國立臺北科技大學
National Taipei University of Technology

資訊安全碩士學位學程

碩士學位論文

Master of Science in Information Security

Master Thesis

ThreatDiffusion：基於分層感知遮罩擴散模型與 Token-Slot 對齊之程式碼漏洞偵測與修復框架

ThreatDiffusion: A Framework Based on Tier-Aware Masked Diffusion and Token-Slot Alignment for Code Vulnerability Detection and Repair

研究生：王子杰

Researcher: Zih-Jie Wang

指導教授：陳香君 博士

Advisor: Shiang-Jiun Chen, Ph.D.

June 2026

國立臺北科技大學
研究所碩士學位論文口試委員會審定書

本校 資訊安全碩士學位學程 研究所 王子杰 君

所提論文，經本委員會審定通過，合於碩士資格，特此證明。

學位考試委員會

委　　員：陳香君

馬斐蕾

[illegible]

指導教授：陳香君

所　　長：[illegible]

中 華 民 國　一百一十五　年　六　月　十二　日

Abstract

Keywords: Vulnerability Detection, Vulnerability Repair, Masked Diffusion Model, Secure Code Analysis, Contrastive Learning

With the growing adoption of AI-assisted code generation tools, automatically generated C/C++ code may contain exploitable vulnerabilities, creating a gap between development efficiency and software security. Although the Secure Software Development Lifecycle (SSDLC) emphasizes Shift Left security practices by introducing vulnerability detection and repair during the implementation stage, existing approaches still exhibit several limitations. Traditional static analysis primarily relies on rule-based detection and generic repair suggestions, lacking context-aware automated repair capability. Existing neural repair models commonly formulate repair as a sequence generation problem while rarely considering security-relevant structural features such as API risk, boundary conditions, and control flow. Consequently, learning fine-grained repair behaviors from highly similar vulnerable-repaired code pairs has become an important challenge in automated vulnerability repair.

To address these issues, this thesis proposes ThreatDiffusion, a tier-aware masked diffusion framework for joint vulnerability detection and repair in C/C++ source code. First, Security-Aware Data Preprocessing (SADP) is designed to transform source code into token-level contexts, security-tier labels, and AST/CFG-derived structural threat features. Second, this study proposes the Tier-Aware Security Curriculum (TASC), which guides masking and edit perturbation toward high-risk tokens during pretraining to improve the model's sensitivity to security-critical regions. Finally, this thesis introduces Tier-Aware Span Repair with Contrastive Detection (TASR-CD), which establishes a unified token-level repair supervision format through Token-Slot Alignment and adopts a routing mechanism over Type A and Type B samples to jointly learn localized repair behaviors and calibrate vulnerability detection boundaries.

Experiments are conducted on the VulGate dataset, which contains 236,663 function-level C/C++ samples collected from real-world CVE-related security commits. ThreatDiffusion achieves

a vulnerability-detection F1 score of 88.97% and a Hard-Negative F1 score of 81.35% on the full test set. On the same dataset, the CodeBERT, UniXcoder-Base, and UniXcoder-Base-Nine baselines reported by Safdar et al. achieve F1 scores of 85.90%, 87.00%, and 88.90%, respectively.

ThreatDiffusion provides detection performance comparable to the strongest baseline while also supporting localized repair prediction within the same framework. For the repair task, ThreatDiffusion achieves 34.59% Span Token Accuracy, 26.53% Aligned-record %PP, and 97.13% Full Token Accuracy on the Type A test samples. The results indicate that ThreatDiffusion can learn localized editing behavior from real-world vulnerability repairs while maintaining stable vulnerability-detection performance.

