomenon we want to measure: they would be wrong when considered during the retrospective analysis, the properly timed training phase, and the properly timed testing phase. Their bias will therefore be uniform across all cases considered in this paper.

Completeness. The chosen datasets have a limitation in that they do not have information on the code availability date. Therefore, we use labeling dates as a proxy for availability dates during implementation. We acknowledge the limitation of this proxy as a possible threat to validity. However, as the assumption that the availability dates are less than or equal to the labeling dates holds, we argue that this proxy is sufficient to show the difference between the retrospective and perspective views.

Future works. In the future, we plan to validate it using other datasets, such as the ProjectKB [Ponta et al. 2019] dataset (which has not been evaluated for ML vulnerability detection but contains mostly Java code and vulnerabilities) to see how the method works for another language. We believe that our methodology can be applied to any time-based dataset, language-agnostic, and we encourage the research community to develop and use more time-based datasets to validate the ML models in the perspective vs. retrospective setting.

9.3 Validation Models

We only validate our methodology by running 5 ML models: Vuldeepecker [Li et al. 2018], Code2Vec [Alon et al. 2019], ReGVD [Nguyen et al. 2022], CodeBERT [Feng et al. 2020], and LineVul [Fu and Tantithamthavorn 2022]. While any comparison is of course limited, this selection covered a sufficient variety of different ML techniques used (RNN/BLSTM, MLP/AST, GNN, and Transformers).

LLMs. We did not include LLMs in this study because ours is a comparison between retrospective and prospective studies. As of today, we can only run a prospective study with LLMs, because available pre-trained LLMs already know the past, and we cannot "remove the past" from them. Unlike other ML-based models, which can be trained from scratch on a given dataset, we cannot retrain LLMs from scratch because we do not have a snapshot of the entire internet in 2022. To be able to use an LLM, we must use an LLM pre-trained in 2021 and test on vulnerability data of 2022, then take an LLM pre-trained in 2022 and test on 2023, and so on. As of today, we do not have enough temporal data to make robust conclusions, but it would be an interesting future study to be done 2 years from now with at least 5 years of observation.

Table 9. Summary of Findings and Implications.

RQ	Main Finding	Implication for Research
RQ1	If you do not account for perspective the ML model will be trained on data points that were either not correctly classified (up to 1/3 of the vulnerabilities seen in the previous period) or not even available (up to 81.5% wrt those seen the previous period)	Today's cat is tomorrow's dog: label of codes can change over time as new vulnerabilities are being found, both in training and testing set. The ML models' performance on retrospective (*Retrospective Training - Retrospective Testing (R-R)*) might not fully capture performance on perspective (*Retrospective Training - Perspective Testing (R-P)*).
RQ2	Observation over the years (Figure 5) and Mann-Kendall test results show no consistent ML performance of the prediction when tested in the *R-P* case.	Presenting ML's performances in time as *trends* will provide a more realistic understanding, as it also shows how much retrospective information affects ML's performance.

Only 5 models: future replication. Even if we cannot include all available models, our methodology is general and applicable to other ML models. Moreover, the trend in the performance of each evaluated model consistently shows a difference between performance with full retrospective information and the performance at a point in time where retrospective information is unavailable. Testing with more models will become a replication study in future work.

9.4 Validation Metrics

We decided to use precision and recall as metrics to show the comparison among the results. One can choose to use another metric, but all metrics are constructed from basic metrics: TP, FP, TN, and FN, which are dependent on each other, i.e., when the TP increases, the FN will decrease, etc. This resulted in metrics that are interrelated with each other by a linear equation. Of course, it is still possible that in particular cases, the precision and recall *always* cancel each other, which removes the fluctuation from F1. However, these cases are extremely unlikely in complex experiments with ML tools and real data.

9.5 Reasons of Fluctuating Trends

In this paper, we only showed and compared the trends between *Retrospective Testing (R-R)* and *Perspective Testing (R-P)*. We did not investigate further the reasons why there is no trend and why the results fluctuate over the years. Yet, our goal is to show that **this is a problem** one will encounter in reality. Models overfitting and intrinsic or seasonal trends (a known phenomenon since [Joh and Malaiya 2009]) can all be subject to further investigation. As far as we know, research on ML result stability in software engineering is limited and remains a future research direction.

