



Malware-Domain Continued Pre-Training for Binary Malware Classification
A Leakage-Aware Study of Code Models on the SBAN Corpus

Costin Teodorescu

Supervisor(s): Dr. Soham Chakraborty, Dr. Przemysław Pawelczak

EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 21, 2026

Name of the student: Costin Teodorescu

Final project course: CSE3000 Research Project

Thesis committee: Dr. Soham Chakraborty, Dr. Przemysław Pawelczak, Prof. Dr. A. van Deursen

An electronic version of this thesis is available at <https://repository.tudelft.nl/>

Abstract

Continued pre-training can adapt language models to a domain, but for malware classification it is unclear whether gains come from malware-specific information or from additional training on code. We study this question on a leakage-controlled binary benchmark derived from the SBAN corpus, using strict BENIGN/MALWARE labels, exact duplicate removal, and matched comparisons between TF-IDF baselines, CodeBERT, and Qwen2.5-Coder variants. Across the tested model sizes and pre-training data budgets, continued pre-training changes model behaviour but does not produce a reliable downstream classification improvement. Even in the best malware-related setting, the margin over an equally trained general-code control is very small. Under these constraints, the overall trend is that malware-related continued pre-training does not improve binary classification in a reliable way.

1 Introduction

Malware classification is difficult because malicious and benign programs can share syntax, libraries, and implementation patterns, while malware authors deliberately modify code to avoid fixed signatures. Automated classifiers are useful for large-scale triage and analyst support, but their results depend strongly on the data representation, label policy, and split construction. EMBER supports static PE malware detection from extracted binary features [1], BODMAS adds temporal PE malware evaluation concerns [2], the Microsoft Malware Classification Challenge focuses on malware-family classification from byte and disassembly representations [3], and SBAN provides a multi-view software-security corpus [4]. These datasets show that malware classification is not only a modelling problem; it is also a data-construction problem.

Code-oriented language models are natural candidates for source-code and program-text inputs, but their cost makes the practical question more specific than whether transformers can be used for malware detection. A useful malware pipeline should first be compared with low-cost baselines, then with pretrained code models, and only then with additional continued pre-training. Malware data can also be represented in several forms, including source code, assembly, binary-derived text, natural-language descriptions, API-call sequences, and dynamic traces. These representations expose different evidence. This work focuses on source code and natural-language descriptions because they are reproducible text views available in the constructed SBAN-derived benchmark.

Continued pre-training is a plausible way to adapt a code model to malware-related text, but a gain after continued training is not automatically evidence of malware-specific learning. Gururangan et al. [5] show that continued pre-training on domain- or task-relevant text can improve later task performance, while Ibrahim et al. [6] emphasise that

continual pre-training needs controls for forgetting and extra training exposure. In this setting, an apparent improvement may come from malware-domain information, but it may also come from additional optimisation, more source-code tokens, duplicate leakage, or biased splits.

Prior work has already applied BERT-style models, transformers, LLMs, and self-supervised learning to malware and security tasks [7; 8; 9; 10; 11]. The question addressed here is therefore narrower than whether transformers can be used for malware detection. The main research question is:

Can continued pre-training on malware-related code improve code language models for binary malware classification?

To make this question testable, we divide it into five sub-questions:

- RQ1.** How well do low-cost baselines perform before expensive model training?
- RQ2.** Which text view gives stronger malware-classification signal: source code or natural-language descriptions?
- RQ3.** Does continued pre-training improve supervised malware classification over the corresponding unadapted model?
- RQ4.** Does malware-related training data add value beyond extra general-code training?
- RQ5.** Does malware-related continued pre-training produce consistent improvements across model families and direct zero-shot scoring?

We answer these questions by building an explicit dataset manifest, applying a strict binary label policy, removing exact cross-split duplicates, and evaluating classical TF-IDF baselines, CodeBERT variants [12], and Qwen2.5-Coder variants [13]. The experiments compare general-code and malware-related continued-pretraining variants under matched downstream evaluation, while also checking direct zero-shot scoring. Under the tested model sizes, pre-training data budgets, and split controls, the observed trend is negative for the main hypothesis: malware-related continued pre-training does not lead to a reliable improvement in downstream classification. The contribution of this paper is therefore a leakage-aware benchmark and a controlled measurement of the limits of malware-domain continued pre-training for this task.

2 Background and Related Work

Malware classification research depends on two linked choices: how programs are represented, and how labels and evaluation splits are constructed. A classifier trained on source code, byte sequences, API calls, or natural-language descriptions is not necessarily learning from the same evidence, and benchmark scores can be misleading when duplicate samples or ambiguous labels cross evaluation boundaries. For this reason, prior work is most relevant to this study when it clarifies either the data representation, the pre-training method, or the evaluation assumptions behind the reported malware classifier.

2.1 Malware Benchmarks and Program Representations

Public malware benchmarks have made learning-based malware classification more reproducible, but they differ substantially in labels, data formats, and intended use. EMBER provides a benchmark for static PE malware detection from extracted binary features [1]. BODMAS focuses on PE malware analysis with temporal evaluation concerns [2]. The Microsoft Malware Classification Challenge is a malware-family benchmark built around byte and disassembly representations [3]. SBAN is closer to the present setting because it provides multiple software-security views, including source code, binary, assembly, and natural-language descriptions [4]. These differences require explicit label and split definitions, since dataset origin alone does not reliably define either malware status or evaluation membership.

Malware can be represented through several complementary forms. Source code and natural-language descriptions are directly compatible with code-oriented language models. Assembly and binary-derived representations preserve lower-level implementation details that may be lost in source-like text. API-call sequences and dynamic traces describe observed behaviour, but they require execution or sandboxed tracing and therefore answer a different experimental question [9; 10]. The present study focuses on source code and natural-language descriptions because these are reproducible text views available under the constructed benchmark. Assembly, binary, API-call, and dynamic-trace experiments remain relevant future comparisons, but they are not completed experiments in this paper.