TAIPEI TECH

Acknowledgements

歲月落筆，旅途終有成章。回首這段求學時光，每一次相遇與陪伴，都留下了珍貴的痕跡。在本論文完成之際，向所有曾給予我指導與支持的人，致上最眞摯的感謝。

首先，想將這份感謝留給自己。感謝當初有勇氣離開熟悉的領域，選擇一條全新的道路。從陌生開始學習，在一次次嘗試與失敗中尋找答案，也曾爲了一個實驗結果反覆推敲，只希望每個結果都經得起檢驗。所幸至今，始終沒有失去對學習的熱忱與對研究的要求。

此外，衷心感謝指導教授陳香君老師，在論文研究期間給予我悉心的指導，使我得以將資訊與數學領域的知識相互結合，逐步完成這篇論文。老師的教誨與提點，我將銘記於心。

也感謝一直陪伴與支持我的家人。謝謝你們始終尊重我的選擇，給予我最大的支持與信任，讓我能夠沒有後顧之憂地完成學業。有些感謝或許難以言盡，但你們始終是我最安心的依靠。

誠摯地感謝研究所期間遇見的夥伴，佳蓉、皓崴、榕佑、雋洋、少雷、凱俊，以及實驗室的所有夥伴。謝謝你們在課堂與論文上的交流，也謝謝研究之外生活中的陪伴。一起討論、一起忙碌，也一起分享生活中的大小事，讓這段研究所時光多了許多值得珍藏的回憶。

最後，感謝在人生不同階段曾教導、幫助與陪伴我的師長及朋友們，每一段相遇，都留下了珍貴的痕跡。也感謝那些在疲憊時，以作品與舞台帶給我力量的偶像們，爲忙碌的日子增添了一份期待。

鐘錶的指針周而復始，就像一路走來的困惑、煩惱與迷惘，總有搖擺不定的時候。但時間仍會向前，而人也終究要繼續前行。如今，鐘聲爲這段求學時光畫下句點，而指針依舊落在前方。新的篇章仍待落筆，而前方的路，也將由我繼續書寫。

