symmetry

MDPI

Article

Beyond Accuracy: A Cross-Benchmark Investigation of Label Consistency and Reliability in AI-Based Software Vulnerability Detection

Maksim Iavich [1,*], Tamari Kuchukhidze [2] and Audrius Lopata [2]

[1] Department of Computer Science, Caucasus University, 0102 Tbilisi, Georgia
[2] Faculty of Informatics, Kaunas University of Technology, LT-44249 Kaunas, Lithuania; tamari.kuchukhidze@ktu.lt or tamari.kuchukhidze@gmail.com (T.K.); audrius.lopata@ktu.lt (A.L.)
* Correspondence: miavich@cu.edu.ge

Abstract

Machine-learning research on software vulnerability detection measures progress against a small set of public benchmark datasets, treated as a shared yardstick for comparing models. This approach rests on the untested premise that benchmarks agree: if two datasets have the same function, they give it the same label. We test this assumption by comparing four popular C and C++ benchmarks against one another on their complete data. Label agreement is a symmetric relation, but its strength is uneven. The two benchmarks that derive labels from CVE-linked fixing commits disagree on only 0.68% of 75,019 shared functions, whereas a third disagrees with them on 51.21% and 41.28%. Leakage, in contrast, is directed: we identify 1083 instances in which training data from one benchmark reappears in test data from another, with different counts in each direction. A manual check of 30 contradictory functions traces the disagreements to commit-level labeling and to treating functions left unchanged by a fix as safe. The costs are asymmetric: contradictory labels impair training only slightly, while up to 3.12% of a test set carries a contradicted label, shifting measured accuracy by up to two points. Therefore, reported accuracy reflects both model capabilities and benchmark selection.

Keywords: vulnerability detection; benchmark datasets; label agreement; symmetric relations; data quality; asymmetry; data leakage; empirical software engineering

1. Introduction

Software vulnerabilities continue to be one of the most serious flaws in contemporary systems, and the number of them being reported keeps rising. In 2024 alone, the CVE Program collected around 40,000 new vulnerability reports [1]. Traditional detection techniques struggle to keep pace. Static analyses scale but produce high false-positive rates. Dynamic techniques are precise but cover only the paths they execute [2]. This gap has motivated sustained interest in learning-based detection, in which a model learns to recognize vulnerable code directly from examples [3,4].

The field has converged on a particular formulation of the task. Early deep-learning systems such as VulDeePecker [3] operated on code slices, and related slice- and token-based systems followed, including SySeVR [5], the large-scale Draper corpus of Russell et al. [6], and line-level successors such as LineVul [7]. The dominant approach today treats detection as binary classification of a single function, a formulation adopted by 88 percent of recent papers [4]. Two families of models dominate: graph-based methods,

Check for updates

Received: 24 September 2026
Revised: 17 October 2026
Accepted: 20 October 2026
Published: 23 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

with Devign [8] the canonical example, and transformer-based methods that fine-tune a pretrained encoder such as CodeBERT [9] or GraphCodeBERT [10], with later variants including UniXcoder [11] and the security-specific VulBERTa [12]. Reported accuracies reach the mid-nineties [13]. Section 2 examines these methods and the benchmarks that evaluate them in detail.

Progress in this field is measured almost entirely against a small number of public benchmark datasets: the synthetic Juliet Test Suite [14]; Devign [8], redistributed as the defect-detection task of CodeXGLUE [15]; Big-Vul [16]; and DiverseVul [17]. When a new method is proposed it is evaluated on one or more of these and compared against numbers other papers obtained on the same datasets, so the benchmarks are, in effect, the shared measuring instrument of the field.

This shared use is what makes the field cumulative. A reported improvement is meaningful only relative to earlier results on the same benchmark, a leaderboard ranks methods only if every entry is scored against the same labels, and a claim that one method generalizes better than another usually rests on results from several benchmarks. Each of these practices treats the benchmarks as interchangeable measuring instruments: if one dataset calls a function vulnerable, another dataset containing the same function is expected to agree. When that expectation fails, differences between reported results can reflect the benchmarks rather than the models, and progress measured on one benchmark may not carry over to another. Whether the benchmarks are consistent with one another is therefore a precondition for comparing results across papers, not a technical detail of dataset construction.

Every such comparison rests on an assumption that is rarely stated and, to our knowledge, has never been tested directly: that benchmarks agree, so that whenever two datasets contain the same function they assign it the same label. Agreement between two datasets is a symmetric relation by definition (if A agrees with B on a function, B agrees with A), so the open question is not its direction but its strength, and whether that strength is close to complete for every pair. The assumption is not obviously safe. Only Juliet obtains labels by construction. The real-world benchmarks infer them from version history, treating a function modified by a security-related commit as vulnerable before the change. That inference is a heuristic, not an oracle. Fixing commits routinely contain refactoring and cleanup alongside the actual patch, commit messages are ambiguous evidence of intent, and different research groups resolve these ambiguities differently. Where two benchmarks make different choices about the same function, they disagree, and nothing in current practice would reveal the disagreement.

Recent work has begun to question benchmark reliability, and the results are sobering. Detectors lose more than half of their measured performance under realistic evaluation [13]; a manual audit found that 20 to 71 percent of vulnerability labels were inaccurate and that duplication rates reached 99 percent [18]; models scoring up to seventy percent accuracy cannot distinguish a vulnerable function from its patched version [19]; and the same model scores 68.26 percent F1 on Big-Vul but only 3.09 percent F1 on PrimeVul [20]. Section 2.3 reviews these studies in detail.

These studies share a structural feature. Each evaluates benchmarks one at a time. Croft et al. [18] measure label consistency within Big-Vul, within Devign, and within Juliet separately, reporting values of 0.999, 0.991, and 0.750 respectively. Ding et al. [20] compare model scores across benchmarks but not the benchmarks' labels against one another. The question of how strongly two independently constructed benchmarks agree on the code they both contain has not been asked. It can be asked, because these datasets are not independent. They mine the same small pool of large, CVE-rich open-source C and C++ projects, chiefly the Linux kernel, QEMU, FFmpeg, and PHP. Identical functions necessarily

recur across them. Each recurrence is a case in which two benchmarks make a claim about the same code, and those claims can be compared.

This paper asks that question and answers it empirically. We conduct a cross-dataset audit of four widely used benchmarks: Juliet [14], Big-Vul [16], the CodeXGLUE defect-detection task derived from Devign [8,15], and DiverseVul [17]. Treating label agreement as a symmetric relation between datasets, we identify every function shared by each pair of complete datasets, compare the labels they assign, and check a sample of the disagreements against the fixing commits. Because leakage, unlike agreement, has a direction, we measure it separately in each direction. We quantify how much training data of one benchmark appears in the test data of another. We test directly whether contradictory labels degrade a trained model. We train six models under a single controlled protocol: four dataset specialists and two combined models. We report every result with confidence intervals and paired significance tests.

Our contributions are as follows.

First, we provide a cross-dataset audit of four major vulnerability benchmarks, measuring function-level overlap between every pair and quantifying cross-dataset leakage, a directed relation whose counts differ by direction. We find 1083 cases in which one benchmark's training data matches another's test data, a total that grows sharply with the number of datasets combined. Second, we measure the strength of label agreement, a symmetric relation, for every pair on the complete datasets: the two curated datasets disagree on only 0.68 percent of the 75,019 functions they share, while a third disagrees with them on 51.21 and 41.28 percent, and relaxed token-level matching confirms the pattern. To our knowledge this is the first direct measurement of cross-benchmark label agreement in this field.

Third, we identify the source of the inconsistency. Agreement tracks labeling methodology rather than dataset age or size: the two benchmarks that derive labels from CVE-linked fixing commits converge, while the one that labels whole commits diverges from both, and a manual check of 30 contradictory functions against their commits identifies the specific mechanisms. Fourth, we assess the impact on evaluation reliability, first with a controlled experiment that trains two otherwise identical models differing only in whether 827 contradictory functions are present, and then by measuring directly how contradictory labels change the scores of every model on every test set. The cost of contradictory labels is asymmetric: the training cost is real but small, whereas the evaluation cost cannot be removed by training. When two benchmarks label a shared function differently, no model that sees only the function can agree with both labels, whichever label is correct in its original context; on our test sets this shifts measured accuracy by up to two points, comparable to the differences on which models are ranked. As a result, reported accuracy includes a component that reflects benchmark choice rather than model capacity.

The remainder of this paper is organized as follows. Section 2 reviews learning-based vulnerability detection, the benchmark datasets, and prior work on data quality. Section 3 states our research questions. Section 4 describes the four datasets and their preparation. Section 5 describes the audit procedure, the models, and the statistical methods. Section 6 reports results. Section 7 discusses their implications for the field. Section 8 states threats to validity. Section 9 concludes.

2. Related Work

2.1. Machine Learning for Vulnerability Detection

Automated vulnerability detection has transitioned from handwritten rules to learned models [21]. Most current work approaches the job as binary classification of a single function: given a function's source code, a model predicts if it includes a security vulnerability.

According to a recent survey, 88 percent of the most advanced approaches released in the previous five years use this formulation [4]. We accept it as well because it is the standard formulation and the benchmarks reviewed here are based on it.

Two families of models account for most recent work. The first represents a function as a graph over its syntax, control flow, and data flow and classifies that structure; Devign [8] is the canonical example, applying a gated graph neural network to a composite code representation. The second treats code as a token sequence and fine-tunes a pretrained encoder. CodeBERT [9] is a bimodal encoder pretrained on paired natural language and code, and GraphCodeBERT [10] adds data-flow information during pretraining. Both are common backbones for vulnerability classifiers and are the models fine-tuned in this study. Later models such as UniXcoder [11] unify encoder and decoder pretraining, and domain-specific models such as VulBERTa [12] pretrain on security-related corpora, but the CodeBERT family remains the usual baseline against which new methods are compared. Recent work also fine-tunes code models with multi-task instructions to improve how detectors generalize [22].

Reported accuracies on established benchmarks approach the mid-nineties [13], raising the question of whether such statistics are valid under practical settings. Section 2.3 examines evidence indicating that, for the most part, they do not.

2.2. Benchmark Datasets for Vulnerability Detection

Almost all development in this field is evaluated against a small set of public benchmarks. The meaning of their labels is determined by how each benchmark was constructed, and the source of the vulnerable label is the key distinction between them.

Juliet [14] is synthetic. Published by NIST as part of the Software Assurance Reference Dataset, it consists of programs generated from templates, each pairing a flawed construct with a corrected counterpart, so that labels are assigned by construction rather than inferred. Version 1.1 contains more than eighty thousand C, C++, and Java programs covering 181 flaw types; we use the later C/C++ release. Juliet was designed for testing static analyzers rather than for training classifiers, and its template-generated code is repetitive and unlike production software. Its labels are nevertheless reliable, which makes it a useful controlled baseline.

The real-world benchmarks instead infer labels from version history. Devign [8] collected functions from four large C projects (the Linux kernel, QEMU, Wireshark, and FFmpeg), identified security-related commits, and validated them manually, an effort its authors estimate at about six hundred person-hours. The defect-detection task of CodeXGLUE [15] redistributes the QEMU and FFmpeg portion of these data with a fixed official train/validation/test split, which has made it a de facto leaderboard for the task.

Big-Vul [16] automates the same idea at scale. Its authors crawled the CVE database, followed links to fixing commits across 348 open-source projects, and labeled as vulnerable every function whose lines a fix modified, covering 91 vulnerability types; its released file contains 188,636 functions. DiverseVul [17] represents a newer generation. It collects vulnerability-fixing commits from security issue trackers across a much wider set of projects and adds explicit quality controls, excluding vulnerability-introducing commits through keyword filtering and manual review and deduplicating function bodies by their MD5 hash. The publication describes 18,945 vulnerable and 330,492 non-vulnerable functions, whereas the released file contains 330,492 function records in total, of which 18,945 are labeled vulnerable (Section 5.2), across 150 CWEs, and it covers 295 more projects than all previous datasets combined. Its authors report that deep-learning models still perform poorly on it and attribute this to the difficulty of the task rather than to the data.

CVEfixes [23] differs in kind: rather than a ready-made classification dataset, it is a relational database linking CVE records to their fixing commits across many languages. We attempted to use it as a fifth benchmark and report the attempt as a negative result in Sections 4.5 and 7.4. Other datasets illustrate the variety of labeling strategies: ReVeal [13] was assembled to study realistic evaluation, CrossVul [24] extends commit-derived labeling to more than forty languages at file granularity, and D2A [25] derives labels from differential static analysis instead of commit history.

2.3. Data Quality in Vulnerability Benchmarks

A growing body of work asks whether these benchmarks measure what they are assumed to measure; this literature is the closest to our contribution.