10 Implications of the Findings

Table 9 summarizes the implications and the main findings from each research question. Our first result (§7.1) shows that ML models' performance could (most probably) be different when used in the *Perspective Testing (R-P)* compared to the testing result which in most papers is in the *Retrospective Testing (R-R)*. This result on ML for source code is aligned with the result by [Jimenez et al. 2019] on code metrics. Research evaluating ML performance should consider adopting our methodology to consider perspective information in their evaluation and present results that reflect more than one scenario at a given point in time. Our result in RQ2 (§7.2) shows that the trends are informative in visualizing the impact of retrospective information on ML performance.

A key issue to debate is our choice of *not* using the complete information for the testing dataset. The motivation behind this choice is that vulnerabilities are discovered sometimes *after years* from the release. This has been a consistent finding since the milk or wine study [Ozment and Schechter 2006], which introduced the notion of foundational vulnerability (present since the very beginning

of a project). Retrospective discovery is common across systems [Nguyen et al. 2016] and ecosystems [Hu et al. 2024]. Even looking at the famous log4j vulnerability, CVE-2021-4104 refers to a version that reached EoL in 2015, an after-life vulnerability [Massacci et al. 2011]. Consider a developer running a model on log4j in 2015. Even if by mistake the model flagged the vulnerable fragment (there was no evidence of similar vulnerabilities at the time), it would have been considered a false positive. The model would have been considered a failure for five long years.

11 Conclusions

In this work, we propose a methodology to produce a timeline of partial, time-actual information datasets from a complete information (retrospective) dataset. For each time point in the timeline, we generate a training set and 2 testing sets: one simulating performance on the *Retrospective Training - Retrospective Testing (R-R)* (testing with the information available until the chosen time point) and one simulating the performance on the *Retrospective Training - Perspective Testing (R-P)* (testing until the next time point). The former corresponds to the case of a researcher arguing on paper for the model to be deployed based on the available information at the time, the latter corresponds to performance experienced (or perceived) on the field for the newly developed software, which will be classified by the ML model until the next time point.

The key, real-world issue we try to capture with this work is the fact that labels change over time. This is probably the most critical difference with datasets used for image classification, where most ML algorithms have been developed. An image of a 'true' cat will never become an image of a 'true' dog. The application of ML methods to software engineering and security vulnerability detection, in particular, must take this difference into account when evaluating models.

We validated the methodology by using time slots that are one year apart from each other. The resulting test datasets show that the number of vulnerabilities an ML model has to identify can be really different from the ones they are tested with. This finding supports our next finding: when tested using Perspective datasets, the ML models performed worse when compared to the results from using the retrospective dataset. This result is partly aligned with the results found in [Jimenez et al. 2019] on code metrics, which broke the full information available to training by releases, but still uses the full information to test the success of models trained on partial information.

From these findings, we want to raise awareness of the impact of retrospective information when evaluating ML models with any time-based dataset and the possibility of evaluating ML models using perspective datasets instead. We also found that ML models have no significant trend as more and more retrospective information is added, as one might expect, as shown by the results of the Mann-Kendall tests and the visual representation of the results. We believe that our methodology also applies to other use cases, e.g., bug/ defect detection and commit classification, therefore, we plan to do more validation in the future with different datasets and ML models and use cases.

Acknowledgments

This work was partly funded by the EU under the H2020 Program AssureMOSS (Grant n. 952647) and the Horizon Europe Program Sec4AI4Sec (Grant n. 101120393), by the Italian Ministry of University and Research (MUR) under the P.N.R.R. – NextGenerationEU grant n. PE00000014 (SERICS subproject COVERT), and by the Dutch Research Council (NWO) under the grant NWA.1215.18.006 (Theseus) and grant KIC1.VE01.20.004 (HEWSTI).

Credit Statements.