王子杰　謹誌於

臺北科技大學資訊安全碩士學位學程

中華民國 115 年 8 月 10 日

Table of Contents

Abstract i

Acknowledgements iii

Table of Contents iv

List of Figures vii

List of Tables viii

Chapter 1 Introduction 1

Chapter 2 Related Work 6

2.1 Automated Vulnerability Detection 6

2.1.1 Traditional and Rule-Based Methods 6

2.1.2 Deep Learning-Based Detection 7

2.2 Automated Vulnerability Repair 8

2.2.1 Traditional and Search-Based Repair 8

2.2.2 Sequence-to-Sequence Approaches 8

2.2.3 Large Language Model-Based Repair 9

2.3 Code Representation Learning 9

2.3.1 From Token to Graph-Based Representations 10

2.3.2 Pre-trained Code Language Models 10

2.4 Masked Diffusion Language Models 11

2.4.1 Discrete Diffusion Models 11

2.4.2 Curriculum and Masking Strategies 13

2.4.3 Diffusion-Based Code Generation and Modeling 13

2.5 Learning Objectives and Multi-Task Optimization 14

2.5.1 Contrastive Learning for Code 15

2.5.2 Class-Imbalance-Aware Objectives 16

2.5.3 Multi-Task Learning and Gradient Conflict 16

Chapter 3 Methodology 18

3.1 Security-Aware Data Preprocessing 19

TAIPEI TECH

3.1.1 VulGate Dataset . . . 19
3.1.2 BPE Tokenization . . . 21
3.1.3 AST-CFG Parser . . . 22
3.1.4 Threat Vector . . . 23
3.1.5 Per-Token Tier Map . . . 24
3.1.6 Input Representation . . . 25
3.2 Tier-Aware Masked Diffusion . . . 26
3.2.1 Shared Transformer Backbone . . . 26
3.2.2 Tier-Aware Security Curriculum . . . 28
3.2.3 Repair-Pair Validation and Type Routing . . . 33
3.2.4 Tier-Aware Span Repair with Contrastive Detection . . . 35
Chapter 4 Implementation . . . 43
4.1 Environment Setup . . . 43
4.2 Security-Aware Data Preprocessing . . . 44
4.2.1 Dataset Loading, Feature Construction, and Sample Routing . . . 45
4.2.2 Tokenization and Static Feature Extraction . . . 45
4.2.3 Unified Input Serialization . . . 48
4.3 Tier-Aware Security Curriculum Implementation . . . 48
4.3.1 Backbone Initialization . . . 48
4.3.2 Two-Phase TASC Training . . . 49
4.4 Token-Slot Alignment Implementation . . . 50
4.5 TASR-CD Fine-Tuning Implementation . . . 52
4.5.1 Prediction Heads . . . 52
4.5.2 Type-A and Type-B Batch Scheduling . . . 52
4.6 Evaluation Protocol . . . 56
4.6.1 Inference and Repair Reconstruction . . . 56
4.6.2 Detection Metrics . . . 57
4.6.3 Repair Metrics . . . 57
4.6.4 Checkpoint Selection . . . 58

TAIPEI TECH

4.6.5 Ablation and Baseline Evaluation . . . 58
4.7 Implementation Summary . . . 59
Chapter 5 Results and Analysis . . . 60
5.1 Evaluation Metrics . . . 60
5.1.1 Vulnerability Detection Metrics . . . 60
5.1.2 Vulnerability Repair Metrics . . . 61
5.2 TASC Pre-training Analysis . . . 63
5.3 TASR-CD Fine-tuning Analysis . . . 65
5.4 Overall Performance Evaluation . . . 68
5.4.1 Detection and Repair Performance of ThreatDiffusion . . . 68
5.4.2 Baseline Comparison on Vulnerability Detection . . . 69
5.5 Detection and Repair Analysis . . . 69
5.6 Ablation Study . . . 71
Chapter 6 Conclusion and Future Work . . . 75
6.1 Conclusion . . . 75
6.2 Future Work . . . 77
References . . . 79
Appendix A: Mathematical Notation . . . 86
A.1 Notation Used in Chapter 2 . . . 86
A.2 Notation Used in Chapter 3 . . . 88
A.3 Notation Used in Chapter 5 . . . 98

TAIPEI TECH

List of Figures

3.1 Two-layer modular architecture of ThreatDiffusion. 18

3.2 Security-Aware AST subtree extraction example. 22

3.3 Two-phase structure of the Tier-Aware Security Curriculum (TASC). 29

3.4 Illustration of the Token-Slot Alignment mechanism and the TASR-CD transformer's end-to-end prediction for an insecure file creation vulnerability patch. 36

5.1 TASC pre-training dynamics. 64

5.2 Training loss trends during TASR-CD fine-tuning. 66

5.3 TASR-CD detection dynamics. 66

5.4 TASR-CD repair dynamics. 67

5.5 Confusion matrix of vulnerability detection on the test set. 70

TAIPEI TECH

List of Tables

3.1 Dataset split used in ThreatDiffusion. 20
3.2 VulGate dataset statistics. 20
3.3 Representative fields extracted from VulGate samples. 21
3.4 Validation rules for corrupted repair pairs. 34
3.5 Repair-pair validation statistics. 34

4.1 Hardware specifications used in the experiments. 43
4.2 Software components used by ThreatDiffusion. 44
4.3 Training sample routing after Repair-Pair Validation. 45
4.4 Shared Transformer backbone configuration. 49
4.5 TASC training configuration. 50
4.6 Prediction heads implemented in TASR-CD. 52
4.7 TASR-CD fine-tuning configuration. 55
4.8 TASR-CD loss weights. 56
4.9 Evaluation metrics used in ThreatDiffusion. 58