Chakraborty et al. [13] tested state-of-the-art detectors in realistic conditions rather than the typical splits and found performance losses of more than 50%. They assign the decline to training data features such as duplication and incorrect class distributions as well as model design.

The most systematic audit to date is that of Croft et al. [18], who assessed five inherent data-quality attributes across four benchmarks, including Big-Vul, Devign, and Juliet. Manual inspection showed that between 20 and 71 percent of the vulnerability labels in the real-world datasets were inaccurate and that between 17 and 99 percent of data points were duplicated. Removing cross-set duplicates reduced measured performance by up to 82 percent, indicating that a substantial part of reported accuracy reflects leakage rather than detection ability. They also measured label consistency, defined as the absence of identical code with conflicting labels, obtaining 0.999 for Big-Vul, 0.991 for Devign, and 0.750 for Juliet. Every one of these measurements is taken within a single dataset. The duplication findings agree with a broader result for machine learning on code: Allamanis [26] showed that duplicated corpora can inflate reported metrics by up to one hundred percent, which makes deduplication a precondition for meaningful evaluation.

Risse and Böhme [19] approached the question from the model side. Models reaching up to seventy percent accuracy on standard benchmarks could not distinguish a vulnerable function from the same function after patching, which the authors attribute to overfitting on features unrelated to the vulnerability. High benchmark scores can therefore coexist with an almost complete absence of the capability the benchmark is meant to measure.

Ding et al. [20] responded by building PrimeVul with stricter labeling, thorough deduplication, and chronological splitting. Their results provide the most direct evidence that benchmark choice dominates reported performance: the same 7B model scores 68.26 percent F1 on Big-Vul and 3.09 percent F1 on PrimeVul, and GPT-4 with chain-of-thought prompting does not outperform random guessing under their most stringent evaluation. Table 1 situates the present study among these prior studies.

Table 1. This work in the context of prior studies of vulnerability-benchmark quality.

Study	Venue	Focus	Key Finding
Chakraborty et al. [13]	TSE 2022	Realistic evaluation	Performance drops > 50% under realistic settings; attributes it to training-data properties
Croft et al. [18]	ICSE 2023	Within-dataset quality	20–71% labels inaccurate; 17–99% duplication; consistency measured within each dataset
Risse & Böhme [19]	USENIX 2024	Model capability	Models cannot tell a vulnerable function from its patch; >90% of functions need external context

Table 1. *Cont.*

Study	Venue	Focus	Key Finding
Ding et al. [20]	ICSE 2025	Stricter benchmark	Same model scores 68.26% F1 on Big-Vul vs. 3.09% on PrimeVul
This work	—	Between-dataset consistency	First to measure label agreement across benchmarks: 0.68–51.21% disagreement on shared functions of the complete datasets; 1083 cross-dataset leaks; measured training and evaluation costs

Prior work evaluates each benchmark in isolation; this study is the first to measure label consistency between benchmarks.

2.4. The Gap This Paper Addresses

Collectively, these investigations reveal that individual benchmarks contain erroneous labels, significant duplication, and evaluation conditions that inflate assessed performance. Each of them, however, evaluates benchmarks independently, judging quality inside a dataset rather than consistency across datasets. None investigates if two independently produced benchmarks that include the same function use the same label. The question is relevant because these datasets are not independent samples: they mine the same projects, therefore the same function may appear in many datasets, and there is no guaranty that their labels match.

This paper addresses that gap empirically. For every pair of four widely used benchmarks, we measure how many functions they share and how often they disagree on the label. We quantify how much training data of one benchmark leaks into another's test set. We trace the disagreement to labeling methodology and check a sample of disagreements against the fixing commits. To our knowledge, this is the first study to measure label consistency between vulnerability benchmarks rather than within them.

3. Research Questions

Our study is organized around five research questions. The first three concern the benchmarks themselves and can be answered by measurement alone, without training models: whether the datasets agree with one another, how much they overlap, and which of them form a mutually consistent group. The last two require models and concern the consequences: whether inconsistent labels compromise evaluation, and whether training on several benchmarks at once can compensate for their disagreement.

RQ1: How consistent are vulnerability labels across benchmark datasets?

RQ2: How common are duplicated functions, within and across benchmarks?

RQ3: Which datasets provide mutually consistent labels?

RQ4: How do label inconsistencies affect evaluation reliability?

RQ5: Can combined training overcome benchmark disagreement?

Sections 6.3 and 6.4 answer RQ1–RQ3 by measuring the four datasets directly. Section 6.5 addresses RQ4, measuring the effect of contradictory labels first on training, through a controlled experiment, and then on evaluation, using the stored per-function predictions; Section 7.2 discusses the difference between the two. RQ5 is addressed in Sections 6.1 and 6.2, and 6.6 by comparing dataset specialists with combined models on every test set. Section 7 synthesizes the answers, and Section 8 states the limits of what the study supports.

4. Datasets

Our study uses four widely used, publicly available benchmarks for function-level vulnerability detection in C and C++. We selected them to cover the main ways in which

such benchmarks are constructed, so that disagreements between them can be related to construction choices. The Juliet Test Suite represents synthetic code whose labels are correct by construction. Big-Vul represents curated real-world code whose labels are linked to individual CVE entries. The CodeXGLUE defect-detection task, derived from Devign, represents real-world code labeled through a commit-level judgment of whether a change is security-related, and it is the benchmark most often used for leaderboard comparison. DiverseVul represents the modern, large-scale generation, collected from security issue trackers with explicit attention to label quality and deduplication. Three of the four mine overlapping open-source projects, which is what makes a cross-benchmark comparison possible, and all four pose the same task: binary classification of a single function as vulnerable or safe. Table 2 summarizes their statistics after preparation, and Table 3 compares how each assigns labels.

Table 2. The four benchmark datasets after preparation.

Dataset	Type	Lang.	Projects	Rows (Train/Test)	Vulnerable (Train/Test)
Juliet 1.3	Synthetic	C/C++	--- (118 CWEs)	70,092/15,932	43.5%/45.2%
Big-Vul	Real, curated	C/C++	348	17,060/4264	50.0%/50.0%
CodeXGLUE (Devign)	Real	C	2 (QEMU, FFmpeg)	21,854/2732	45.8%/45.9%
DiverseVul	Real, modern	C/C++	797	28,997/7337	50.5%/47.9%

Rows are given as train/test, and the last column gives the vulnerable share of each split. CodeXGLUE additionally ships an official 2732-row validation set, which we preserved unmodified.

Table 3. How the four benchmarks assign labels.

Benchmark	Label Source	Commit-History Extraction	Manual Validation	Label Conflicts in the Release	Conflict Handling in This Study
Juliet	Template construction: each test case pairs a flawed (bad) and a fixed (good) variant	None (synthetic)	Not needed: labels correct by construction	Arise only after comments and label-bearing names are removed	None; exact duplicates removed, first copy kept (Section 4.1)
Big-Vul	CVE entries linked to fixing commits; functions changed by the fix are vulnerable, other functions in the changed files are safe	Automatic crawl of CVE references to commits (348 projects)	None reported	Present (1314 functions carry both labels)	Conflicting functions removed
CodeXGLUE (Devign)	Commit-level: functions changed by commits judged security-related are vulnerable, functions changed by other commits are safe	Keyword-filtered commits of QEMU and FFmpeg	Commit-level: four security researchers, about 600 person-hours [8]	Present (86 functions carry both labels)	None; official split kept unchanged
DiverseVul	Vulnerability-fixing commits from security issue trackers; functions changed by the fix are vulnerable, unchanged functions are safe	Crawl of security issue trackers linked to commits (797 projects)	Not for the full dataset; 60% of 50 sampled vulnerable labels accurate [17]	Present (743 functions carry both labels)	Conflicting functions removed

Label conflicts are counted on all public datasets at the whitespace-insensitive level of Section 5.2 (Section 6.4). Sources include the original papers [8,14–17] and the study's preparation scripts.

Where we create a split ourselves, preparation follows a common discipline: we remove label leakage and duplicates, balance the classes by keeping every vulnerable

function together with an equal-sized random sample of safe ones, and split 80/20; for Big-Vul and DiverseVul we also remove intra-dataset label conflicts, in which identical code carries both labels. The implementation differs by dataset and is described in the subsections below, which also report the residual train–test overlap found by the independent audit of Section 5.2, because such residuals are themselves a finding about benchmark quality. CodeXGLUE is deliberately left untouched to preserve comparability with published results.

4.1. Juliet Test Suite

Juliet [14] is a synthetic test suite published by NIST's Software Assurance Reference Dataset (SARD). Its C/C++ release contains 105,183 files in 118 CWE categories, generated from templates as matched pairs of flawed and corrected functions. It offers explicit CWE coverage, near-balanced classes, and labels that are correct by construction, but its template code is highly repetitive, so a model can score well by exploiting surface artifacts rather than vulnerability semantics.

That risk is real, and neutralizing it is the central preparation task for this dataset. Inspection of the raw suite revealed several channels through which the label leaks into the input, and our preparation script removes them in a fixed order. It first skips multi-file test cases, whose logic is split across several files. It then strips all comments, which removes the FLAW and FIX markers. Next it extracts each function whose name marks it as flawed (bad) or fixed (good), takes the label from that name, discards trivial wrapper functions that only call other functions, and removes a leading static keyword, because in Juliet only the safe variants are static, so the keyword would act as a hidden label. It then renames every identifier that encodes the label, including the bad and good names, to a neutral token, consistently within a function, which also neutralizes the label inside string literals. It drops functions shorter than 60 characters. It removes exact duplicate functions, keeping the first copy. Finally it splits the data by test-case family, defined as the file name without its variant number, so that variants of one test case cannot appear on both sides of the split. A final scan confirms that no label token survives. The prepared dataset contains 70,092 training functions (30,461 vulnerable and 39,631 safe) and 15,932 test functions (7208 vulnerable and 8724 safe).

The identifier renaming and the duplicate removal interact in a way that matters for Sections 6.5 and 8. Juliet has no separate conflict-resolution step. Once comments and label-bearing names disappear, a flawed function and its fixed counterpart can become textually identical, typically when the flaw lies in a function it calls rather than in the function itself. Of the 150,316 functions extracted, 3628 distinct functions occurred with both labels. Deduplication kept one copy of each with whichever label came first, which gave 3516 vulnerable copies and 112 safe copies. A whitespace-only variant of the same effect survives deduplication, because removing a comment can leave a blank line in one variant but not in the other. At the whitespace-insensitive level used in our audit, the prepared training set contains 234 opposite-label groups (490 functions) and the test set contains 35 groups (72 functions, 0.45 percent of the test set), and a further 22 test functions have a training twin with the opposite label. These conflicts are an artifact of preparation, not a disagreement between benchmarks, and Section 6.5 tests whether they affect the training experiment.

For transparency, the post-preparation audit still found 412 exact and 413 near-identical train-test pairs, about 2.6% of test rows. These are identical template code recurring across different test-case families, something family-grouped splitting cannot detect. We left these residuals in place for the single-dataset setting and removed them during the construction of the merged corpora, which Section 5 describes.

4.2. Big-Vul

Big-Vul [16] is a curated real-world dataset mined from CVE-linked vulnerability-fixing commits, the pre-fix version of every function modified by such a commit is labeled vulnerable. We obtained the distributed file, MSR_data_cleaned.csv (about 10 GB, 188,636 functions), through its maintained Kaggle mirror. Its labels are traceable to individual CVE entries, which has made it one of the most widely used benchmarks in the field [27]; its main weakness is the mining assumption itself, which Section 6.4 tests directly. The raw dataset is severely imbalanced, with vulnerable functions a small minority.

Preparation reduced the 36-column file to code and label, which are func_before and vul. It removed 453 exact duplicates and 140 intra-dataset label conflicts, which are identical code appearing with both labels. It retained all 10,662 vulnerable functions with an equal-sized random sample of safe ones and split the resulting balanced 21,324 functions 80/20 at random, stratified by label; unlike the Juliet and DiverseVul splits, this split is not grouped by commit. Before deduplication, every function was truncated to its first 3000 characters, well beyond the 512-token input limit of the models. The training set contains 17,060 functions, with 8530 vulnerable and 8530 safe. The test set contains 4264 functions, with 2132 vulnerable and 2132 safe. The post-preparation audit found 27 exact pairs, which is 0.63% of the test set, and 39 near-identical pairs, which is 0.91%, residual train-test pairs. These are small enough to be negligible for the single-dataset setting and removed in the merged corpora.

4.3. CodeXGLUE (Devign)

The defect-detection task of CodeXGLUE [15] is built on Devign [8] and contains C functions from QEMU and FFmpeg. It ships a fixed official split of 21,854, 2732, and 2732 functions for training, validation, and test, which allows direct comparison across papers. We confirmed that the separately distributed Devign data is identical to the CodeXGLUE defect-detection data, so results on one apply to the other.