Conceptualization: RP, FM; *Methodology:* RP, FM;
Software: RP, YF; *Validation:* RP, YF;
Formal analysis: RP, YF, FM; *Investigation:* RP, YF;

Data Curation: RP, YF; *Writing - Original Draft:* RP, YF;
Writing - Review & Editing: RP, YF, FM;
Visualization: RP; *Supervision:* FM;
Project administration: FM; *Funding acquisition:* FM;

12 Data Availability Statement

We made available the replication packages of this work in Zenodo [Paramitha et al. 2025a,b,c,d]. There are four repositories, each one for each dataset we used to validate our methodologies: one [Paramitha et al. 2025a] from NVD Vuldeepecker [Li et al. 2018] and three from BigVul [Fan et al. 2020]: `linux` [Paramitha et al. 2025b], `openSSL` [Paramitha et al. 2025c], and `poppler` [Paramitha et al. 2025d]. Each replication package includes: (1) *Code* that (a) implements our methodology to generate the datasets, (b) runs the models in our validation, and (c) generates charts from the ML evaluation results, (2) *Datasets* contains (a) the original datasets from Vuldeepecker [Li et al. 2018] and BigVul [Fan et al. 2020] and (b) the datasets we created using our methodology, (3) *Pretrained-models* that we generated during our evaluation, and (4) *Results* of our evaluation.

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Kristina Toutanova, Anna Rumshisky, Luke Zettlemoyer, Dilek Hakkani-Tur, Iz Beltagy, Steven Bethard, Ryan Cotterell, Tanmoy Chakraborty, and Yichao Zhou (Eds.). Association for Computational Linguistics, Online, 2655–2668. doi:10.18653/v1/2021.naacl-main.211

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. doi:10.1145/3290353

Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988. doi:10.48550/arXiv.2010.09470

Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39. doi:10.1145/3475960.3475985

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021). doi:10.1109/TSE.2021.3087402

Haipeng Chen, Rui Liu, Noseong Park, and VS Subrahmanian. 2019. Using twitter to predict when vulnerabilities will be exploited. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data Mining*. 3143–3152. doi:10.1145/3292500.3330742

Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (Hong Kong, China) *(RAID '23)*. Association for Computing Machinery, New York, NY, USA, 654–668. doi:10.1145/3607199.3607242

Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133. doi:10.1109/ICSE48619.2023.00022

Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 433–436. doi:10.1145/1321631.1321702

Nelson Tavares De Sousa and Wilhelm Hasselbring. 2021. JavaBERT: Training a Transformer-Based Model for the Java Programming Language. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*. 90–95. doi:10.1109/ASEW52652.2021.00028

Geanderson E dos Santos and Eduardo Figueiredo. 2020. Commit Classification using Natural Language Processing: Experiments over Labeled Datasets.. In *CIbSE*. 110–123.

Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512. doi:10.1145/3379597.3387501

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of*

the Association for Computational Linguistics: EMNLP 2020, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. doi:10.18653/v1/2020.findings-emnlp.139

Michael Fu and Chakkrit Tantithamthavorn. 2022. Linevul: A transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 608–620. doi:10.1145/3524842.3528452

Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud. 2021. Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model. *Information and Software Technology* 135 (2021), 106566. doi:10.1016/j.infsof.2021.106566

Hazim Hanif and Sergio Maffeis. 2022. Vulberta: Simplified source code pre-training for vulnerability detection. In *2022 International joint conference on neural networks (IJCNN)*. IEEE, 1–8. doi:10.1109/IJCNN55064.2022.9892280

Jinchang Hu, Lyuye Zhang, Chengwei Liu, Sen Yang, Song Huang, and Yang Liu. 2024. Empirical Analysis of Vulnerabilities Life Cycle in Golang Ecosystem. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639230

Matthieu Jimenez, Yves Le Traon, and Mike Papadakis. 2018. [Engineering Paper] Enabling the Continuous Analysis of Security Vulnerabilities with VulData7. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 56–61. doi:10.1109/SCAM.2018.00014

Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. 2016. Vulnerability Prediction Models: A Case Study on the Linux Kernel. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–10. doi:10.1109/SCAM.2016.15

Matthieu Jimenez, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon, and Mark Harman. 2019. The importance of accounting for real-world labelling when predicting software vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 695–705. doi:10.1145/3338906.3338941

HyunChul Joh and Yashwant K Malaiya. 2009. Seasonal variation in the vulnerability discovery process. In *2009 International Conference on Software Testing Verification and Validation*. IEEE, 191–200. doi:10.1109/ICST.2009.9

Johnb110. 2022. VDPython. URL: https://github.com/johnb110/VDPython.