5.1 Masking-policy comparison in the TASC pre-training evaluation. 65
5.2 Overall test performance of ThreatDiffusion. 68
5.3 Detection comparison on the VulGate dataset. 69
5.4 Tier-wise vulnerability detection performance. 70
5.5 Token-level top-K oracle over repair-span positions. 71
5.6 Detection performance ablation. 72
5.7 Repair performance ablation. 73

TAIPEI TECH

Chapter 1 Introduction

The broad adoption of AI-assisted code generation tools has improved software development productivity, but it has also introduced new security concerns. Recent studies show that source code generated by these tools can contain exploitable vulnerabilities. Pearce et al. [1] report that approximately 40% of programs generated by GitHub Copilot contain vulnerabilities associated with the CWE Top 25. Tihanyi et al. [2] further show that, across multiple state-of-the-art models, at least 62.07% of LLM-generated C programs contain detectable vulnerabilities. When developers accept generated code without sufficient security expertise, such defects may not be identified during code review. As AI-assisted development becomes increasingly common, the gap between development efficiency and security assurance continues to widen.

The growing frequency and complexity of software security issues also increases the need to move security validation earlier in the development process. Khan et al. [3] argue that security practices should be incorporated into each phase of the Software Development Lifecycle (SDLC), although this integration remains difficult in practice. This security-integrated development model is commonly referred to as the Secure Software Development Lifecycle (SSDLC). It emphasizes vulnerability detection and repair during implementation, before code reaches later verification stages. In practice, however, the tools available to developers at this stage remain limited. Static Application Security Testing (SAST) tools are commonly used to detect vulnerabilities earlier in development, but recent empirical evidence shows deficiencies in their practical adoption and configuration [4]. These tools can mark suspicious code regions and provide generic repair suggestions, but such suggestions are usually rule-based templates that still require manual inspection and modification. They do not directly generate context-aware repair outputs for developers. Empirical analysis further reports that at least 76% of alerts produced by a single SAST tool are unrelated to real vulnerabilities [5], which results in considerable manual triage overhead. As a result, vulnerability repair in practical C/C++ development still relies heavily on developer expertise, particularly for defects involving unsafe APIs, memory safety, and complex execution logic. Recent evaluations of LLM-based vulnerability repair on real-world C/C++ programs also show that existing models perform better on localized memory

errors, but remain limited when repairs require deeper program semantics and cross-concern reasoning [6].

This dependence on manual repair is also reflected in academic research. Germano et al. [7] report that vulnerability detection accounts for 91.3% of the security literature, whereas automated repair accounts for only 11.1%. Existing repair frameworks are still largely based on Sequence-to-Sequence architectures [8], such as VulRepair [9] and RAP-Gen [10], which formulate repair as a translation task over the entire function. In real-world repair scenarios, however, security patches often involve only localized code changes. A repair may modify only a small number of tokens or statements while leaving most of the function unchanged. Under highly similar vulnerable-repaired pairs, conventional sequence generation frameworks may therefore struggle to model fine-grained repair behavior. Diffusion-based methods that model repair as masked token reconstruction over the full sequence remain relatively underexplored in vulnerability repair [11, 12], despite their potential for fine-grained and position-aware repair learning.

Based on these observations, this thesis develops ThreatDiffusion, a tier-aware masked diffusion framework for joint vulnerability detection and repair in C/C++ source code. Unlike approaches that treat repair as pure sequence generation, ThreatDiffusion focuses on localized repair behavior under highly similar vulnerable-repaired code pairs and incorporates structural security information into the repair process. The framework is designed around four challenges commonly encountered in real-world vulnerability repair.

1. **Hard-Negative Samples Blurring Vulnerability Boundaries**

   Many vulnerable-repaired code pairs differ by only a small number of tokens, yet their execution behavior may be fundamentally different. In VulGate, 89% of vulnerable-repaired pairs have cosine similarity of at least 0.90 [13]. Many repairs change only one or two tokens while preserving nearly identical lexical structure. Therefore, models that rely mainly on lexical similarity may fail to distinguish vulnerable code from its repaired counterpart [14]. Recent realistic benchmarks further show that the performance of existing code language models can be overestimated on conventional vulnerability datasets, indicating the need for models that capture subtle security-relevant differences in real-world

C/C++ code [15].