2.2 Code Models and Continued Pre-Training

Pretrained code models provide the modelling basis for using language-model methods on program text. CodeBERT learns joint representations of programming language and natural language [12]. GraphCodeBERT extends this direction by incorporating data-flow information [14]. CodeT5 uses an encoder-decoder formulation for code understanding and generation [15]. Qwen2.5-Coder is a newer code-oriented LLM family used in the reported continued-pretraining comparisons [13]. These models support the choice of code-model baselines, but they do not by themselves establish that malware-specific adaptation improves malware classification.

Domain-adaptive pre-training and continued pre-training provide the methodological basis for the comparison. Prior work shows that continuing language-model pre-training on domain- or task-relevant text can improve later task performance in some settings [5]. Work on continual pre-training also motivates controls such as replay, since additional training can change model behaviour for reasons unrelated to the target domain alone [6]. In this study, that distinction directly shapes the experimental design: malware-related continued-pretraining variants are compared against unadapted models and matched general-code controls before any malware-specific benefit is claimed.

2.3 Malware-Specific Transformer and Self-Supervised Methods

Prior malware work has already used transformer and BERT-style models. Xu et al. propose Malbert as a pre-training method for malware detection [7], while Rahali and Akhloufi use the MalBERT name for Android malware detection with bidirectional transformer representations [8]. Oak et al. study sequence modelling for highly imbalanced malware detection [9], and Demirkiran et al. use pretrained transformer ensembles for imbalanced malware-family classification [10]. More recent work also uses LLMs as part of malware-detection pipelines, such as AppPoet’s multi-view prompt-engineering approach for Android malware detection [16]. These studies establish transformer-based malware analysis as prior work. The present paper therefore focuses on a narrower question: whether malware-domain continued pre-training improves binary malware classification under controlled data and training conditions.

Self-supervised malware representation learning is another relevant line of work. MalSSL studies label-efficient malware classification using self-supervised image representations [11]. Graph-based malware work instead uses control-flow or program-structure representations for detection [17]. These approaches reduce dependence on manual labels and exploit structure beyond flat text. However, they do not answer the narrower question studied here: whether continued pre-training on malware-related program text improves matched binary malware classification.

2.4 Binary and Assembly Representation Learning

Binary and assembly representation learning form a closely related but distinct research area. PalmTree learns assembly instruction embeddings [18], and jTrans uses a jump-aware transformer for binary-code similarity [19]. BERT-style binary-code models such as DeepSemantic and BinBert adapt transformer pre-training to binary analysis tasks [20; 21; 22]. AAPFE is especially relevant to malware analysis because it studies aligned assembly pre-training for function embeddings [23]. This work supports the broader view that low-level program representations can benefit from language-model-style training, but the current study is deliberately narrower: it first tests text-based continued pre-training under a leakage-aware binary classification protocol before extending the benchmark to assembly or binary comparisons.

2.5 Positioning of This Work

Prior work already applies pre-training, transformers, LLMs, self-supervised learning, and binary representation learning to malware detection. This paper contributes a controlled study of whether malware-related continued pre-training improves binary malware classification under matched evaluation conditions. The central question is tested against low-cost baselines, general-code continued pre-training, multiple model families, and direct zero-shot scoring. This framing is conservative, but necessary: without matched controls, an apparent gain could come from extra optimisation, extra source-code exposure, duplicate leakage, or biased data construction rather than malware-domain knowledge.

3 Methodology

The methodology treats the study as a controlled comparison rather than a single-model demonstration. We first fix the label policy, split construction, and duplicate-leakage controls. We then compare source and NLD views, low-cost baselines, unadapted code models, and continued-pretraining variants. This order keeps any claimed effect of malware-domain pre-training separate from data choices, extra code exposure, and threshold selection.

3.1 Research Question Mapping

The five subquestions are tied to the following comparisons:

- RQ1.** TF-IDF classifiers and non-adapted code-model baselines set the reference point before continued pre-training is interpreted.
- RQ2.** Source code and natural-language descriptions are compared on matched sample IDs.
- RQ3.** Unadapted models are compared with continued-pretraining variants under the same downstream classifier protocol.
- RQ4.** Malware-related continued pre-training is compared with general-code continued pre-training to separate domain exposure from extra code training.
- RQ5.** Cross-model supervised checks and direct zero-shot diagnostics test whether any effect is consistent beyond one model family or evaluation mode.

Language-model performance is affected by scale choices such as model size, data size, and training compute [24]. This study therefore treats malware-related continued pre-training as useful for classification only if it improves over matched baselines under the same split, model family, downstream classifier, training budget, threshold-selection rule, and held-out test set. The comparison design separates three questions: whether adaptation helps over an unadapted model, whether malware-related data helps beyond extra general-code training, and whether the same pattern appears beyond one model family or evaluation mode.

The continued-pretraining experiments are organised around two comparison settings. The main matched setting uses 300M-token corpora for Qwen2.5-Coder-0.5B and CodeBERT, while the scale check uses Qwen2.5-Coder-1.5B under a smaller continued-pretraining budget. Figure 1 summarises this comparison logic; the exact run settings are given in the experimental setup.

3.2 Metadata and Label Policy

All dataset handling is based on an explicit metadata index rather than directory names. The raw SBAN-style corpus contains several datasets and program representations, including source code, assembly, binary-derived text, and natural-language descriptions [4]. Because schemas and identifiers differ across files, each sample is tracked by dataset name, sample identifier, available representations, normalised label, and eligibility flags. This prevents the experiment from assuming, for example, that every file in a malware-oriented dataset directory is necessarily malicious.