René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA, 437–440. doi:10.1145/2610384.2628055 Tool demo.

Christian Kästner. 2022. From Models to Systems: Rethinking the Role of Software Engineering for ML. URL: https://www.youtube.com/watch?v=_m-m90S_4Gg.

Triet Huynh Minh Le, David Hin, Roland Croft, and M Ali Babar. 2021. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 717–729. doi:10.1109/ASE51524.2021.9678622

Stanislav Levin and Amiram Yehudai. 2017. Boosting automatic commit classification into maintenance activities by utilizing source code changes. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. 97–106. doi:10.1145/3127005.3127016

Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 157, 12 pages. doi:10.1145/3597503.3639218

Zhen Li, Deqing Zou, Jing Tang, Zhihao Zhang, Mingqian Sun, and Hai Jin. 2019. A comparative study of deep learning-based vulnerability detection system. *IEEE Access* 7 (2019), 103184–103197. doi:10.1109/ACCESS.2019.2930578

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022a. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (July 2022), 2244–2258. doi:10.1109/tdsc.2021.3051525

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings 2018 Network and Distributed System Security Symposium*. doi:10.14722/ndss.2018.23158 arXiv:1801.01681 [cs]

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2022b. VulDeePecker. URL: https://github.com/CGCL-codes/VulDeePecker.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. doi:10.48550/arXiv.1907.11692 arXiv:1907.11692 [cs.CL]

Lucy Ellen Lwakatare, Aiswarya Raj, Ivica Crnkovic, Jan Bosch, and Helena Holmström Olsson. 2020. Large-scale machine learning systems in real-world industrial settings: A review of challenges and solutions. *Information and software technology* 127 (2020), 106368. doi:10.1016/j.infsof.2020.106368

Cláudia Mamede, Eduard Pinconschi, and Rui Abreu. 2023. A transformer-based IDE plugin for vulnerability detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 149, 4 pages. doi:10.1145/3551349.3559534

Qiheng Mao, Zhenhao Li, Xing Hu, Kui Liu, Xin Xia, and Jianling Sun. 2024. Towards Effectively Detecting and Explaining Vulnerabilities Using Large Language Models. *arXiv preprint arXiv:2406.09701* (2024). doi:10.48550/arXiv.2406.09701

Tina Marjanov, Ivan Pashchenko, and Fabio Massacci. 2022. Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet. *IEEE Security & Privacy* 20, 5 (2022), 60–76. doi:10.1109/MSEC.2022.3176058

Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. 2011. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *International Symposium on Engineering Secure Software and Systems*. Springer, 195–208. doi:10.1007/978-3-642-19125-1_15

Andreas Mauczka, Florian Brosch, Christian Schanes, and Thomas Grechenig. 2015. Dataset of developer-labeled commit messages. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 490–493. doi:10.1109/MSR.2015.71

Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruett, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. VulChecker: Graph-based Vulnerability Localization in Source Code. In *31st USENIX Security Symposium, Security 2022*. https://dl.acm.org/doi/abs/10.5555/3620237.3620604

Van-Anh Nguyen, Dai Quoc Nguyen, Van Nguyen, Trung Le, Quan Hung Tran, and Dinh Phung. 2022. ReGVD: Revisiting graph neural networks for vulnerability detection. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 178–182. doi:10.1145/3510454.3516865

Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering* 21 (2016), 2268–2297. doi:10.1007/s10664-015-9408-2

Chao Ni, Xin Yin, Kaiwen Yang, Dehai Zhao, Zhenchang Xing, and Xin Xia. 2023. Distinguishing Look-Alike Innocent and Vulnerable Code by Subtle Semantic Representation Learning and Explanation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1611–1622. doi:10.1145/3611643.3616358

Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1565–1569. doi:10.1145/3468264.3473122

NIST. 2024a. National Vulnerability Database. https://nvd.nist.gov/

NIST. 2024b. National Vulnerability Database: Vulnerability API. https://nvd.nist.gov/developers/vulnerabilities

NIST. 2024c. Software Assurance Reference Dataset. https://samate.nist.gov/SRD/index.php