2. **Sequence Misalignment of Repair Signals**

   Real-world vulnerability repair often involves insertion, deletion, and substitution operations, which cause repaired sequences to differ in length from their vulnerable counterparts. Encoder-only masked diffusion models operate on fixed-length sequences by design [16]. Existing diffusion-based repair frameworks that adopt this formulation [11] are not designed to supervise variable-length repair operations at the token level.

3. **Gradient Interference in Joint Optimization**

   Joint optimization of repair and detection objectives introduces gradient interference between generative and classification supervision. In multi-task learning, shared models typically aggregate task-specific gradients into a common update direction, and uncertainty in this aggregation can affect the stability of shared representations [17]. Recent work further shows that task imbalance and dominance may cause one objective to exert disproportionate influence during joint optimization [18]. In ThreatDiffusion, repeatedly computing detection loss on repair samples carrying the same vulnerable label may bias the classification boundary toward high-recall predictions. This potential shift is referred to in this thesis as Recall Drift.

4. **Sparse Security Semantics Under Uniform Masking**

   Uniform masking, commonly used in masked diffusion models [16, 19], does not distinguish token positions by their security relevance. When security-critical API tokens, such as API calls covered by SEI CERT C rules [20], occupy only a small part of a function, many masked positions may fall on benign background tokens. Security-critical regions may therefore receive limited reconstruction emphasis.

To address these challenges, ThreatDiffusion integrates structural threat modeling, Token-Slot Alignment, sample-routed joint optimization, and a Tier-Aware Security Curriculum (TASC) within a unified diffusion learning architecture.

TAIPEI TECH

1. **Structural Threat Embedding**

   ThreatDiffusion constructs per-token security tier labels $\tau^{(i)} \in \{0, 1, 2, 5\}$ through SEI CERT C API classification [20] and guard-condition analysis derived from AST and CFG parsing [21, 22]. A 64-dimensional threat feature vector further encodes API risk, control-flow complexity, and taint-propagation behavior. These structural features are fused with the other feature streams to form the unified input, providing the diffusion model with explicit security context that is not directly represented by lexical token embeddings alone. The empirical analysis in Chapter 5 shows the largest HN-F1 reductions under the TASC-to-uniform and –TASR-CD configurations, while the individual tier and structural-feature ablations produce smaller changes.

2. **Token-Slot Span Alignment**

   A `difflib`-based alignment mechanism constructs fixed-capacity span buffers at each edit location. This design keeps vulnerable and repaired token sequences at the same length and preserves positional correspondence across insertion, deletion, and substitution operations. It allows an encoder-only diffusion framework [16] to perform position-level repair supervision without approximating variable-length edits as full-sequence generation.

3. **Sample-Routed Joint Training via TASR-CD**

   Tier-Aware Span Repair with Contrastive Detection (TASR-CD) separates repair and detection training into two sample-routing paths. Type A samples optimize the full joint objective, including repair, contrastive learning [23, 24], operation-type prediction, slot-count prediction, and deletion supervision. Type B samples optimize only the detection objective using balanced vulnerable and benign labels. Type B batches are interleaved with Type A processing to provide balanced detection gradients while preserving repair supervision. During fine-tuning, the UniXcoder embedding module remains frozen to prevent repair gradients from shifting the pre-trained lexical representation.

TAIPEI TECH

4. **Tier-Aware Security Curriculum**

   Tier-Aware Security Curriculum (TASC) is inspired by the two-stage curriculum of Seed Diffusion [19] and adapts it to vulnerability repair through tier-aware masking and reconstruction weighting. Phase 1 applies non-uniform masking probabilities scaled by security tiers, concentrating diffusion learning on security-critical token regions. Phase 2 introduces tier-weighted edit supervision to improve sensitivity to fine-grained repair operations. Together, the two phases guide the model toward security-critical regions during repair and detection learning.