The classification task is binary malware detection. Only samples with explicit BENIGN or MALWARE labels are included. Samples labelled PARTIALLY_MALICIOUS, UNKNOWN, or missing a binary label are excluded from the classification benchmark. This policy is deliberately strict: partially malicious samples are not merged into the malware class, and unknown samples are not treated as benign negatives. The resulting task is narrower, but avoids introducing label decisions that would be difficult to defend.

3.3 Split and Leakage Policy

The benchmark split is deterministic and stratified by dataset and label. Samples are assigned to train, validation, and test partitions using a fixed seed, with intended proportions of approximately 70%, 10%, and 20%. The validation split is used for model selection and threshold selection; the test split is held out for final reporting.

Duplicate leakage is controlled separately for source code and natural-language descriptions (NLD). Text is standardised by normalising line endings and stripping outer whitespace before hashing. If the same standardised text appears across train, validation, or test partitions, the full cross-partition duplicate group is removed from the corresponding no-leak split. This process removed 4,123 source-code documents across 644 duplicate groups and 24,142 NLD documents across 6,177 duplicate groups. These counts show that exact duplicate leakage is substantially stronger for NLD than for source text. The procedure removes exact duplicates only; near-duplicate and semantic leakage remain threats to validity.

For direct source-versus-NLD comparisons, the study uses a paired source-NLD benchmark. A sample is retained only if it appears in both representations and has the same split assignment and binary label in both input splits. Larger source-only and NLD-only no-leak benchmarks are still useful for modality-specific experiments, but direct modality comparisons use the paired benchmark so that both models are evaluated on the same sample IDs.

3.4 Baseline Systems

The baseline stage includes low-cost text models and non-adapted code-model baselines. Classical baselines use word and character TF-IDF features with logistic regression because they are cheap, reproducible, and strong enough to show whether the benchmark contains useful textual signal. The non-adapted CodeBERT and Qwen2.5-Coder baselines test how pretrained code models behave before malware-related continued pre-training [12; 13]. This keeps each continued-pretraining result tied to the corresponding unadapted reference point.

3.5 Continued-Pretraining Variants

Continued pre-training is tested on source-code text with model-appropriate objectives: causal language modelling for Qwen2.5-Coder and masked language modelling for CodeBERT. The variants separate no adaptation, extra general-code training, mixed malware-domain training, and malware-only training where the corresponding artefacts are available. The general-code variants are essential controls because they

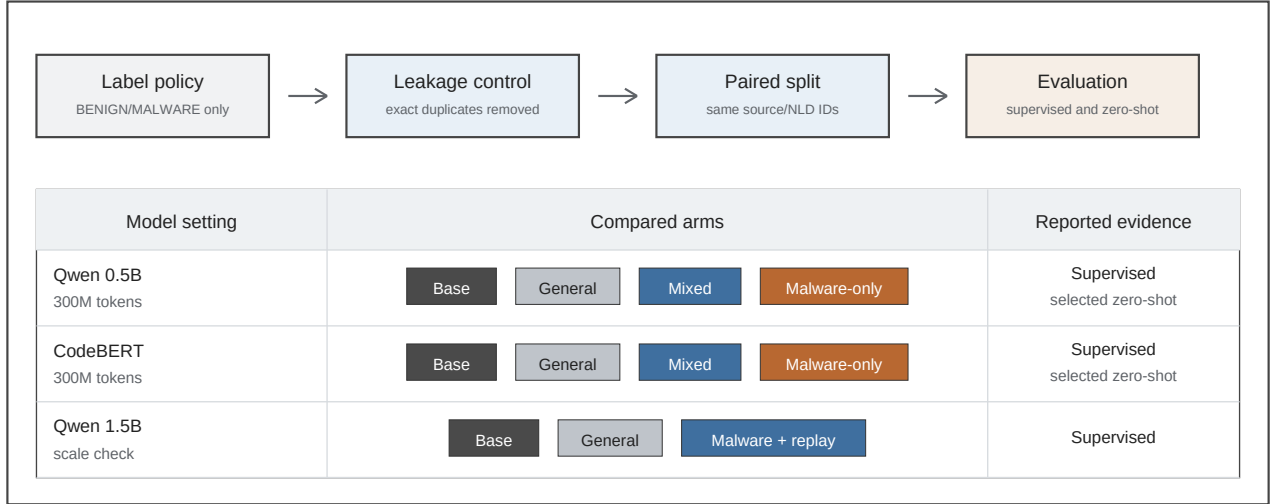


Figure 1: Reported comparison design. CPT means continued pre-training and NLD means natural-language description. The 300M-token setting is the main matched comparison; Qwen2.5-Coder 1.5B is the limited-budget scale check.

test whether any change comes from malware-related content or simply from more source-code training.

The malware-domain variants test whether adding malware-related source changes downstream classification beyond that control, while malware-only variants test whether removing general replay and benign context helps or shifts the model toward malware-heavy behaviour. Some malware-related pretraining material, especially xLang-derived source, is treated as domain-related text rather than as trusted downstream classification labels.

All continued-pretraining corpora exclude validation and test sample IDs from the paired downstream benchmark, with exclusion ledgers stored for audit. After adaptation, supervised comparisons use the same paired source-code benchmark, balanced class weighting, validation-only threshold selection, and held-out test protocol within each model family.

3.6 Evaluation and Comparison Protocol

All classifiers are evaluated with F1 score, AUROC, precision, and recall. Thresholds are selected on the validation split and then applied once to the held-out test split. This prevents the test set from influencing either model selection or decision-threshold tuning.