Precisely because comparability is this benchmark's main value, we applied no preparation at all. The official split is preserved exactly. The training set contains 21,854 functions, of which 10,018 are vulnerable and 11,836 are safe. The test set contains 2732 functions, of which 1255 are vulnerable and 1477 are safe. Our audit did find that the official split itself contains 10 exact and 13 near-identical train-test duplicate pairs. We deliberately retained them because altering the split would break comparability with every published result. We instead record the observation as part of the data-quality analysis in Section 6.3.

4.4. DiverseVul

The largest and most current of the four is DiverseVul [17], which has 330,492 real-world C and C++ function records from security-issue-linked changes across 797 projects and 150 CWE categories in its public file (diversevul_20230702.json), of which 18,945 (5.7%) are classified as vulnerable. Where Devign samples two projects, DiverseVul samples an ecosystem, and its authors document that deep-learning models struggle on it, which makes it the most demanding realistic testbed available to us.

Preparation followed the shared discipline. We deduplicated, removed 743 intra-dataset label-conflict groups, retained all 18,167 surviving vulnerable functions with an equal-sized random sample of safe ones, giving 36,334 total functions, and split 80/20 grouped by originating commit so that related functions stay on one side. Because the split is grouped rather than purely random, the per-split class ratios are approximate. The training set contains 28,997 functions, of which 14,652 are vulnerable and 14,345 are safe. The test set contains 7337 functions, of which 3515 are vulnerable and 3822 are safe. Notably, DiverseVul is the only dataset of the four whose prepared split shows zero exact and

zero near-identical train-test pairs in our audit. This is consistent with its quality-focused construction, a point Section 6 returns to.

4.5. CVEfixes, Excluded from the Benchmark Suite

CVEfixes [23] is a 48 GB relational database mapping CVE entries to their fixing commits across many languages. We verified its eight tables against the published statistics and initially intended to use it as a fifth benchmark, but 62.9 percent of its function records are JavaScript, much of it minified. After filtering to C and C++ (and, in one variant, four languages) and constructing leak-free splits grouped by CVE and commit, five controlled training runs across two task formulations produced the same signature: training loss fell steadily while validation accuracy stayed at chance, indicating memorization without generalization. Under a pre-registered stop rule no model was retained and the held-out test set was never evaluated. We therefore report the attempt as a documented negative result (Section 7.4) rather than as a fifth benchmark.

4.6. Relationships Between the Datasets

These four datasets do not represent independent samples. Big-Vul, CodeXGLUE, and DiverseVul mine overlapping projects such as Linux, QEMU, FFmpeg, and PHP. Big-Vul and DiverseVul also share CVE heritage, thus same routines appear across datasets, sometimes with different names. Sections 6.3 and 6.4 quantify the overlap and disputes, while Section 5.3 explains how the combined training corpora are devoid of both.

5. Methodology

This section describes how we prepared the four benchmark datasets, how we audited them for shared code and contradictory labels, how we built the two merged training corpora, the six models and their training configurations, the evaluation protocol, and the statistical methods we used to quantify uncertainty and compare models. A guiding principle runs through all of it: official benchmark splits are never altered, so that our results remain comparable with published work, while any split we create ourselves is cleaned and constructed to avoid train-test leakage. Every preparation and audit step writes a machine-generated report.

5.1. Dataset Preparation

Section 4 describes the preparation of each dataset. Figure 1 summarizes the full pipeline, from the raw benchmarks to the merged training corpora of Section 5.3.

Figure 1 shows the data preparation pipeline. Each raw benchmark passes through dataset-specific cleaning, leak-channel removal for Juliet, deduplication and class balancing for Big-Vul and DiverseVul, and no modification for CodeXGLUE's official split, followed by a family-grouped 80/20 split, a cross-dataset fingerprint audit, and construction of the merged v2 and v3 training corpora.

5.2. Function Matching and the Cross-Dataset Audit

Every comparison of code in this study, both within a dataset and across datasets, uses the same matching method. We normalize a function by removing whitespace and compute an MD5 fingerprint of the result. We use two normalization levels. At level 1, runs of whitespace are collapsed to a single space. At level 2, all whitespace is removed. Level 2 is the stricter of the two and matches a superset of what level 1 matches. Every decision to drop a row, described below, uses level 2. This matching is deliberately conservative. It identifies functions that are identical up to whitespace, but it does not attempt to match functions that differ in variable names, comments, or formatting. A consequence worth stating plainly is that every overlap count and every disagreement count we report is a

lower bound on the true amount because subtler duplicates and contradictions escape an exact-fingerprint method. To evaluate how much the counts depend on this option, we perform the agreement analysis with a looser, token-level matcher. It eliminates comments and layout and renames every identifier that is not a C or C++ keyword by order of first appearance, such that functions varying only in names, comments, or formatting match; literals are maintained, thus functions that differ in a constant such as a buffer size do not. Functions with less than 30 tokens are omitted from relaxed matching to avoid inconsequential collisions.

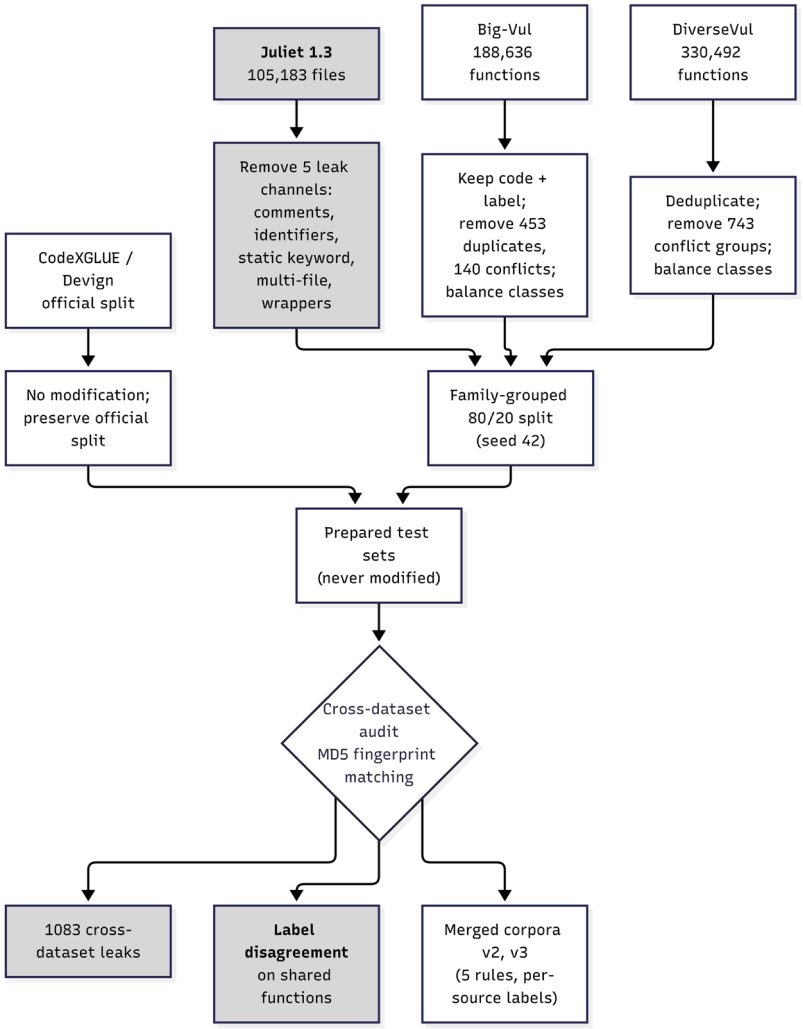


Figure 1. Data preparation pipeline.

The cross-dataset audit runs before any merged corpus is built and changes no data. It answers two questions. First, does the training portion of one dataset overlap the test portion of another? Section 6.3 reports the resulting leakage.

Second, when two datasets contain the same function, do they agree on its label? To make every pair comparable, we answer this question on one uniform basis: the complete published version of each dataset (all 188,636 Big-Vul functions, all 330,492 DiverseVul functions, the full CodeXGLUE training, validation, and test sets, and all 86,024 extracted Juliet functions), before any sampling or splitting. In particular, we use the Big-Vul file MSR_data_cleaned.csv (188,636 functions, of which 10,900 are labeled vulnerable), the DiverseVul release diversevul_20230702.json (330,492 function records, of which 18,945 are labeled vulnerable; the original publication [17] reports 18,945 vulnerable and 330,492 non-vulnerable functions, but the released file contains 330,492 records in total), the CodeXGLUE

defect-detection release (27,318 functions, of which 12,460 are labeled vulnerable), and the 86,024 Juliet functions extracted in Section 4.1. No further filtering is applied. After identical functions are collapsed, 185,555, 326,907, and 27,228 unique functions remain for Big-Vul, DiverseVul, and CodeXGLUE. Of these, 1314, 743, and 86 contain both labels and are removed as explained below. Every count reported in Section 6.4 can be replicated since the replication package (Data Availability Statement) contains the fingerprints and labels of every function. The unit of comparison is a unique function, identified by its fingerprint. We exclude a function that carries both labels inside one dataset from that dataset's comparisons and count it separately, so that every reported disagreement is between two datasets rather than within one. For each pair, the denominator is the number of functions present in both datasets with a single label in each, and the numerator is the number of those functions whose labels differ. For comparison with the training data, we also report the same measure on the prepared splits. The audit preserves the full source and both labels of every disagreeing function so that the contradictions can be inspected directly.

To assess which labels are supported by evidence, we examined a random sample of 30 disagreeing functions, ten from each pair of real-world datasets (fixed seed 42), drawn from the complete-dataset comparison. For each function we retrieved from the project's public repository the commit behind the vulnerable label and the commit associated with the safe label, and applied a fixed rule. The vulnerable label is supported if that commit fixes a security weakness in this specific function. If the commit does not fix a vulnerability in this function, for example because it moves, traces, refactors, or cleans up code, fixes a functional bug, or edits the function only as a side effect of a fix made elsewhere, the vulnerable label is not supported by the examined commit. The function could have a vulnerability that this commit did not address, therefore the lack of evidence for the vulnerable label does not prove that the function is safe. Only positive evidence, a commit that addresses a security flaw in the same code, contradicts a safe classification. If the evidence is inadequate to support either judgment or the modification is a robustness patch with unknown security relevance, the situation is ambiguous. We retrieved and pre-screened the commits under this rule, and Appendix A lists all 30 functions, the commits, and the verdicts, so that each verdict can be checked.

5.3. Construction of the Merged Training Corpora

We build two merged training corpora. The first, combined v2, merges Juliet, Big-Vul, and CodeXGLUE. The second, combined v3, adds DiverseVul to make a four-dataset corpus. Both follow the same five rules.

The first rule is that the test sets are never modified. The merge scripts do not write test files at all, and evaluation always uses the original, untouched test sets. The second rule removes from the combined training pool any function whose level-2 fingerprint matches any test function of any dataset. This is how the leakage measured in the audit is prevented from entering training. The third rule removes contradictions and duplicates in two steps. First, any group of identical functions that carries more than one label across sources is removed in its entirety because a function labeled both ways cannot teach a consistent lesson. Then the remaining identical functions are deduplicated to a single copy, with the kept copy chosen by a fixed source priority. That priority affects only which source label a retained function carries and never affects evaluation since evaluation is always reported per dataset. The fourth rule is that every retained row keeps a column recording its source dataset, which is what makes per-dataset reporting possible. The fifth rule carves out validation data, about ten percent of the cleaned training pool, per source with a fixed random seed. It groups by test-case family for Juliet and by commit for CodeXGLUE and

stratifies by label for Big-Vul and DiverseVul. Because the training pool is deduplicated first, the training and validation portions share no fingerprints.

For combined v2 the merge begins with 109,006 training functions. It removes 482 that match a test set, 718 that belong to cross-source label-conflict groups, and 699 duplicates, leaving 96,341 training functions. For combined v3 the merge begins with 138,003 training functions. It removes 1544 that match one of the four test sets, 827 that belong to 391 label-conflict groups, and 2740 duplicates, leaving 119,603 training functions together with 13,289 validation functions. The 827 contradictory functions removed for combined v3 are not merely discarded. They are the subject of a dedicated experiment in Section 6.5, which trains two otherwise identical models with and without them to measure directly what contradictory labels cost.

5.4. Models and Training

We train six models: four dataset specialists, each trained and evaluated on one dataset, and two combined models trained on the merged corpora and evaluated on every test set.

The Juliet specialist is a small convolutional network of roughly two million parameters. Its architecture comprises a word-embedding layer, a one-dimensional convolution, global max pooling, a dense layer with dropout, and a single sigmoid output. It uses a custom tokenizer with a ten-thousand-word vocabulary that treats punctuation as separate tokens, a fixed input length of 512 tokens, class weights to balance the two classes, and early stopping on validation performance. We keep this architecture for Juliet and disclose the difference openly. Because the real-world datasets use transformer models, the Juliet figure should be read as benchmark-specific performance rather than as a direct architectural comparison. Retraining Juliet with a transformer would improve uniformity but would not change the paper's conclusions, which concern benchmark quality rather than the strongest possible model, and we note a transformer-based Juliet run as possible future work.