The remainder of this thesis is organized as follows. Chapter 2 reviews related work on vulnerability detection, automated repair, code representation learning, and masked diffusion models. Chapter 3 presents the ThreatDiffusion framework and methodology. Chapter 4 details the implementation of the framework. Chapter 5 reports experimental results and ablation studies. Chapter 6 concludes this thesis and discusses future work.

TAIPEI TECH

Chapter 2 Related Work

Research on C/C++ vulnerability analysis spans detection, repair, code representation, generative modeling, and optimization. These areas are often studied separately, although they are closely related in systems that must both identify and modify vulnerable code. This chapter reviews prior work across these areas, with particular attention to limitations that remain under realistic evaluation settings.

2.1 Automated Vulnerability Detection

Vulnerability detectors either reason from explicitly defined program properties or learn patterns from data. Rule-based analysis offers traceable evidence but is bounded by its specifications; learning-based methods broaden coverage at the cost of data dependence and uncertain generalization. This distinction provides the starting point for the following review.

2.1.1 Traditional and Rule-Based Methods

Early tools such as Flawfinder scanned source code for dangerous functions and vulnerable patterns [25]. Although lightweight, they could not reason beyond surface syntax. Later analyzers introduced program-aware reasoning: CodeQL expresses security queries over structured code databases [26], while Meta Infer uses the interprocedural Pulse analysis for memory-safety defects, replacing its original bi-abduction analysis [27]. Symbolic execution likewise provides precise reasoning over bounded paths by treating inputs symbolically, but its path space can grow prohibitively on large systems.

Rule coverage and analysis cost remain practical constraints. Semantic patterns outside the specification may be missed, and path-sensitive reasoning becomes expensive as programs grow. Most analyzers also stop after reporting a defect. The need for broader coverage led researchers to learn vulnerability patterns from code rather than specify every pattern by hand.

2.1.2 Deep Learning-Based Detection

Russell et al. [28] showed that neural networks can learn vulnerability patterns from lexer-generated token sequences. VulDeePecker modeled data flow with bidirectional LSTMs over code gadgets centered on library and API calls [29]. SySeVR broadened the input by combining syntactic and semantic program slices [30]. These models reduced dependence on fixed rules, but their primary representation remained sequential.

Devign made program structure explicit through an AST-backed graph with typed control-flow and data-dependency edges, followed by a gated graph neural network for function classification [31]. Its gains over the compared baselines did not settle the question of generalization. On Chromium and Debian, Chakraborty et al. [32] observed substantial degradation outside controlled settings.

Pre-trained code models offered another route to stronger representations. LineVul fine-tuned CodeBERT for function-level prediction and outperformed earlier graph-based approaches on large C/C++ datasets [33]. Later systems combined pre-training with explicit structure. Vul-LMGNNs coupled language models and GNNs through online knowledge distillation [34]; LLMxCPG used CPG-guided slices to retain vulnerability-relevant context while reducing the amount of code supplied to an LLM [35].

Benchmark design emerged as a separate source of error. Risse and Böhme studied this failure mode [14]. They found that modern detectors often exploit label-unrelated features and fail to separate vulnerable functions from their repaired counterparts. PrimeVul likewise showed that conventional benchmarks can overestimate performance under realistic splits [15]. Safdar et al. [13] estimated that nearly 89% of their C/C++ training set consisted of Hard-Negative samples, defined as vulnerable and non-vulnerable functions with cosine similarity of at least 0.90. The broader literature remains detection-heavy: among the 208 studies reviewed by Germano et al. [7] from 2018 through August 2025, 91.3% addressed detection and 11.1% repair, with studies allowed to cover more than one task.