The comparison design is interpreted conservatively. Source-only and NLD-only baselines estimate representation signal, paired source-NLD runs compare views on the same samples, general-code continued pre-training controls for extra code exposure, and malware-related variants test added domain exposure. A malware-related variant is treated as useful only if it improves AUROC or F1 over matched controls, not merely because it changes precision and recall.

Direct zero-shot scoring is included as a diagnostic check rather than as the main malware classifier. Because a zero-shot system can obtain high recall by predicting malware for most samples, zero-shot results are interpreted together with predicted malware rate and not as standalone deployment ev-

idence.

4 Experimental Setup

The reported setup is tied to a public reproducibility package rather than to informal local scripts.¹ The source repository contains the scripts, configurations, saved metrics, predictions, logs, figures, run summaries, and documentation used to inspect the reported results. The large-artefact repository² stores raw and materialised corpora, split databases, duplicate-cluster artefacts, and selected model artefacts so that the source repository remains cloneable.

4.1 Datasets and Split Artefacts

All reported experiments use the manifest-backed binary malware-classification benchmark described in Section 3. The downstream benchmark uses BODMAS, Dike, MalwareBazzar, and Sorel20m under the strict BENIGN/MALWARE label policy.

The saved supervised and zero-shot scores use the result-protocol split layer. This layer contains source-only, NLD-only, and paired source-NLD splits. The paired split is the main split for direct comparisons and contains 118,450 training samples, 15,608 validation samples, and 31,461 test samples. A stricter clustered split layer is included in the reproducibility package for audit and future reruns, but the reported model scores are not evaluated on that newer split. Metrics should therefore not be compared across split protocols unless the corresponding models are rerun under the same protocol.

The downstream Qwen and CodeBERT experiments use the source-code view of the paired benchmark. Classical

¹Source repository: <https://github.com/TeodorescuCostin/malware-llm-pretraining-classification>.

²Large data, split, corpus, and selected checkpoint artefacts: <https://huggingface.co/datasets/TeodorescuCostin/malware-llm-pretraining-classification-artifacts>.

baselines are run on both source code and natural-language descriptions so that the two textual representations can be compared under the same sample IDs. Assembly, binary, API-call, and dynamic-trace representations are outside the reported setup.

4.2 Classical Baselines

The classical baselines use hashed TF-IDF features with sparse logistic regression. The word model uses hashed word unigram and bigram features, while the character model uses hashed character 3–5 gram features. Both use a hashing dimensionality of 2^{18} , balanced class weighting, and deterministic corpus materialisation from the manifest-backed split. The implementation uses an online sparse logistic-regression learner with AdaGrad updates, rather than relying on a separate machine-learning library.

The classical baselines are run separately on source-code text and NLD text, using the same paired sample IDs for the direct representation comparison.

4.3 CodeBERT Experiments

The CodeBERT experiments use microsoft/codebert-base [12]. The full fine-tuning baseline trains on the paired source-code corpus with a maximum sequence length of 256, balanced class weighting, validation-F1 threshold selection, and the same train, validation, and test partitions as the other downstream source-code models.

The frozen baseline keeps the pretrained encoder fixed, extracts mean-pooled CodeBERT embeddings from the source-code corpus, and trains a lightweight linear classifier on top of the saved embeddings. This tests whether pretrained code representations are useful without updating the transformer encoder.

The 300M-token CodeBERT continued-pretraining runs use masked language modelling with 15% token masking and a maximum sequence length of 512. The compared variants are the unadapted model, general-code continued pre-training, mixed malware-domain continued pre-training, and malware-only continued pre-training. After continued pre-training, each variant is evaluated with the same supervised fine-tuning protocol on the paired source-code benchmark.

4.4 Qwen Experiments

The 300M-token Qwen comparison uses Qwen/Qwen2.5-Coder-0.5B with unadapted, general-code continued-pretraining, mixed malware-domain continued-pretraining, and malware-only continued-pretraining variants. Continued pre-training uses causal language modelling with a maximum sequence length of 2048 and LoRA-style adapter training with rank 16, alpha 32, and dropout 0.05 [25]. This is the cleaner scale-matched Qwen setup used for the main continued-pretraining interpretation.

The limited-budget Qwen scale check uses Qwen/Qwen2.5-Coder-1.5B as the base code LLM [13]. It compares the unadapted model, a general-code continued-pretraining control, and a malware-source-with-general-code-replay variant. This comparison is not treated as the final answer because the continued-pretraining token budget was limited relative to the model size, but it is useful

for checking whether the same trend appears for a larger model.

Additional Qwen 0.5B xLang-matched runs are retained as supporting artefacts. They are useful for understanding weak malware-related source material, but xLang-derived text is treated as malware-related domain text rather than as trusted downstream classification labels.

4.5 Downstream Classification and Zero-Shot Scoring

All supervised neural variants are evaluated on the paired source-code benchmark with the same split and validation-only threshold-selection policy within each model family. Qwen classifiers use a maximum sequence length of 1024, one downstream epoch, balanced class weighting, and validation-F1 threshold selection over a fixed threshold grid. CodeBERT fine-tuning uses a maximum sequence length of 256, balanced class weighting, and the same threshold-selection rule. The selected threshold is applied once to the held-out test split.

Direct zero-shot scoring is evaluated separately for selected Qwen and CodeBERT arms. It compares each model’s direct preference for benign or malware labels without supervised classifier training. These results are reported as diagnostics because prediction-rate shifts can make F1 look better even when the model mostly predicts one class.

4.6 Compute, Software, and Artefact Policy

Experiments were run across DelftBlue-oriented Slurm workflows and JarvisLabs H100 runs. The DelftBlue runs used the DelftBlue supercomputer provided by Delft High Performance Computing Centre [26]. The baselines and limited-budget Qwen 1.5B work used the DelftBlue setup, while the 300M-token Qwen 0.5B and CodeBERT runs used JarvisLabs resources where more GPU capacity was available. The reproducibility package records the exact scripts, configurations, logs, metrics, predictions, and run summaries used for the reported results.