The three real-world specialists are transformer models. Big-Vul and DiverseVul use CodeBERT [9], and CodeXGLUE uses GraphCodeBERT [10], which incorporates data-flow information during pretraining. Both are pretrained encoders of about 125 million parameters. Each is fine-tuned with a learning rate of 2×10^{-5} for a small number of epochs at a maximum sequence length of 512 tokens, with the best epoch retained by validation accuracy. The maximum length is a hard limit of this model family rather than a tuning choice.

The two combined models use GraphCodeBERT [10] with a gentler configuration suited to the larger corpora: a learning rate of 1×10^{-5} with warmup over more epochs. Combined v2 is trained on 96,341 functions, and combined v3 on 119,603 functions with 13,289 held out for validation. The original combined v3 run was ended by the time limit of the training environment after the epoch-3 checkpoint had been saved; resuming from that checkpoint for one epoch under the identical configuration recovered the epoch-4 model, whose validation accuracy of 85.73 percent matched the value the original run had reached. The combined v3 of record is this epoch-4 model. As a result, the two combined runs did not finish the same number of epochs: combined v3 finished four epochs, whereas combined v2 trained for six epochs and achieved its greatest validation accuracy at epoch 4. Although they were chosen from varying numbers of finished epochs, both models of record are hence epoch-4 checkpoints. All training used a single cloud T4 GPU and a fixed random seed of 42.

All use a maximum sequence length of 512 tokens, the hard limit of the transformer family; the Juliet CNN uses the same length for its input. Random seed 42 throughout. Table 4 summarizes the full training configuration for all six models.

Table 4. Training configuration for the six models.

Model	Architecture	LR	Max Len	Epochs	Notes
Juliet specialist	CNN (~2 M params)		512	—	class-weighted, early stopping
Big-Vul specialist	CodeBERT (125 M)	2×10^{-5}	512	3	best epoch kept
CodeXGLUE specialist	GraphCodeBERT (125 M)	2×10^{-5}	512	4	best epoch kept
DiverseVul specialist	CodeBERT (125 M)	2×10^{-5}	512	3	best epoch kept
Combined v2	GraphCodeBERT (125 M)	1×10^{-5}	512	6	warmup 10%
Combined v3	GraphCodeBERT (125 M)	1×10^{-5}	512	4 (3 + 1 recovered)	warmup 10%, seed 42

5.5. Evaluation Protocol

All models are evaluated on the original test sets, which are never modified. Each specialist is evaluated on its own dataset's test set. Each combined model is evaluated on every available test set, which for combined v3 means all four. Every evaluation records, for each test function, the model's predicted probability of the vulnerable class alongside the true label. These per-function records are the basis for all reported metrics and for the paired statistical comparisons in Section 5.6. Because predictions are stored per function rather than only in aggregate, every number in the results can be recomputed and every pair of models can be compared on exactly the same inputs.

Section 6.5 uses this same protocol for the contradictory-label experiment. Two combined models were trained, one on the cleaned four-dataset corpus of 119,603 functions and one on the identical corpus with the 827 contradictory functions restored, giving 120,430. The per-source composition of the two training sets is otherwise identical, and both models were evaluated on the same four untouched test sets. Because contradictory functions are, by definition, duplicated code carrying opposite labels, we held them out of the validation split so that every contradictory pair reached training with both of its labels intact. The validation set therefore contains no contradictions and is identical in size for both models and only the training data differs: Model B's training set contains the same 119,603 functions as Model A's plus the 827 contradictory ones.

5.6. Statistical Methods

All metrics are computed from the stored per-function predictions, with the vulnerable class as positive. Every measure is reported with a 95 percent bootstrap confidence interval (2000 resamples of test rows, seed 42), and every disagreement rate with a Wilson interval [28]. Following common guidance for statistical testing in software engineering [29], we compare two models on the same test set with the exact two-sided McNemar test [30] on the rows that exactly one of them classifies correctly, together with Cohen's h [31] as an effect size (below 0.2 negligible, below 0.5 small, below 0.8 medium). The area under the ROC curve uses the Mann–Whitney rank formulation [32] with average ranks for ties. We apply Holm's step-down correction [33] within each family of tests: the ten specialist-versus-combined comparisons of Section 6.2 and the four comparisons of the contradiction experiment in Section 6.5. For the experiment of Section 6.5 we also report the minimum detectable effect, the smallest accuracy difference that a McNemar test with the given test-set size and observed rate of discordant predictions would detect with 80 percent power at the 5 percent level.

6. Results

We organize the results around the research questions. Sections 6.1 and 6.2 report model performance and compare combined models with dataset specialists. Sections 6.3 and 6.4 turn to the datasets themselves, quantifying how much they overlap and how often they contradict each other. Section 6.5 measures what contradictory labels cost, separately for training and for evaluation, and Section 6.6 draws the results together on the gap between real-world and synthetic performance. Confidence intervals, tests, and effect sizes follow Section 5.6.

6.1. Individual Model Performance

Table 5 reports every model on every test set it was evaluated on, with accuracy, precision, recall, F1, and area under the ROC curve. The single most visible pattern is a sharp divide by dataset type rather than by model. On synthetic and curated code the specialists score above ninety percent accuracy, with the Juliet specialist at 94.98 and the Big-Vul specialist at 92.31. On real-world code drawn from production projects the same class of models scores in the sixties, with the CodeXGLUE specialist at 66.18 and the DiverseVul specialist at 69.50. These real-world figures are not a shortcoming of our training. They match the performance range that the DiverseVul authors and subsequent work report for transformer models on this task [17]. Figure 2 shows the specialist results visually.

Table 5. Every model on every test set.

Model	Test Set	n	Accuracy	Precision	Recall	F1	AUC
Juliet (specialist)	Juliet	15,932	94.98	94.39	94.53	94.46	98.77
Big-Vul (specialist)	Big-Vul	4264	92.31	94.83	89.49	92.08	97.24
CodeXGLUE (specialist)	CodeXGLUE	2732	66.18	63.78	61.04	62.38	72.75
DiverseVul (specialist)	DiverseVul	7337	69.50	69.08	65.78	67.39	76.74
Combined v2	Juliet	15,932	93.23	93.05	91.91	92.48	98.60
Combined v2	Big-Vul	4264	90.27	90.36	90.15	90.26	96.40
Combined v2	CodeXGLUE	2732	63.32	60.60	57.61	59.07	69.13
Combined v3	Juliet	15,932	93.57	93.29	92.44	92.86	98.79
Combined v3	Big-Vul	4264	89.66	88.45	91.23	89.82	96.33
Combined v3	CodeXGLUE	2732	62.88	62.70	47.41	53.99	68.25
Combined v3	DiverseVul	7337	68.58	70.21	59.80	64.59	74.15

All values are percentages. The positive class is vulnerable. Confidence intervals for each value are summarized in the text. Combined v3 is the recovered epoch-4 model.

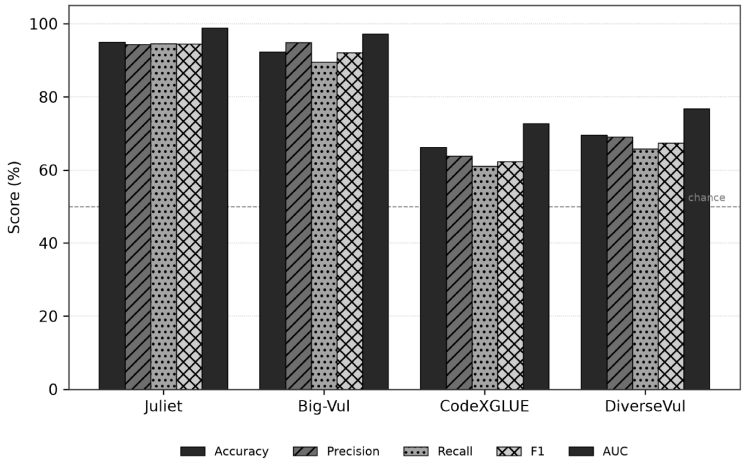


Figure 2. Specialist model performance by dataset.

The area under the ROC curve, which does not depend on a decision threshold, makes the divide even starker: 98.77 for Juliet and 97.24 for Big-Vul against 72.75 for CodeXGLUE and 76.74 for DiverseVul, with confidence intervals that are far from overlapping.

The Juliet specialist, a convolutional network roughly sixty times smaller than the transformers, achieves the highest accuracy in the study. This reflects how easily synthetic template code is classified by surface pattern rather than the merit of the architecture, and it is a further reason to read the Juliet figure as benchmark-specific (Section 5.4).

Figure 2 shows accuracy, precision, recall, F1, and AUC for each dataset's specialist model. Scores are high on Juliet (synthetic) and Big-Vul, and fall sharply on the real-world CodeXGLUE and DiverseVul datasets, where accuracy falls to 66.18 and 69.50 percent and AUC to 72.75 and 76.74 percent: well above chance, but far below the synthetic and curated datasets.

6.2. Combined Models Versus Specialists

A natural question is whether one model trained on several datasets can replace the separate specialists. Table 6 compares the two combined models with each specialist on the specialist's own test set. On every dataset except DiverseVul, the specialist retains a statistically significant advantage, with p-values between 3.7×10^{-17} and 5.7×10^{-3}, but every Cohen's h is at or below 0.093, which is negligible: the combined models give up a few points of accuracy and never collapse. Figure 3 shows these comparisons. Rather than focusing on the effect of combining datasets, these comparisons compare two complete training configurations: the specialists use CodeBERT (Big-Vul, DiverseVul), GraphCodeBERT (CodeXGLUE), or a convolutional network (Juliet), whereas both combined models use GraphCodeBERT with a different learning rate and schedule (Table 4). The difference between a specialist and a combined model may therefore reflect architecture, recipe, and training data. Only the controlled experiment in Section 6.5 keeps the entire training configuration unchanged. The two combined models share architecture and learning rate, but they differ in training data and the number of completed epochs (Section 5.4), thus their comparison is not free of this ambiguity either.

Table 6. Paired comparisons on identical test rows.

Comparison (A vs. B)	Test Set	Diff. A Minus B [95% CI]	p	Cohen's h	Verdict
Juliet vs. Combined v2	Juliet	+1.75 [+1.34, +2.16]	3.7×10^{-17}	0.075	real
Juliet vs. Combined v3	Juliet	+1.41 [+0.99, +1.83]	4.1×10^{-11}	0.061	real
Big-Vul vs. Combined v2	Big-Vul	+2.04 [+1.22, +2.91]	8.5×10^{-7}	0.072	real
Big-Vul vs. Combined v3	Big-Vul	+2.65 [+1.78, +3.54]	3.5×10^{-9}	0.093	real
CodeXGLUE vs. Combined v2	CodeXGLUE	+2.86 [+0.88, +4.83]	5.7×10^{-3}	0.060	real
CodeXGLUE vs. Combined v3	CodeXGLUE	+3.29 [+1.24, +5.23]	2.0×10^{-3}	0.069	real
DiverseVul vs. Combined v3	DiverseVul	+0.91 [−0.14, +1.94]	0.094	0.020	tie
Combined v2 vs. Combined v3	Juliet	−0.34 [−0.57, −0.11]	4.7×10^{-3}	0.014	real (v3 higher)
Combined v2 vs. Combined v3	Big-Vul	+0.61 [−0.21, +1.45]	0.161	0.020	tie
Combined v2 vs. Combined v3	CodeXGLUE	+0.44 [−1.46, +2.23]	0.674	0.009	tie

The difference is model A minus model B in percentage points, with its bootstrap 95 percent interval. The p-value is the exact two-sided McNemar p-value. Cohen's h is the effect size. A verdict of "tie" means the difference is not statistically significant at the 5 percent level. Combined v3 is the recovered epoch-4 model.

The most important comparison is on DiverseVul, the hardest and most realistic dataset, where the four-dataset combined model and the specialist are statistically indistinguishable: 68.58 against 69.50 percent, a difference of 0.91 points (95% CI −0.14 to 1.94, $p = 0.094$). On the dataset that best represents real software, the combined configuration costs nothing that we can detect.

Figure 3 shows accuracy of each specialist alongside the combined v2 (three datasets) and v3 (four datasets) models on the same test set. The combined models trail the specialists by only a few points, and on DiverseVul the four-dataset model and the specialist are statistically indistinguishable (69.50 vs. 68.58, $p = 0.094$).