Vadim Okun, Aurelien Delaitre, Paul E Black, et al. 2013. Report on the static analysis tool exposition (sate) iv. *NIST Special Publication* 500 (2013), 297. doi:10.6028/NIST.SP.500-297

Andy Ozment and Stuart E Schechter. 2006. Milk or wine: does software security improve with age?. In *USENIX Security Symposium*, Vol. 6. 10–5555. doi:10.5555/1267336.1267343

Jalaj Pachouly, Swati Ahirrao, Ketan Kotecha, Ganeshsree Selvachandran, and Ajith Abraham. 2022. A systematic literature review on software defect prediction using artificial intelligence: Datasets, Data Validation Methods, Approaches, and Tools. *Engineering Applications of Artificial Intelligence* 111 (2022), 104773. doi:10.1016/j.engappai.2022.104773

Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shanping Li. 2023. Fine-grained Commit-level Vulnerability Type Prediction by CWE Tree Structure. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 957–969. doi:10.1109/ICSE48619.2023.00088

Ranindya Paramitha, Yuan Feng, and Fabio Massacci. 2025a. Replication package on Zenodo Part 1 (NVD Vuldeepecker Dataset). Zenodo link. If the link does not work, copy and paste the following link https://doi.org/10.5281/zenodo.8207883.

Ranindya Paramitha, Yuan Feng, and Fabio Massacci. 2025b. Replication package on Zenodo Part 2 (LINUX Dataset). Zenodo link. If the link does not work, copy and paste the link in the following link https://doi.org/10.5281/zenodo.10960662.

Ranindya Paramitha, Yuan Feng, and Fabio Massacci. 2025c. Replication package on Zenodo Part 3 (OPENSSL Dataset). Zenodo link. If the link does not work, copy and paste the link in the following link https://doi.org/10.5281/zenodo.10966117.

Ranindya Paramitha, Yuan Feng, and Fabio Massacci. 2025d. Replication package on Zenodo Part 4 (POPPLER Dataset). Zenodo link. If the link does not work, copy and paste the following link https://doi.org/10.5281/zenodo.14713143.

Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 426–437. doi:10.1145/2810103.2813604

Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and C´edric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories*. doi:10.1109/MSR.2019.00064

Niklas Risse and Marcel Böhme. 2024. Uncovering the Limits of Machine Learning for Automatic Vulnerability Detection. In *USENIX Security Symposium 2024*. 19. doi:10.5555/3698900.3699138

John Ruscio and Benjamin Lee Gera. 2013. Generalizations and extensions of the probability of superiority effect size estimator. *Multivariate behavioral research* 48, 2 (2013), 208–219. doi:10.1080/00273171.2012.738184

Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762. doi:10.1109/CCWC60891.2024.10427574

Nasir Safdari. 2018. Multi-Class-Text-Classification—-Random-Forest. Github. https://github.com/nxs5899/Multi-Class-Text-Classification----Random-Forest

J. SayyadShirabad and T.J. Menzies. 2005. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada. http://promise.site.uottawa.ca/SERepository

Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006. doi:10.1109/TSE.2014.2340398

Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. 2013. Data quality: Some comments on the nasa software defect datasets. *IEEE Transactions on software engineering* 39, 9 (2013), 1208–1215. doi:10.1109/TSE.2013.11

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Trans. Software Eng.* 37 (11 2011), 772–787. doi:10.1109/TSE.2010.81

Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2237–2248. doi:10.1109/ICSE48619.2023.00188

Suraj Yatish, Jirayus Jiarpakdee, Patanamon Thongtanunam, and Chakkrit Tantithamthavorn. 2019. Mining software defects: Should we consider affected releases?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 654–665. doi:10.1109/ICSE.2019.00075

Bin Yuan, Yifan Lu, Yilin Fang, Yueming Wu, Deqing Zou, Zhen Li, Zhi Li, and Hai Jin. 2023. Enhancing Deep Learning-based Vulnerability Detection by Building Behavior Graph Model. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2262–2274. doi:10.1109/ICSE48619.2023.00190

Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2a: A dataset built for ai-based vulnerability detection methods using differential analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 111–120. doi:10.1109/ICSE-SEIP52600.2021.00020

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems*. 10197–10207. doi:10.5555/3454287.3455202

Received 2024-09-11; accepted 2025-01-14