Saved metrics and predictions are the canonical evidence for the reported numbers. Some selected checkpoints and embeddings can be restored from the large-artefact repository, but not every full merged checkpoint or adapter directory is included because of storage size. This limits exact model-level reruns for some arms, but not inspection of the reported metrics, predictions, run summaries, and configurations.

5 Results

All reported supervised metrics use the result-protocol paired benchmark and validation-selected thresholds described in the experimental setup.

Figure 2 gives a compact overview of the supervised AUROC and F1 results that are reported in detail in the later result tables.

5.1 Benchmark and Low-Cost Baselines

The paired source-NLD benchmark contains 165,519 samples after label filtering, exact duplicate removal, and intersection of the source and NLD no-leak splits. The label distribution is imbalanced, with 79.5% BENIGN samples and

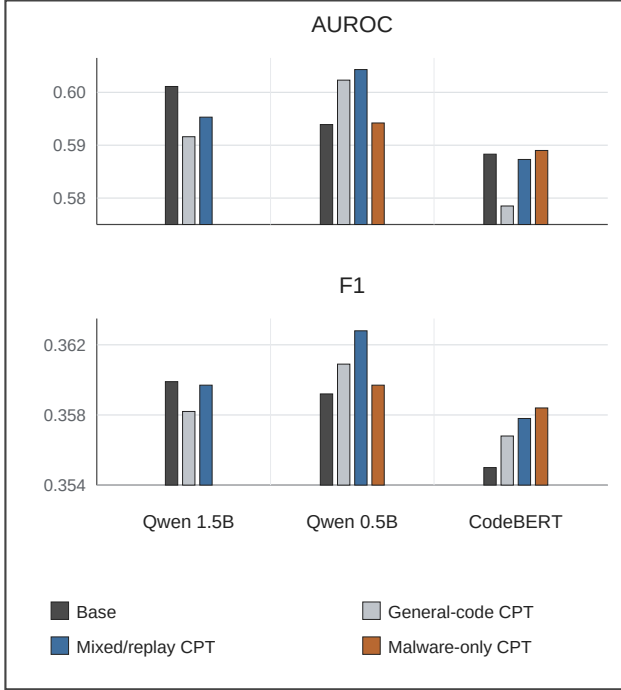


Figure 2: Supervised AUROC and F1 overview.

Table 1: Dataset composition of the paired source-code and NLD benchmark.

Dataset	Train	Val.	Test	Total
BODMAS	53,671	7,058	14,213	74,942
Dike	9,540	1,270	2,555	13,365
MalwareBazzar	7,849	1,006	2,037	10,892
Sorel20m	47,390	6,274	12,656	66,320
Total	118,450	15,608	31,461	165,519

20.5% MALWARE samples. Table 1 gives the dataset composition.

The paired split construction found no source/NLD label or split mismatches among the retained samples. This matters for RQ2 because source and NLD baselines are evaluated on the same sample IDs. The separate source-only and NLD-only no-leak splits remain useful for representation-specific experiments, but they are not used for direct source-versus-NLD comparison in this paper.

Table 2 reports the low-cost baselines and early CodeBERT baselines. The source character TF-IDF model is the strongest classical baseline, while the best NLD baseline is slightly weaker. Frozen CodeBERT embeddings with a linear classifier are the strongest early baseline. The full CodeBERT fine-tuning baseline selects a threshold of 0.0 and predicts every test sample as malware, so it is treated as unstable rather than as a useful classifier.

These results show that the benchmark retains signal after leakage controls and that low-cost baselines are necessary comparison points. Source-code text is stronger than NLD under the current TF-IDF setup.

Table 2: Baseline test results on the paired benchmark. NLD means natural-language descriptions; the CodeBERT rows use source code only.

Family	Model	AUROC	F1	Prec.	Rec.
Classical	Source char TF-IDF	0.5772	0.3325	0.2513	0.4911
Classical	NLD word TF-IDF	0.5589	0.3200	0.2369	0.4929
CodeBERT	Frozen + linear	0.5832	0.3536	0.2224	0.8616
CodeBERT	Full fine-tune	0.4784	0.3432	0.2071	1.0000

Table 3: Limited-budget Qwen2.5-Coder-1.5B supervised scale check.

Arm	AUROC	F1	Prec.	Rec.
Base	0.6011	0.3599	0.2482	0.6541
General-code CPT	0.5916	0.3582	0.2336	0.7676
Malware replay CPT	0.5953	0.3597	0.2339	0.7775

5.2 Limited-Budget Qwen 1.5B Scale Check

The Qwen2.5-Coder-1.5B scale check uses a larger model with a smaller continued-pretraining budget than the 300M-token setting. Table 3 shows that the unadapted model has the strongest AUROC and F1 among the three Qwen 1.5B supervised arms. Both continued-pretraining variants increase recall compared with the base model, but this comes with lower precision and lower AUROC.

The scale check therefore does not show a supervised gain from continued pre-training. Because its continued-pretraining budget was limited relative to the 1.5B-parameter model size, the 300M-token Qwen 0.5B setup remains the clearer Qwen comparison.

5.3 300M-Token Supervised Comparisons

Table 4 reports the main 300M-token supervised comparisons. For Qwen2.5-Coder-0.5B, the mixed malware-domain arm is the best Qwen arm, but the gain over the matched general-code arm is only 0.0020 AUROC and 0.0018 F1. The malware-only Qwen arm is close to the base model.

The CodeBERT 300M results provide the cross-model check. CodeBERT does not reproduce a clear mixed malware-domain AUROC gain. Its malware-only arm has the highest CodeBERT F1, but the AUROC differences between CodeBERT base, mixed, and malware-only are very small.