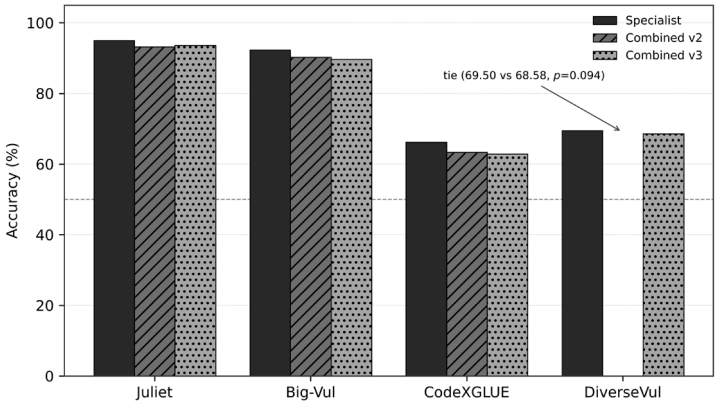


Figure 3. Combined models versus dataset specialists.

The two combined models are also close to each other (ties on Big-Vul, $p = 0.161$, and CodeXGLUE, $p = 0.674$, and a small gain for combined v3 on Juliet). This comparison discusses two training configurations rather than separating the impact of adding DiverseVul since the two runs differ in both training data and the number of completed epochs (Section 5.4). Within these configurations, the larger four-dataset corpus came at no measurable cost on the datasets that the three-dataset model already covered, although it does not close the real-world gap.

Because Table 6 reports ten comparisons, we apply Holm's step-down correction to the family of tests. Under the correction the significant specialist-versus-combined results remain significant, and the ties remain ties, so no verdict in the table changes in substance. We report exact p-values throughout so the effect of the correction is transparent.

6.3. Cross-Dataset Overlap and Leakage

Using the fingerprint matching from Section 5.2, we discover that the four datasets are far from independent. Across all four, 1083 training functions from one dataset are almost comparable to test functions from another (Table 7). Leakage is a directional relationship, with two directions: 572 Big-Vul training functions match DiverseVul test functions, whereas 459 DiverseVul training functions match Big-Vul test functions. These two approaches account for the vast majority of clashes, indicating common projects and CVE heritage.

Table 7. Cross-dataset leakage.

Train\Test	Juliet	Big-Vul	CodeXGLUE	DiverseVul	Row Total
Juliet	—	0	0	0	0
Big-Vul	0	—	6	572	578
CodeXGLUE	0	11	—	20	31
DiverseVul	0	459	15	—	474
Total into each test set	0	470	21	592	1083

The practical hazard is obvious: if a researcher concatenates these benchmarks into a single training set, the model will be tested on functions it has already encountered without

warning. The issue worsens with the number of datasets: a previous audit of the first three revealed 17 collisions, and adding DiverseVul increased the total to 1083. Figure 4 depicts the whole matrix.

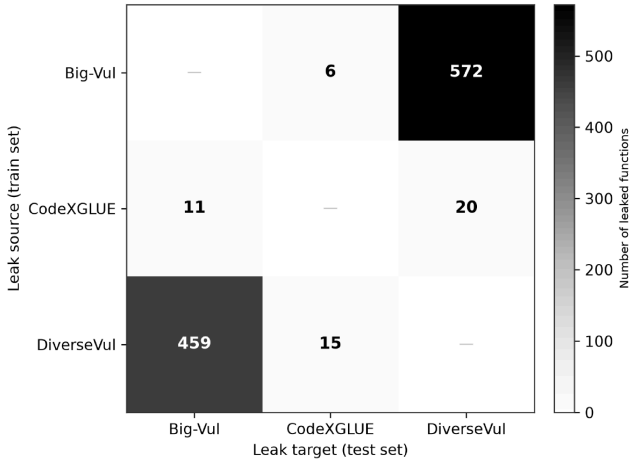


Figure 4. Cross-dataset train → test leakage.

In Figure 4 each cell counts training functions of the row dataset that near-identically match a test function of the column dataset, for a total of 1083 leaks. The overlap is concentrated between Big-Vul and DiverseVul (572 and 459), which share projects and CVE ancestry.

Overlap also exists inside individual datasets (Section 4): three of the four contain train–test duplication, including ten exact duplicate pairs in CodeXGLUE's official split, whereas DiverseVul's prepared split has none.

6.4. Label Disagreement Across Datasets

The central finding of this paper concerns contradiction, not overlap. When two datasets contain the same function, do they agree on whether it is vulnerable? Table 8 answers this for every pair of complete datasets on the uniform basis defined in Section 5.2, with Wilson 95 percent confidence intervals, and repeats the measurement with relaxed token-level matching and on the prepared splits used for training.

Table 8. Label disagreement on shared functions, by matching method and scope.

Dataset Pair	Shared	Disagree	Rate [95% CI]	Shared (Token)	Rate (Token)	Shared (Prep.)	Rate (Prep.)
Big-Vul vs. DiverseVul	75,019	513	0.68% [0.63, 0.75]	66,065	0.73%	3126	1.15%
CodeXGLUE vs. DiverseVul	688	284	41.28% [37.66, 45.00]	773	37.65%	153	26.14%
Big-Vul vs. CodeXGLUE	373	191	51.21% [46.15, 56.24]	440	43.86%	45	33.33%
Juliet vs. any other	0	0	n/a	–	–	0	n/a

Shared, Disagree, Rate: complete released datasets, whitespace-insensitive finger-print (level 2), with Wilson 95 percent intervals. Unit: unique function. Shared: functions present in both datasets with a single label in each; functions carrying both labels inside one dataset are excluded (Section 5.2). Rate: disagreements divided by shared functions. Token level: comments removed and identifiers renamed, functions of at least 30 tokens, complete real-world datasets. Prep.: the train and test splits used in this study, in which Big-Vul and DiverseVul are class-balanced subsamples. Juliet, being synthetic, shares no functions with the real-world datasets.

Agreement is symmetric by definition, but its strength varies widely between pairs, and the variation identifies one dataset as an outlier. Big-Vul and DiverseVul, the two datasets that derive labels from CVE-linked fixing commits, agree almost perfectly: of the 75,019 functions they share, they disagree on only 0.68 percent (95% CI 0.63–0.75).

CodeXGLUE, by contrast, disagrees with both, on 41.28 percent of the 688 functions it shares with DiverseVul and on 51.21 percent of the 373 it shares with Big-Vul. Juliet shares no code with the real-world datasets. The disagreement therefore does not spread evenly across pairs; it concentrates in the pairs that involve one benchmark.

Given that Big-Vul and DiverseVul exhibit significant imbalance, with around 5.7 percent of functions labeled vulnerable, a low overall disagreement rate could in principle reflect agreement on safe functions alone. It does not. Of the 75,019 functions they share, 3169 are labeled vulnerable by both, 71,337 are labeled safe by both, and 513 carry conflicting labels; among the 3682 shared functions that at least one of them labels vulnerable, they agree on 86.1 percent, and the chance-corrected agreement (Cohen's kappa) is 0.92. For the pairs involving CodeXGLUE, kappa is 0.19 against DiverseVul (96 vulnerable in both, 308 safe in both, 284 conflicting) and 0.07 against Big-Vul (23, 159 and 191). The direction of the conflicts is also systematic: in 477 of the 513 Big-Vul–DiverseVul conflicts, Big-Vul labels the function safe and DiverseVul vulnerable, and in 257 of the 284 and 185 of the 191 conflicts that involve CodeXGLUE, CodeXGLUE is the benchmark that labels the function vulnerable. The source projects do not explain the pattern either. Every function that CodeXGLUE shares with the other benchmarks comes from QEMU or FFmpeg, yet on the 3892 QEMU and FFmpeg functions that Big-Vul and DiverseVul share, the two disagree on only 0.92 percent (95% CI 0.67–1.28). The differences in agreement therefore persist within the vulnerable class and within the same projects, which supports attributing them to how each benchmark assigns labels.

These conclusions depend neither on the matching method nor on the scope of the comparison. With relaxed token-level matching, which also pairs functions that differ only in identifier names, comments, or formatting, the rates are 0.73, 37.65, and 43.86 percent on 66,065, 773, and 440 shared functions. On the prepared splits, in which Big-Vul and DiverseVul are class-balanced subsamples, the rates are 1.15, 26.14, and 33.33 percent on 3126, 153, and 45 shared functions. In every setting, the Big-Vul–DiverseVul pair agrees an order of magnitude more closely than either pair involving CodeXGLUE. The smallest comparison now rests on 373 functions of the complete datasets, so the Wilson intervals are narrow. The disagreeing functions are core systems code from projects such as QEMU and FFmpeg, not obscure edge cases, and Figure 5 presents the rates for every pair.

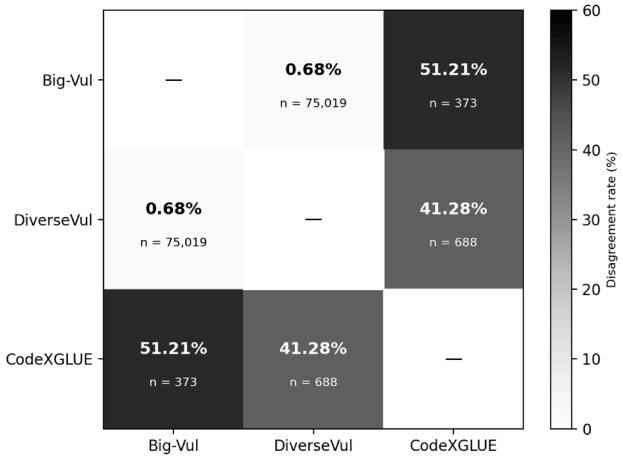


Figure 5. Label disagreement rate on shared functions of the complete datasets.

The manual check of 30 disagreeing functions (Section 5.2; Appendix A) shows which labels are supported by evidence and why. In 14 cases the commit behind the vulnerable label fixes a security weakness in the function itself; in 14 it does not, so that vulnerable label is not supported by the examined commit; and 2 are ambiguous. The evidence differs by

benchmark. Of the 18 sampled CodeXGLUE vulnerable labels, 5 are supported, 11 are not, and 2 are ambiguous, and of the 12 DiverseVul vulnerable labels, 9 are supported and 3 are not. Of the 20 Big-Vul safe labels, 11 are contradicted by positive evidence of a vulnerability, 7 are not contradicted, and 2 are ambiguous. Two mechanisms account for most of the disagreements. First, CodeXGLUE labels at the commit level: every function changed by a commit judged security-related is labeled vulnerable, including functions that the commit only moves to another file, instruments with tracing calls, refactors, or tidies, which account for the 11 unsupported CodeXGLUE labels. Second, a Big-Vul safe label means that the function was present in a fixing commit but not changed by it, not that the function is free of vulnerabilities. In all ten sampled Big-Vul–DiverseVul disagreements, Big-Vul labels safe a function whose identical code was changed by a different commit recorded in DiverseVul, and in eight of them that change fixes a real weakness, which is positive evidence that the code Big-Vul labels safe contained a vulnerability. DiverseVul is not free of error either: two of its three unsupported vulnerable labels come from functions edited only as a side effect of a fix made elsewhere in the same commit. An unsupported vulnerable label is not thereby shown to be wrong, since the function may contain a vulnerability that the examined commit did not fix; the check therefore shows where labels lack support, not which functions are safe. These mechanisms agree with earlier manual audits: Croft et al. [18] attributed most inaccurate Devign labels to irrelevant or clean-up changes, and the DiverseVul authors found 60 percent of sampled DiverseVul vulnerable labels accurate, citing changes to non-vulnerable functions in fixing commits as a main source of error [17]. Because the sample is small, we report counts rather than rates.

Figure 5 shows, for each pair of datasets that share code, the percentage of shared functions assigned opposite labels on the complete datasets; the matrix is symmetric because agreement is. CodeXGLUE is the outlier, disagreeing with Big-Vul on 51.2% and with DiverseVul on 41.3% of shared functions, whereas Big-Vul and DiverseVul disagree on 0.7%.

6.5. The Effect of Contradictory Labels on Training and Evaluation

Section 6.4 demonstrates how benchmarks contradict one another; this section investigates the cost of these conflicts. A contradictory label can function in two ways. It provides inconsistent instances to the model during training, which may impair what it learns; this cost varies per model and may be minimized in theory. In assessment, it alters the score a model receives, regardless of how it was trained. We thus assess the two costs separately: the training cost using a controlled experiment, and the evaluation cost directly from the stored per-function predictions. For the training cost we use the experiment described in Sections 5.3 and 5.5: two combined models identical in architecture, seed, optimizer, learning rate, epochs, and preprocessing, which differ only in whether the 827 contradictory functions are present in training. Because the restored functions also make Model B's training set 827 functions (0.7 percent) larger, the experiment measures the effect of adding functions with contradictory labels to an otherwise identical training set, not the effect of label inconsistency at a fixed training-set size; a size-matched control, in which consistently labeled functions are replaced rather than added, would isolate the latter and is left to future work.