In the xLang-matched Qwen 0.5B supporting runs, the matched general-code run slightly outperforms the matched malware-related run. Since the weak xLang-derived source is malware-related but not a trusted downstream label source, these runs are supporting evidence rather than the central result.

5.4 Direct Zero-Shot Diagnostics

Direct zero-shot scoring checks whether continued pre-training changes the model’s direct preference for benign or malware labels. Table 5 shows that malware-only continued pre-training increases zero-shot F1 mainly by pushing predictions strongly toward the malware class. Qwen 0.5B malware-only predicts 98.9% of test samples as malware, and CodeBERT malware-only predicts 100.0% as malware.

These zero-shot results show that continued pre-training can change label preference, but they do not show a practical

Table 4: Supervised 300M-token CPT results on the paired source-code benchmark. CPT means continued pre-training.

Arm	AUROC	F1	Prec.	Rec.	Pred.
<i>Qwen 0.5B</i>					
Base	0.5939	0.3592	0.2300	0.8191	0.7375
General-code CPT	0.6023	0.3609	0.2521	0.6354	0.5220
Mixed-domain CPT	0.6043	0.3628	0.2480	0.6757	0.5644
Malware-only CPT	0.5942	0.3597	0.2303	0.8207	0.7382
<i>CodeBERT</i>					
Base	0.5883	0.3550	0.2247	0.8448	0.7786
General-code CPT	0.5785	0.3568	0.2260	0.8473	0.7764
Mixed-domain CPT	0.5873	0.3578	0.2317	0.7853	0.7021
Malware-only CPT	0.5890	0.3584	0.2326	0.7804	0.6948

Table 5: Direct zero-shot diagnostics. Pred. malware is the test-set malware prediction rate without supervised classifier training.

Family	Arm	AUROC	F1	Pred. malware
Qwen 0.5B	Base	0.4816	0.2438	0.3116
Qwen 0.5B	Malware-only 300M CPT	0.5210	0.3432	0.9887
CodeBERT	Base	0.4678	0.0737	0.0525
CodeBERT	Malware-only 300M CPT	0.5408	0.3432	1.0000

zero-shot detector. The supervised results remain the main evidence for classification performance.

5.5 Overall Result

Taken together, the result tables do not show a clear classification benefit from malware-related continued pre-training over matched controls. The only clear Qwen 0.5B gain over the base model remains in a modest performance range and is very close to the general-code continued-pretraining control. The Qwen 1.5B scale check, CodeBERT runs, and zero-shot diagnostics do not support a consistent improvement.

6 Discussion

The results point in a clear direction for the tested settings. Continued pre-training changes model behaviour, but it does not translate into a reliable malware-classification improvement. The current evidence therefore supports a negative trend for the main hypothesis under the tested conditions.

6.1 Baseline Signal and Input View

The low-cost baselines are not only sanity checks. They define the minimum level that expensive model training has to beat. The source-code character TF-IDF model and the frozen CodeBERT classifier both reach modest but meaningful performance, while the full CodeBERT fine-tuning run collapses to an all-malware operating point. This shows that the benchmark contains learnable signal after filtering, but also that larger or more trainable models are not automatically better under a limited downstream setup.

The source-versus-NLD comparison should also be read narrowly. Source code performs better than NLD in the paired TF-IDF setup, but this does not mean that source code is always the better malware representation. The conclusion is only that, for this benchmark and this shallow feature pipeline, source text carries more useful signal than the available natural-language-description field. A different NLD construction, a better description model, or a task focused on behavioural explanations could lead to a different outcome.

6.2 What Continued Pre-Training Added

The limited-budget Qwen 1.5B scale check is useful because it separates a successful continued-pretraining run from a successful downstream classifier. The continued-pretraining variants increase recall, but they reduce precision and do not improve AUROC or F1 over the base model. Since AUROC is independent of the selected threshold, this suggests that the adapted model does not rank benign and malware samples more effectively in that setting.

The 300M-token Qwen 0.5B experiment is the only setting that could be read as positive for continued pre-training, but it still supports a cautious conclusion. The mixed malware-domain arm is better than the Qwen base model, yet it is only 0.0020 AUROC and 0.0018 F1 above the matched general-code arm. That makes the result too small to treat as clear malware-specific learning. The malware-only Qwen arm is close to the base model, which also weakens the idea that more malware-related text alone is enough.

The CodeBERT 300M-token results reinforce this caution. They do not reproduce the same mixed-domain improvement seen for Qwen 0.5B. The best CodeBERT differences are small, and the ordering of the variants changes depending on the metric. This matters because a useful research conclusion should not depend only on one model family and one favourable comparison.

6.3 Operating Points

The supervised models often trade precision for recall when thresholds are selected on validation F1. In malware triage, high recall can be useful because missed malware is costly. However, a classifier that raises too many false positives can also become impractical for analysts. For that reason, AUROC, F1, precision, recall, and the predicted-malware rate need to be interpreted together.

The zero-shot results are included only as a diagnostic example of this issue. Malware-only continued pre-training increases zero-shot F1 mostly by pushing the model toward predicting nearly every sample as malware. This label-preference shift leaves the supervised results as the main evidence for classification performance.

6.4 Threats to Validity

The largest internal-validity risk is leakage. The benchmark removes exact cross-split duplicates, and the paired split avoids comparing source and NLD on different sample IDs. Still, near-duplicates, related malware variants, reused templates, or semantically equivalent generated descriptions may remain. The results should therefore be described as leakage-aware, not leakage-free.

The second limitation is that the reported model results use the result-protocol split layer, while a stricter clustered split is available in the reproducibility package for future reruns. This does not invalidate the current comparisons, because the compared models use the same reported split, but it means the numbers should not be mixed with future clustered-split results unless all models are rerun under the same protocol.