Table 9 reports the outcome. The model trained with contradictions is less accurate on all four test sets, but the size and the statistical strength of the effect differ sharply between them. On Juliet the clean model scores 93.57 percent against 92.54 percent, a difference of 1.03 points whose bootstrap confidence interval, 0.77 to 1.31 points, excludes zero, and whose exact McNemar p-value is 8.3×10^{-14}. Of the 15,932 Juliet test functions, 325 are classified correctly only by the clean model against 161 only by the contradiction-

trained model, a two-to-one imbalance that cannot plausibly arise by chance. On the three real-world test sets the differences are smaller and none is statistically significant. The differences are 0.49 points on Big-Vul with $p = 0.24$, 0.26 points on DiverseVul with $p = 0.59$, and 0.15 points on CodeXGLUE with $p = 0.90$. Under Holm correction across the four tests, only the Juliet result survives.

Table 9. The effect of contradictory labels on training.

Test Set	n	Model A Clean	Model B with Contradictions	Difference [95% CI]	p	Cohen's h	Verdict	MDE (pts)
Juliet	15,932	93.57	92.54	+1.03 [+0.77, +1.31]	8.3×10^{-14}	0.040	significant	0.39
Big-Vul	4264	89.66	89.17	+0.49 [−0.28, +1.27]	0.239	0.016	not significant	1.12
DiverseVul	7337	68.58	68.32	+0.26 [−0.55, +1.17]	0.589	0.006	not significant	1.27
CodeXGLUE	2732	62.88	62.74	+0.15 [−1.54, +1.87]	0.902	0.003	not significant	2.50
Juliet, excl. 79 affected	15,853	93.78	92.71	+1.07 [+0.81, +1.34]	5.8×10^{-15}	0.043	significant	0.38

Both models use the identical recipe, seed, and per-source training composition. The only difference is the 827 contradictory functions present in Model B. Differences are Model A minus Model B in percentage points with bootstrap 95 percent intervals. The p-value is the exact two-sided McNemar test on identical test rows. MDE: minimum detectable effect at 80 percent power (Section 5.6). The last row repeats the Juliet comparison, but excludes the 79 test functions that correspond to a contradicting training function at the token level or belong to an opposite-label group within the test set.

Of the 827 contradictory functions, 472 originate in Juliet, 125 in Big-Vul, 158 in CodeXGLUE, and 72 in DiverseVul. Only 177 of them (75 from Big-Vul, 30 from CodeXGLUE, and 72 from DiverseVul) are contradictions between two benchmarks; the remaining 650 are conflicts inside a single source, including all 472 from Juliet. Measured as a share of each source's own training data, however, the dose is similar for three of the four sources: 0.76 percent for Juliet, 0.85 percent for Big-Vul, 0.81 percent for CodeXGLUE, and 0.30 percent for DiverseVul. A larger dose alone therefore does not explain why the significant effect concentrates on Juliet.

Two factors plausibly explain why only Juliet shows a significant decrease. The first is statistical power. Table 9 reports, for each test set, the minimum detectable effect: 0.39 points on Juliet, but 1.12 on Big-Vul, 1.27 on DiverseVul, and 2.50 on CodeXGLUE. A decrease of the size observed on Juliet, about one point, would therefore not be detectable on any of the real-world test sets. The second is label noise. On the real-world test sets the models already misclassify between 10 and 37 percent of functions, partly because of inconsistent labels, so a small additional source of inconsistency changes little; on Juliet, where accuracy exceeds 93 percent, the same dose stands out. The real-world null results should therefore be read as an upper bound on an effect that our test sets cannot resolve, not as evidence that contradictory labels are harmless there.

Because the Juliet contradictions are an artifact of our preparation (Section 4.1), we tested whether the Juliet result depends on the test functions closest to them. We marked as affected every Juliet test function that matches one of the 827 contradictory training functions at the token level or belongs to an opposite-label group inside the test set, 79 of 15,932 test functions. Excluding them leaves the result unchanged: the clean model scores 93.78 percent against 92.71 percent, a difference of 1.07 points (95% CI 0.81–1.34, $p = 5.8 \times 10^{-15}$) that remains significant under Holm correction, and none of the real-world results changes (Table 9). The degradation is therefore a general effect of inconsistent training labels on the model, not a local effect on a few test functions that resemble the

contradictory ones. It is nevertheless small: Cohen's h is 0.04 on Juliet and below 0.02 elsewhere, negligible by conventional bands. The experiment thus shows that adding a small set of functions with contradictory labels to the training data degrades learning, but the dose at which it does so comes largely from synthetic templates rather than from the real-world benchmarks whose disagreement is this paper's subject.

A dose-response experiment based solely on real-world contradictions is impractical with these benchmarks: the combined training pool contains only 177 cross-benchmark contradictions, or 0.15 percent of it, so a range of doses could only be produced by artificially flipping labels, which would measure random label noise rather than benchmark disagreement. As a result, we provide the per-source dosages and the power analysis above, leaving a bigger real-world experiment for future research.

At the rate at which contradictions occur between the real-world benchmarks, roughly 0.1 percent of the merged training corpus for each, their effect on training is not measurable in our setting. Their effect on evaluation does not depend on training at all, and we measure it next.

For the evaluation cost, we count the number of test functions in each test set whose label contradicts another benchmark, that is, functions that appear in at least one other full dataset with the opposing label. Each such function is scored against one of two incompatible labels. The two labels may be defensible in their original context, for example, when the function is only vulnerable when combined with other code, but a model that receives only the function text cannot satisfy both, so their share of the test set limits the change in accuracy that the choice between the two labels can cause. We also calculate the realized change for each model by rescoring those functions using the other benchmark's label. Table 10 reports both, together with conflicts inside each benchmark.

Table 10. Test functions whose label is contradicted by another benchmark, and the resulting change in accuracy.

Test Set	n	Shared	Contra.	Max. Shift	Shift, Spec.	Shift, Comb. v3	Contra., Token	Within
Juliet	15,932	0	0	0.00	0.00	0.00	–	72/22
Big-Vul	4264	1453	100	2.35	−2.06	−1.74	114 (2.67)	4/21
CodeXGLUE	2732	88	42	1.54	−0.22	+0.29	50 (1.83)	4/11
DiverseVul	7337	1804	229	3.12	−1.02	+0.26	259 (3.53)	0/0

Shared: test functions that occur in at least one other complete dataset (level 2). Contra.: those among them that carry the opposite label there. Max. shift: contradicted functions as a share of the test set, in accuracy points. Shift, spec. and Shift, comb. v3: realized change in accuracy, in points, when the contradicted functions are scored with the other benchmark's label, for the dataset's own specialist and for the four-dataset combined model. Contra., token: contradicted functions under relaxed matching, with the maximum shift in parentheses. Within: within-benchmark conflicts, test functions in opposite-label groups inside the test set/test functions whose twin in the same benchmark's training set has the opposite label.

The shares are small but not negligible. Another benchmark contradicts up to 3.12 percent of a test set on the complete data, and 3.53 percent under relaxed matching. The realized shifts on Big-Vul lie between 1.3 and 2.1 points for every model, the same size as the differences between specialists and combined models in Table 6 (2.04 to 2.65 points). The shift also depends on the model, so the choice of label can change how models compare, not only their absolute scores. On DiverseVul, the specialist, trained on DiverseVul's own labels, loses 1.02 points when we rescore the contradicted functions with Big-Vul's labels, whereas the four-dataset combined model, which also saw Big-Vul training data, gains 0.26 points; the specialist's lead of 0.91 points, which Table 6 already classifies as a tie, becomes a deficit of 0.37 points. On CodeXGLUE, the four-dataset combined models

classify fewer than half of the contradicted functions correctly under CodeXGLUE's own labels and gain when we rescore the functions, indicating that they side with the labels of the other benchmarks. The evaluation cost of contradictory labels is therefore bounded, but it is of the same order as the differences on which models are ranked, and unlike the training cost it cannot be reduced by any change to the model.

6.6. The Gap Between Real-World and Synthetic Performance

Across all models we trained, performance on real-world code remains much lower than performance on synthetic and curated code. Specialists and combined models both achieve only the sixties on CodeXGLUE and DiverseVul, and increasing data or combining benchmarks could not overcome this gap in the setups we tried. Because model capacity was not modified while the other circumstances remained constant, we cannot say that the disparity is independent of the model.

The gap is clearest in recall. On CodeXGLUE the four-dataset combined model attains a recall of 47.41 percent (95% CI 44.64–50.29), so it misses about half of the real vulnerabilities in the test set. On DiverseVul its recall is higher, 59.80 percent, but it still misses two in five vulnerable functions. Figure 6 plots accuracy against recall for every model and test set.

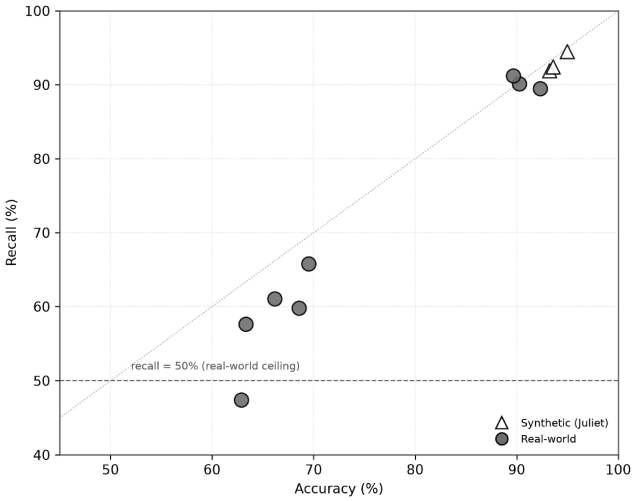


Figure 6. Accuracy against recall for every model and test set.

Points on synthetic Juliet (triangles) sit in the top-right. Every real-world evaluation (circles) falls below, several near or beneath the fifty percent recall line. High accuracy coexists with recall that misses roughly half of real vulnerabilities. This is the real-world gap.

Section 6.4 suggests a likely explanation: a model cannot exceed the consistency of the signal it is trained and tested on, so if benchmarks disagree on a substantial share of shared functions, the gap may be caused partly by the labels themselves. Section 6.5 tests this directly for training and for evaluation.

7. Discussion

Our results describe a field measuring itself with instruments it has not calibrated. Models are compared on benchmarks whose labels disagree with one another, whose contents overlap in ways that invalidate naive combination, and whose difficulty varies so much that a single accuracy figure means almost nothing without knowing which benchmark produced it. This section asks what explains these findings, what they imply for how vulnerability detection research is conducted, and what can be done about it.

7.1. Why the Benchmarks Disagree

The disagreements we measure are not random noise. The three real-world benchmarks label by inference rather than direct analysis: Big-Vul and DiverseVul treat the pre-fix version of a function modified by a CVE-linked commit as vulnerable, while CodeXGLUE infers labels from whether a commit's metadata reads as security-related. Both rest on the assumption that a commit touching a function tells you that function's security status, and that assumption fails in several ways at once. Fixing commits sweep refactoring, renaming, and adjacent cleanup into the vulnerable class; a commit may fix one function while merely adapting a caller, yet both are labeled alike; commit messages are unreliable evidence of intent; and a function safe in one release may be vulnerable in another depending on its callers, which a function-level label cannot represent. Each benchmark resolves these ambiguities differently, and the choices surface as disagreement on shared code—a phenomenon Herzig and Zeller [34] documented when they showed that tangled code changes systematically distort datasets derived from version history.

Our measurements follow this explanation closely. Agreement tracks methodological similarity rather than abstract quality: the two datasets that infer labels from CVE-linked fixes converge, while the one that labels whole commits diverges from both. The manual examination solidifies the mechanisms. Most unsupported CodeXGLUE vulnerable labels in the sample arise because every function altered by a security-related commit is tagged vulnerable, even functions that the commit just moves, traces, refactors, or tidies, resulting in the tangled-change effect mentioned above. Most contradicted Big-Vul safe labels arise because functions left unaltered by a fixing commit are labeled safe, despite the fact that the same code may possess a distinct vulnerability that has been addressed elsewhere. Reliable labels, therefore, require function-level proof that a modification solves a flaw in that function, rather than just evidence that a commit was security related.

The overlap between datasets has a simpler explanation and an equally direct consequence. These benchmarks draw from the same small pool of large, well-audited C and C++ projects, chiefly the Linux kernel, QEMU, FFmpeg, and PHP, because those projects have the CVE histories that make mining feasible. Sampling the same ecosystem repeatedly guarantees that independently built datasets will contain the same functions. Our finding that cross-dataset leakage grew from 17 collisions among three datasets to 1083 among four is a direct consequence. Each new benchmark drawn from the same pool overlaps every existing one.

7.2. What Contradictory Labels Cost

Section 6.4 establishes that benchmarks contradict each other, and Section 6.5 measures what this costs in training and in evaluation; the two costs differ in kind. The training cost is real but bounded. Adding 827 contradictory functions to a corpus of about 120,000 reduced accuracy significantly only on Juliet, by about one point, and the real-world test sets could not have detected an effect of that size. Contradictory labels do impair learning, but at the rate at which real benchmarks contradict one another the cost is small enough that a study would be unlikely to notice it. According to Nie et al. [35], label mistakes can drop conventional detectors' F1 scores by 20.7 percent on average in the worst-case scenario; the considerably smaller effect we see reflects the low rate at which benchmarks contradict one another.

The evaluation cost is of a distinct kind. If two benchmarks label a common function in opposing directions, a model that sees only the function can only agree with one of them, regardless of model, therefore training cannot minimize this portion of the error. This is true even when both labels are defensible in their original contexts, such as when the vulnerability is dependent on calling circumstances that the function text does not include.

Its magnitude is limited by the percentage of contradictory functions in the test set, which is 3.12%, and the realized shifts are about two accuracy points, which is comparable to many reported advances and the gaps between our own models. Because the shift is model-dependent, a label decision can also flip the sign of the difference between two models, as DiverseVul does. Cross-paper comparisons inherit this component invisibly: when one paper reports 66 percent on CodeXGLUE and another 69 percent on DiverseVul, the gap cannot be attributed to the models without accounting for the fact that the two benchmarks disagree on 41 percent of the code they share. The leaderboard culture around these datasets assumes a common yardstick that our measurements show does not exist.

7.3. The Gap on Real-World Code

Every model we trained, from a two-million-parameter convolutional network to fine-tuned transformers with sixty times the capacity, performs in the sixties on real-world code while exceeding ninety percent on synthetic and curated code (Section 6.6). Adding data did not bridge this gap. Because these models differ in architecture and training recipe, they are not a controlled test of model capacity, and we do not assert that larger models cannot narrow it.

Recall makes the practical meaning unmistakable: the four-dataset combined model detects under half of the vulnerable functions in the CodeXGLUE test set, so it is not deployable as a primary safeguard, whatever its headline accuracy suggests. The results constrain the explanations only in part: the gap is not explained by a shortage of training data, since adding a large modern dataset changed nothing, but our experiments do not isolate model capacity. What they do show is that the labels limit what can be learned and measured: a model cannot learn a distinction more reliably than the labels express it, and a benchmark cannot measure a model more precisely than its own labels are consistent.

A further limitation of the task formulation deserves mention. Function-level classification asks whether a function is vulnerable in isolation, but many real vulnerabilities are properties of interactions between functions, of how a function is called, or of assumptions the caller fails to honor. Recent work reports that state-of-the-art models cannot reliably distinguish a vulnerable function from its patched version [19], which is the sharpest available evidence that the signal available at function level may be genuinely insufficient rather than merely difficult to extract. Consistent with this, Jimenez et al. [36] show that properly accounting for real-world labeling, where code labeled non-vulnerable may simply harbor an as-yet-unreported flaw, substantially changes measured performance.

7.4. The CVEfixes Negative Result

Section 4.5 reports the CVEfixes attempt, which belongs in this discussion because it is the clearest illustration of the limits of function-level labels. Across five controlled training attempts, training loss fell while validation accuracy stayed at chance, so under a pre-registered stop rule no model was retained and the held-out test set was never evaluated. The function-level labels derived from this database thus generated no learnable signal under our settings, which is consistent with the weak and inconsistent real-world signals assessed elsewhere in this research, as well as prior results that performance deteriorates under realistic evaluation [13].

7.5. Implications and Recommendations

Our findings suggest several changes in how benchmark-based research is conducted, each following from a measurement in this paper. Results should be reported on more than one benchmark, since a single number cannot be interpreted without knowing which dataset produced it and small cross-paper differences may reflect the benchmarks rather than the models. Datasets should be audited before they are combined, because the

1083 train-test collisions we found are silent contamination that grows rapidly with each dataset added. Labels should be treated as measurements carrying error rather than as ground truth: benchmarks that disagree on 40 percent or more of their shared functions cannot all be correct as function-level labels, so confidence intervals, paired significance tests, and effect sizes are minimum requirements for claims of improvement rather than optional refinements.

For a security task, recall deserves more prominence than accuracy, which can look acceptable on a balanced test set while the model misses half the vulnerabilities. And investment should go to label quality rather than only to model capacity: the two benchmarks whose labels are tied to individual fixes are the ones that agree with each other, so function-level evidence for labels, richer context than a single function, and task formulations that admit uncertainty are likely to matter at least as much as larger models trained on the same signals. Finally, audits should be published alongside models. Every count here was produced by scripts that report their own ledgers, and making such audits routine would let readers see the state of the data a study relies on rather than inferring it.

8. Threats to Validity

This paper argues for more skepticism toward widely used benchmarks, so we apply the same skepticism to our own study, using the conventional four categories of threats.

8.1. Internal Validity

The most consequential internal threat is train-test leakage. Wherever we constructed a split ourselves we grouped it by family (test-case family for Juliet, originating commit for DiverseVul; the Big-Vul split is stratified but not grouped) so near-identical variants cannot straddle the boundary, and when building the merged corpora we removed every training function whose fingerprint matched any test function. We then audited rather than assumed the outcome, reporting residual overlap per dataset in Section 4. Some residual overlap remains: Juliet retains about 2.6 percent of its test set as recurring template code that family-grouped splitting cannot catch, and Big-Vul well under one percent. This inflates the affected accuracies slightly, most plausibly Juliet's, reinforcing rather than undermining our argument that synthetic benchmarks overstate performance.

We deliberately did not clean the CodeXGLUE official split, even though it contains 10 exact and 13 near-identical train-test duplicate pairs, because cleaning it would make our results incomparable with every published result on that benchmark; we report the duplication as a finding instead, so CodeXGLUE results should be read as very slightly optimistic. A third threat is that our Juliet preparation creates label conflicts: removing comments and label-bearing names can make a flawed function and its fixed counterpart identical, and deduplication then keeps whichever label came first (Section 4.1). This affects 3628 distinct functions at extraction and leaves 72 conflicting functions, 0.45 percent, in the Juliet test set. It explains why 472 of the 827 contradictory functions in the four-dataset pool originate in Juliet. It does not affect the cross-benchmark measurements of Section 6.4, because Juliet shares no code with the other datasets, and excluding the affected test functions leaves the Juliet result of Section 6.5 unchanged. Section 6.5 should nevertheless be read as demonstrating the training cost chiefly on contradictions created by our own preparation.

8.2. External Validity

Our results are limited to C and C++ and to function-level classification. We cannot say whether similar disagreement rates hold for other languages, where vulnerability classes and fixing practices differ, and many real vulnerabilities involve interactions across functions that a function-level label cannot express.

The choice of benchmarks is itself a threat. The four datasets are widely used and span synthetic, curated, and modern real-world construction, but they are not a random sample of benchmarks, and a different selection could produce a different outlier or none.

Finally, the real-world benchmarks draw on large, mature, heavily audited projects such as the Linux kernel, QEMU, FFmpeg, and PHP, whose vulnerability patterns may differ from those of smaller or younger codebases.

8.3. Construct Validity

The central construct threat is that we treat benchmark labels as the object of study rather than as ground truth, which is deliberate but has a limit. When two datasets disagree about a function, our main measurements establish that they disagree, not which one is correct. The manual check of 30 functions (Section 6.4; Appendix A) indicates, in a sample, which labels are supported by commit evidence and why. Because an examined commit that fixes no vulnerability does not show that a function is safe, the check identifies missing support rather than proven errors, and some judgments, such as whether a robustness fix counts as a security fix, are debatable; we therefore report counts rather than rates. A rigorous expert study of all opposing roles remains the inevitable next step. Disagreement does not necessarily imply that one label is incorrect: vulnerability might be determined by the calling context or interactions between functions that a function-level label cannot communicate (Section 8.2), thus two opposing labels may both be justifiable in their original contexts. The contradiction does show that the two benchmarks cannot both be used as consistent ground truth for a function-level model. Our core assertion is thus about consistency among benchmarks rather than the validity of any of them.

Our matching method is a second construct threat. Our primary fingerprint detects functions that are identical up to whitespace, but not functions that differ in variable names, comments, or formatting while being semantically the same. We thus conducted the agreement analysis with a relaxed token-level matcher (Section 5.2). It finds identical numbers of common functions and the same pattern of agreement (Table 8), thus our results are not influenced by the cautious option. Even the relaxed matcher misses semantically comparable code with a different structure, so all counts stay lower. Furthermore, the prepared Big-Vul splits were trimmed to 3000 characters per function, which may obscure matches for large functions in the prepared-split analysis; the complete-dataset analysis use untruncated code.

A third construct threat is that accuracy, a reasonable summary on approximately balanced test sets, treats a missed vulnerability and a false alarm alike. We therefore report recall, precision, F1, and AUC throughout Section 6 and lead the real-world argument in Section 6.6 with recall.

8.4. Conclusion Validity

A previous version of this research centered its smallest comparison on 27 Big-Vul-CodeXGLUE functions tested within a training pool. When measured across all datasets, every pair of real-world datasets now has at least 373 common functions, and the Wilson intervals are similarly tight (Table 8). The pattern also holds for relaxed matching and prepared splits, thus the argument we defend, that CodeXGLUE differs severely with two other real-world benchmarks while Big-Vul and DiverseVul agree closely, is not based on a tiny sample size.

For the ten paired model comparisons, we use exact McNemar tests with effect sizes and Holm's correction, under which no verdict in Table 6 changes. Each model was trained once with a single seed, so we cannot fully separate a training condition from run-to-run

variation; multi-seed training is the obvious next improvement. The central findings, however, concern the datasets and involve no training at all.

The Section 6.5 experiment manipulates 827 functions, about 0.7 percent of the training data, and the real-world test sets can detect only differences of 1.1 to 2.5 accuracy points with adequate power (Table 9). Its null results therefore bound the effect of a small contamination rather than proving contradictory labels harmless at any scale. It also compares two single runs, so the three smaller differences should be read as consistent in direction rather than as established effects. The specialist-versus-mixed comparisons also employ different architectures and training recipes: the Juliet specialist is a convolutional network, the Big-Vul and DiverseVul specialists use CodeBERT, and the CodeXGLUE specialist and combination models use Graph-CodeBERT. These comparisons thus contrast training settings rather than isolating the effect of combined training; the two combined models share architecture and learning rate but differ in training data and completed epochs, so only the controlled experiment in Section 6.5 keeps the full training configuration constant.

9. Conclusions

This paper asked whether the benchmarks that machine-learning research uses as a shared yardstick for vulnerability detection are themselves consistent. We audited four widely used C and C++ benchmarks against one another rather than one at a time, and we found they are neither consistent nor independent. When two datasets contain the same function, they frequently assign it opposite labels. The agreement between two datasets is symmetric, but its strength depends on the labeling methodology: the two datasets that derive labels from CVE-linked fixing commits disagree on only 0.68 percent of the 75,019 functions they share, whereas CodeXGLUE, which labels whole commits, disagrees on 51.21 and 41.28 percent, respectively. A manual check of 30 contradictory functions reveals two mechanisms: commit-level labels that mark functions as vulnerable when a commit simply moves, traces, or tidies them, and safe labels for functions that a fix leaves unchanged despite the fact that the same code contains a vulnerability fixed elsewhere. All four datasets draw from the same small pool of CVE-rich projects, which also produced 1083 cases where one dataset's training data matches another's test data, a directed relation whose counts differ by direction. That contamination grows sharply as more datasets are combined. The functions in question are core systems code, not obscure edge cases, and anyone merging datasets to get more training data would not see the leakage.

These properties have measurable consequences, and they hurt evaluation more than training. A model trained on contradictory labels was only slightly less accurate than one trained without them, and at the rate the real-world benchmarks actually disagree, we could not detect the effect. Evaluations are influenced differently. When two benchmarks label the same function differently, no function-level model can agree with both labels, regardless of training, even if each label is valid in its own context. Up to 3.12% of a test set is impacted, which alters assessed accuracy by up to two points and can reverse the sign of the difference between two models. As a result, reported accuracy includes a component that indicates which benchmark was picked rather than how effectively the model finds vulnerabilities. Across all models we trained, real-world performance remained substantially below synthetic and curated performance, and neither additional data nor combining four datasets bridged the gap; model capacity was not evaluated under controlled circumstances. Recall highlights the gap: the four-dataset combined model identifies less than half of the vulnerable functions in the CodeXGLUE test set, which is not deployable regardless of headline accuracy.