Label quality is another concern. The benchmark uses only explicit BENIGN and MALWARE labels and excludes

PARTIALLY_MALICIOUS, UNKNOWN, and missing labels. This avoids making an implicit label-policy decision, but it also removes ambiguous cases that would appear in real security work. The binary task controls the label policy but does not model operational malware detection fully.

The experiments are also limited by compute budget and repetition. The main comparisons are one-run measurements without confidence intervals across random seeds. Some full checkpoints are not preserved because of storage size, although metrics, predictions, configurations, logs, and selected artefacts are preserved. Small margins, especially the Qwen 0.5B mixed-versus-general gap, should therefore not be used to overturn the broader negative trend.

6.5 Relation to Prior Work

Prior malware research has already shown that BERT-style models, transformers, LLM-assisted pipelines, and self-supervised learning can be useful for security tasks [7; 8; 9; 10; 11]. This paper asks a narrower question: whether continuing the pre-training of code models on malware-related text improves binary malware classification.

That distinction is important because domain-adaptive and continual pre-training can help, but they can also reflect extra training exposure, forgetting, duplicated data, or task mismatch [5; 6]. The current results fit that warning: continued pre-training is technically feasible and sometimes changes downstream behaviour, but the tested malware-related variants do not provide a clear classification benefit over the controls.

7 Responsible Research and Ethical Considerations

This study uses malware-related data, so the work is limited to static text representations, metadata, manifests, and offline learning pipelines. No malware binary is executed, no recovered source-code sample is compiled or run, and the paper does not publish operational malware snippets. The goal is to measure classification methods under controlled conditions, not to build or improve offensive tooling.

7.1 Data Handling and Leakage Controls

The raw corpus is treated as read-only. Derived artefacts such as metadata indexes, splits, filtered corpora, configurations, logs, predictions, metrics, and selected checkpoints are stored separately from the raw data. The benchmark is built from record-level metadata rather than folder names, because the corpus contains different schemas, partially aligned representations, and mixed label spaces.

The strict binary label policy is also an ethical and scientific choice. Samples labelled PARTIALLY_MALICIOUS, UNKNOWN, or without a binary label are excluded instead of being forced into either class. Exact cross-split duplicates are removed for the reported source and NLD split layers, and direct source-versus-NLD comparisons use the same sample IDs. These controls reduce inflated performance estimates and make the reported comparisons more transparent.

7.2 Scope and Bias

The benchmark is built from public malware and software-security datasets, which reflect their own collection methods, time periods, label sources, and representation choices [1; 2; 3; 4]. The results should therefore not be interpreted as general malware-detection performance in the wild. The models are evaluated on a constructed binary classification task, not on live endpoint detection, adversarial robustness, family attribution, or dynamic behaviour analysis.

The paired source-NLD benchmark also narrows the scope because it keeps only samples available in both representations. This makes the comparison fair, but it does not represent all available data. The reported results are most useful as a controlled study of whether malware-related continued pre-training helps under fixed constraints, not as a deployment-ready detector evaluation.

7.3 Claim Boundaries and Reproducibility

Continued pre-training can change a model because of malware-domain exposure, but also because of extra optimisation, extra source-code tokens, duplicated information, or a changed label preference. For this reason, the paper treats successful continued-pretraining runs and language-modelling improvements as intermediate evidence, not as proof of better malware classification. The main claims are based on downstream test metrics under matched evaluation.

The reproducibility package records the source code, configurations, split artefacts, logs, metrics, predictions, run summaries, and selected large artefacts needed to inspect the reported results. Some full checkpoints are not preserved because of storage size, so exact model restoration is not equally complete for every run. This limitation is reported explicitly. It does not change the saved metrics and predictions, but it does limit how easily some runs can be reproduced bit-for-bit.

7.4 Use of AI Tools

AI tools were used as support tools during this project. We used them for code-debugging support, repository organisation, and spelling, grammar, and consistency checks on text that had already been written. We did not use them as a source of experimental results or conclusions. The reported metrics, tables, and claims are based on the project artefacts and the conducted runs.

8 Conclusion and Future Work

This paper studied whether continued pre-training on malware-related code improves code language models for binary malware classification. Under the tested constraints, the answer is negative for the main hypothesis: continued pre-training can change model behaviour, but those changes do not become reliable classification gains. The strongest supervised Qwen 0.5B score comes from the mixed malware-domain variant, yet its margin over the matched general-code control is very small and remains in a modest performance range. The limited-budget Qwen 1.5B scale check does not improve over the base model, CodeBERT does not

show a consistent malware-domain gain, and zero-shot scoring mainly shows label-preference shifts. The overall conclusion is therefore that malware-related continued pre-training does not provide a reliable downstream classification benefit over matched controls.

The five subquestions lead to the following conclusions. For RQ1, low-cost baselines are necessary comparison points: the source-code character TF-IDF baseline reaches AUROC 0.5772 and F1 0.3325, and frozen CodeBERT with a linear classifier reaches AUROC 0.5832 and F1 0.3536. For RQ2, source-code text gives stronger classification signal than NLD text in the paired shallow baseline setup, but the result should not be generalised beyond this benchmark and feature pipeline. For RQ3, continued pre-training does not reliably improve classification: the 300M-token Qwen 0.5B setup gives a small gain over the base model, while the limited-budget Qwen 1.5B scale check does not. For RQ4, malware-related data does not add a clear benefit beyond extra general-code training in the current experiments. For RQ5, malware-related continued pre-training does not produce consistent improvements across model families or direct zero-shot scoring.

The work provides a leakage-aware paired benchmark, a strict binary label policy, low-cost baselines, CodeBERT and Qwen comparisons, zero-shot diagnostics, and a public reproducibility package. The results show why malware-domain adaptation should be evaluated against matched controls rather than against an unadapted model alone.

Future work should first rerun the main comparisons on the stricter clustered split and across multiple random seeds. This would make it clearer whether the small Qwen 0.5B mixed-domain gain is stable or just a one-run effect. The next step is to improve the operating-point analysis with stronger calibration and cost-sensitive threshold selection, because malware classification depends on the balance between missed malware and false alerts.

The benchmark can also be extended beyond source code and NLD. Assembly, binary-derived text, API-call sequences, dynamic traces, and evidence or reasoning fields may capture malware behaviour that source-like text misses. These extensions are promising, but they should follow the same rule as the current study: every claimed improvement should be tied to a fixed split, a matched control, and recoverable experiment artefacts.

References

- [1] Hyrum S. Anderson and Phil Roth. EMBER: An open dataset for training static PE malware machine learning models, 2018. <https://doi.org/10.48550/arXiv.1804.04637>.
- [2] Limin Yang, Arridhana Ciptadi, Ihar Laziuk, Ali Ahmadzadeh, and Gang Wang. BODMAS: An open dataset for learning based temporal analysis of PE malware, 2021. <https://doi.org/10.1109/SPW53761.2021.00020>.
- [3] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge, 2018. <https://doi.org/10.48550/arXiv.1802.10135>.
- [4] Hamed Jelodar, Mohammad Meymani, Samita Bai, Roozbeh Razavi-Far, and Ali A. Ghorbani. SBAN: A framework & multi-dimensional dataset for large language model pre-training and software code mining, 2025. <https://doi.org/10.48550/arXiv.2510.18936>.
- [5] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don't stop pretraining: Adapt language models to domains and tasks, 2020. <https://doi.org/10.18653/v1/2020.acl-main.740>.
- [6] Adam Ibrahim, Benjamin Thérien, Kshitij Gupta, Mats L. Richter, Quentin Anthony, Timothée Lesort, Eugene Belilovsky, and Irina Rish. Simple and scalable strategies to continually pre-train large language models, 2024. <https://doi.org/10.48550/arXiv.2403.08763>.
- [7] Zhifeng Xu, Xianjin Fang, and Gaoming Yang. Malbert: A novel pre-training method for malware detection, 2021. <https://doi.org/10.1016/j.cose.2021.102458>.
- [8] Abir Rahali and Moulay A. Akhloufi. MalBERT: Malware detection using bidirectional encoder representations from transformers, 2021. <https://doi.org/10.1109/SMC52423.2021.9659287>.
- [9] Rajvardhan Oak, Min Du, David Yan, Harshvardhan Takawale, and Idan Amit. Malware detection on highly imbalanced data through sequence modeling, 2019. <https://doi.org/10.1145/3338501.3357374>.
- [10] Ferhat Demirkiran, Aykut Çayır, Uğur Ünal, and Hasan Dağ. An ensemble of pre-trained transformer models for imbalanced multiclass malware classification, 2022. <https://doi.org/10.1016/j.cose.2022.102846>.
- [11] Setia Juli Irzal Ismail, Hendrawan, Budi Rahardjo, Tutun Juhana, and Yasuo Musashi. MalSSL: Self-supervised learning for accurate and label-efficient malware classification, 2024. <https://doi.org/10.1109/ACCESS.2024.3392251>.
- [12] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages, 2020. <https://doi.org/10.18653/v1/2020.findings-emnlp.139>.
- [13] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report, 2024. <https://doi.org/10.48550/arXiv.2409.12186>.
- [14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan,

- Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow, 2020. <https://doi.org/10.48550/arXiv.2009.08366>.
- [15] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021. <https://doi.org/10.18653/v1/2021.emnlp-main.685>.
- [16] Wenxiang Zhao, Juntao Wu, and Zhaoyi Meng. App-Poet: Large language model based android malware detection via multi-view prompt engineering, 2025. <https://doi.org/10.1016/j.eswa.2024.125546>.
- [17] Yun Gao, Hirokazu Hasegawa, Yukiko Yamaguchi, and Hajime Shimada. Malware detection using attributed CFG generated by pre-trained language model with graph isomorphism network, 2022. <https://doi.org/10.1109/COMPSAC54236.2022.00237>.
- [18] Xuezixiang Li, Yu Qu, and Heng Yin. PalmTree: Learning an assembly language model for instruction embedding, 2021. <https://doi.org/10.1145/3460120.3484587>.
- [19] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. jTrans: Jump-aware transformer for binary code similarity detection, 2022. <https://doi.org/10.1145/3533767.3534367>.
- [20] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. Semantic-aware binary code representation with BERT, 2021. <https://doi.org/10.48550/arXiv.2106.05478>.
- [21] Fiorella Artuso, Marco Mormando, Giuseppe Antonio Di Luna, and Leonardo Querzoni. BinBert: Binary code understanding with a fine-tunable and execution-aware transformer, 2025. <https://doi.org/10.1109/TDSC.2024.3397660>.
- [22] Hyungjoon Koo, Soyeon Park, Daejin Choi, and Taesoo Kim. Binary code representation with well-balanced instruction normalization, 2023. <https://doi.org/10.1109/ACCESS.2023.3259481>.
- [23] Hairen Gui, Ke Tang, Zheng Shan, Meng Qiao, Chunyan Zhang, Yizhao Huang, and Fudong Liu. AAPFE: Aligned assembly pre-training function embedding for malware analysis, 2022. <https://doi.org/10.3390/electronics11060940>.
- [24] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. <https://doi.org/10.48550/arXiv.2001.08361>.
- [25] Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models, 2021. <https://doi.org/10.48550/arXiv.2106.09685>.
- [26] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2), 2024. <https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2>.