The practical upshot is that results should appear on more than one benchmark, datasets should undergo an audit before anyone combines them, labels should count as

measurements carrying error rather than as ground truth, and recall deserves more weight than accuracy for a security task. Better labels, more context than a single function, and task formulations that account for uncertainty are likely to be just as important as bigger models trained on the same data. Two constraints indicate natural expansions. Even our relaxed matching misses semantically comparable code, so every count is a lower bound, and our human verification on which label is valid covers just a small sample. Our pipeline covers preparation, audit, and contradiction detection, with fixed seeds and per-function predictions recorded behind every number, so every result is reproducible from the recorded outputs. Benchmarks are how this field observes its own progress, and calibrating them deserves the same attention we give to the models they measure.

Author Contributions: Conceptualization, M.I.; methodology, M.I. and T.K.; software, M.I. and T.K.; validation M.I., T.K. and A.L.; formal analysis, M.I. and T.K.; investigation, M.I. and T.K.; resources, M.I. and T.K.; data curation, T.K.; writing—original draft preparation, T.K.; writing—review and editing, M.I., T.K. and A.L.; visualization, T.K.; supervision, M.I., T.K. and A.L.; project administration, A.L.; funding acquisition, A.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the European Union under the Horizon Europe programme, grant agreement No. 101244751 (project AISSAM—"Automated Vulnerability Detection in Software Development Using AI Techniques"; call HORIZON-WIDERA-2024-TALENTS-02; type of action: HORIZON TMA MSCA Postdoctoral Fellowships).

Data Availability Statement: The four benchmark datasets analyzed in this study are publicly available from their original sources [14–17]. The analysis scripts, training and evaluation notebooks, audit reports, fingerprint lists, per-function prediction files, and the manual check of contradictory functions generated in this study are openly available in Zenodo at https://doi.org/10.5281/zenodo.23236512.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A

Table A1 lists the 30 disagreeing functions examined in the manual check (Sections 5.2 and 6.4), sampled at random (seed 42) from the complete-dataset comparison, ten per pair of real-world datasets. For each function it gives the benchmark that labels it vulnerable, the commit behind that label, what the commit changes in the function and whether that change supports the vulnerable label. Items 4–6 come from the same commit, and items 1 and 26 are the same function appearing in two pairs.

Table A1. Manual check of 30 functions with contradictory labels.

No.	Pair	Function	Label 1 from	Commit (Label 1)	Change in the Function	Vulnerable Label
1	BV–CX	virtio_queue_host_notifier_read	CX	qemu 344dc16	Refactor: split for new aio handler	Not supported
2	BV–CX	decompress_i	CX	FFmpeg 5666b95	Bit mask stops out-of-bounds read	Supported
3	BV–CX	qcow2_expand_zero_clusters	CX	qemu de82815	Graceful failure on huge allocation	Ambiguous
4	BV–CX	vmstate_subsection_load	CX	qemu 60fe637	Pure file move, code unchanged	Not supported
5	BV–CX	put_buffer	CX	qemu 60fe637	Pure file move, code unchanged	Not supported
6	BV–CX	put_unused_buffer	CX	qemu 60fe637	Pure file move, code unchanged	Not supported

Table A1. *Cont.*

No.	Pair	Function	Label 1 from	Commit (Label 1)	Change in the Function	Vulnerable Label
7	BV–CX	audio_capture	CX	qemu e2b72cb	Output throttling against memory exhaustion	Supported
8	BV–CX	ff_thread_ref_frame	CX	FFmpeg 083300b	Feature change (per-field owners)	Not supported
9	BV–CX	vmxnet3_set_events	CX	qemu c508277	IOMMU-aware accessor (mechanical)	Ambiguous
10	BV–CX	mov_read_chpl	CX	FFmpeg 5c72065	Return check prevents uninitialized read	Supported
11	BV–DV	set_fflags_platform	DV	libarchive e2ad1a2	O_NOFOLLOW prevents symlink attack	Supported
12	BV–DV	jpc_dec_process_sot	DV	jasper fcbabda	abort() replaced by error return (CVE-2018-9154)	Supported
13	BV–DV	fuse_readpage	DV	linux 5d069db	Side effect of fix made elsewhere	Not supported
14	BV–DV	dtls1_process_out_of_seq_message	DV	openssl bff1ce4	Removes double free	Supported
15	BV–DV	xps_decode_font_char_imp	DV	ghostpdl 961b10c	Bounds checks on font offsets	Supported
16	BV–DV	drop_sync	DV	qemu 2563c9c	Fixes inverted size check (CVE-2017-2630)	Supported
17	BV–DV	complete_mmio	DV	linux f78146b	Functional fix (page-crossing MMIO)	Not supported
18	BV–DV	get_xref_linear_skipped	DV	pdfresurrect 7e35d18	Prevents infinite loop on crafted PDF	Supported
19	BV–DV	WriteTIFFImage	DV	ImageMagick 728dc6a	Avoids division by zero	Supported
20	BV–DV	dom_document_save_html_file	DV	php-src f7d7bef	Rejects paths containing NUL bytes	Supported
21	CX–DV	protocol_client_auth	CX	qemu 7364dbd	Debug prints replaced by tracing	Not supported
22	CX–DV	output_frame	DV	FFmpeg 8c55ff3	Fixes out-of-array read	Supported
23	CX–DV	mxf_read_content_storage	CX	FFmpeg 0834f20	Fixes memory leak	Supported
24	CX–DV	get_qcd	CX	FFmpeg 55fa898	Initializes memory before use	Supported
25	CX–DV	sysbus_ahci_class_init	CX	qemu e4f4fb1	Configuration flag only	Not supported
26	CX–DV	virtio_queue_host_notifier_read	CX	qemu 344dc16	Refactor: split for new aio handler	Not supported
27	CX–DV	cirrus_bitblt_fill_nop	DV	qemu 026aeff	Signature change of an empty function	Not supported
28	CX–DV	vdi_close	CX	qemu 17cce73	Side effect of allocator change	Not supported
29	CX–DV	vnc_job_new	CX	qemu fedf0d3	Style clean-up (g_new0)	Not supported
30	CX–DV	ahci_irq_raise	CX	qemu bb639f8	Functional crash fix	Not supported

BV = Big-Vul, CX = CodeXGLUE, DV = DiverseVul. Label 1 from: the benchmark that labels the function vulnerable. Vulnerable label: Supported if the change fixes a security weakness in the function, Not supported if it does not (absence of evidence for the vulnerable label, not proof that the function is safe), or Ambiguous; the decision rule is given in Section 5.2.

References

1. Baran, G. 40,000+ CVEs Published in 2024, Marking a 38% Increase from 2023. *Cyber Security News*, January 2025. Available online: https://cybersecuritynews.com/40000-cves-published-in-2024/ (accessed on 25 July 2026).
2. Ghaffarian, S.M.; Shahriari, H.R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Comput. Surv. (CSUR)* **2017**, *50*, 56. [CrossRef]
3. Li, Z.; Zou, D.; Xu, S.; Ou, X.; Jin, H.; Wang, S.; Deng, Z.; Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv* **2018**, arXiv:1801.01681. [CrossRef]
4. Risse, N.; Liu, J.; Böhme, M. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *Proc. ACM Softw. Eng.* **2025**, *2*, 388–410. [CrossRef]
5. Li, Z.; Zou, D.; Xu, S.; Jin, H.; Zhu, Y.; Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2244–2258. [CrossRef]
6. Russell, R.; Kim, L.; Hamilton, L.; Lazovich, T.; Harer, J.; Ozdemir, O.; Ellingwood, P.M.; McConley, M. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*; IEEE: Piscataway, NJ, USA, 2018; pp. 757–762.
7. Fu, M.; Tantithamthavorn, C. Linevul: A transformer-based line-level vulnerability prediction. In Proceedings of the 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 23–24 May 2022; pp. 608–620.
8. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Advances in Neural Information Processing Systems 32 (NeurIPS 2019), Proceedings of the 33rd International Conference on Neural Information Processing Systems, Vancouver, BC, Canada, 8–14 December 2019*; Curran Associates, Inc.: Red Hook, NY, USA, 2019; pp. 10197–10207.
9. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2020, Online, 16–20 November 2020; pp. 1536–1547.
10. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. Graphcodebert: Pre-training code representations with data flow. *arXiv* **2020**, arXiv:2009.08366.
11. Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; Yin, J. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2022; pp. 7212–7225.
12. Hanif, H.; Maffeis, S. Vulberta: Simplified source code pre-training for vulnerability detection. In *Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN)*; IEEE: Piscataway, NJ, USA, 2022; pp. 1–8.
13. Chakraborty, S.; Krishna, R.; Ding, Y.; Ray, B. Deep learning based vulnerability detection: Are we there yet? *IEEE Trans. Softw. Eng.* **2021**, *48*, 3280–3296. [CrossRef]
14. Boland, T.; Black, P.E. Juliet 1.1 C/C++ and java test suite. *Computer* **2012**, *45*, 88–90. [CrossRef]
15. Lu, S.; Guo, D.; Ren, S.; Huang, J.; Svyatkovskiy, A.; Blanco, A.; Clement, C.; Drain, D.; Jiang, D.; Tang, D.; et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv* **2021**, arXiv:2102.04664.
16. Fan, J.; Li, Y.; Wang, S.; Nguyen, T.N. AC/C++ code vulnerability dataset with code changes and CVE summaries. In Proceedings of the 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29–30 June 2020; pp. 508–512.
17. Chen, Y.; Ding, Z.; Alowain, L.; Chen, X.; Wagner, D. 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, 16–18 October 2023; pp. 654–668.
18. Croft, R.; Babar, M.A.; Kholoosi, M.M. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*; IEEE: Piscataway, NJ, USA, 2023; pp. 121–133.
19. Risse, N.; Böhme, M. Uncovering the limits of machine learning for automatic vulnerability detection. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 4247–4264.
20. Ding, Y.; Fu, Y.; Ibrahim, O.; Sitawarin, C.; Chen, X.; Alomair, B.; Wagner, D.; Ray, B.; Chen, Y. Vulnerability detection with code language models: How far are we? *arXiv* **2024**, arXiv:2403.18624.
21. Lin, G.; Wen, S.; Han, Q.L.; Zhang, J.; Xiang, Y. Software vulnerability detection using deep neural networks: A survey. *Proc. IEEE* **2020**, *108*, 1825–1848. [CrossRef]
22. Du, X.; Wen, M.; Zhu, J.; Xie, Z.; Ji, B.; Liu, H.; Shi, X.; Jin, H. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. In *Findings of the Association for Computational Linguistics: ACL 2024*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2024; pp. 10507–10521.
23. Bhandari, G.; Naseer, A.; Moonen, L. 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, Athens, Greece, 19–20 August 2021; pp. 30–39.

24. Nikitopoulos, G.; Dritsa, K.; Louridas, P.; Mitropoulos, D. 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, Athens, Greece, 23–28 August 2021; pp. 1565–1569.
25. Zheng, Y.; Pujar, S.; Lewis, B.; Buratti, L.; Epstein, E.; Yang, B.; Laredo, J.; Morari, A.; Su, Z. 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: Piscataway, NJ, USA, 2021; pp. 111–120.
26. Allamanis, M. The adverse effects of code duplication in machine learning models of code. In Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Athens, Greece, 23–24 October 2019; pp. 143–153.
27. Steenhoek, B.; Gao, H.; Le, W. Dataflow analysis-inspired deep learning for efficient vulnerability detection. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE), Lisbon, Portugal, 14–20 April 2024; pp. 166–178.
28. Wilson, E.B. Probable inference, the law of succession, and statistical inference. *J. Am. Stat. Assoc.* **1927**, *22*, 209–212. [CrossRef]
29. Arcuri, A.; Briand, L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test. Verif. Reliab.* **2014**, *24*, 219–250. [CrossRef]
30. McNemar, Q. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* **1947**, *12*, 153–157. [CrossRef] [PubMed]
31. Sullivan, G.M.; Feinn, R. Using effect size—Or why the P value is not enough. *J. Grad. Med. Educ.* **2012**, *4*, 279–282. [CrossRef] [PubMed]
32. Dror, R.; Baumer, G.; Shlomov, S.; Reichart, R. The hitchhiker's guide to testing statistical significance in natural language processing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2018; pp. 1383–1392.
33. Holm, S. A simple sequentially rejective multiple test procedure. *Scand. J. Stat.* **1979**, *6*, 65–70.
34. Herzig, K.; Zeller, A. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*; IEEE: Piscataway, NJ, USA, 2013; pp. 121–130.
35. Nie, X.; Li, N.; Wang, K.; Wang, S.; Luo, X.; Wang, H. Understanding and tackling label errors in deep learning-based vulnerability detection (experience paper). In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, USA, 17–21 July 2023; pp. 52–63.
36. Jimenez, M.; Rwemalika, R.; Papadakis, M.; Sarro, F.; Le Traon, Y.; Harman, M. 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, Tallinn, Estonia, 26–30 August 2019; pp. 695–705.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.