Benchmark gains therefore do not establish robustness to realistic splits or Hard-Negative pairs. Detection also ends before remediation: a vulnerability label does not specify which to-

kens must change or whether the resulting program remains correct. Automated repair addresses this second problem.

2.2 Automated Vulnerability Repair

Automated vulnerability repair generates program changes rather than vulnerability labels. Early systems searched a predefined patch space; later work learned transformations from historical fixes and, more recently, from pre-trained language models. Across these approaches, producing a plausible patch is easier than establishing its semantic correctness.

2.2.1 Traditional and Search-Based Repair

Early systems generated candidate patches from predefined transformations and validated them with tests. GenProg used genetic programming to mutate and recombine statements, demonstrating test-suite-adequate repair [36]. APR4Vul later separated plausible, security-fixing, and correct patches; passing tests or removing a security symptom did not establish semantic correctness [37]. Angelix derived repair constraints from failing executions and synthesized changes across multiple expressions and locations [38]. Its scope still depends on the supported expressions, fault localization, and available tests.

Incomplete security tests leave many generated patches weakly specified, while the search space grows rapidly with program complexity. Learning-based repair replaces explicit exploration with transformations inferred from historical fixes.

2.2.2 Sequence-to-Sequence Approaches

Sequence-to-sequence learning formulates repair as a mapping from buggy code to a corrected sequence, following the general neural translation framework [8]. VulRepair fine-tuned CodeT5 on repair pairs from CVEfixes and Big-Vul; pre-trained initialization produced the largest gain in its ablation study [9]. RAP-Gen supplied CodeT5 with prior fixes selected by BM25 and dense retrieval [10]. Iteration alone was not sufficient: under equal inference bud-

gets, Olausson et al. [39] reported inconsistent self-repair gains on Python generation tasks.

The validity of learned repair also depends on how historical pairs are collected and split. PrimeVul studied detection rather than repair, but its evidence that duplicates and unrealistic splits inflate performance is directly relevant when repair models are evaluated on related code revisions [15].

2.2.3 Large Language Model-Based Repair

Pre-trained language models broadened the set of repair transformations available to the decoder. Zhang et al. [40] reported that five evaluated code models outperformed VRepair in their setting, although accuracy varied across CWE categories. Results on real Java vulnerabilities were less favorable: on Vul4J and VJBench, Wu et al. [41] found that the evaluated LLM and deep-learning systems repaired few cases and often produced uncompilable patches. VulMaster incorporated additional evidence by extending CodeT5 with Fusion-in-Decoder over segmented vulnerable functions, AST structure, and CWE knowledge [42].

The same limitation appears in C/C++. For real memory-corruption vulnerabilities, the evaluated LLMs handled localized errors more effectively than repairs that required cross-cutting changes or deeper program semantics [6].

The move from test-guided search to neural generation expanded the available patch space but did not close the gap between plausible and correct repairs. Insertions, deletions, and coordinated changes across locations remain harder than local replacement, particularly when tests provide little behavioral evidence. For learned repair, this difficulty begins with the input representation: information omitted before generation cannot be recovered by the decoder.

2.3 Code Representation Learning

Code representation determines which lexical, structural, and semantic signals remain available to a detector or repair model. The main approaches use token sequences, explicit program graphs, or representations learned by pre-trained code models [43]. They preserve different program properties and impose different preprocessing and computational costs.

2.3.1 From Token to Graph-Based Representations

Early systems treated code as flat token sequences, a progression surveyed by Sun et al. [44] and demonstrated for vulnerability detection by Russell et al. [28]. Such sequences retain lexical order but do not explicitly encode control flow or data dependencies. Consistent with this limitation, Agarwal et al. [45] found that concrete-syntax-tree adaptation improved the evaluated translation, generation, and summarization tasks over plain-text modeling.

Graph representations address this omission directly. The Code Property Graph unifies AST, control-flow, and program-dependence information and supported the discovery of 18 previously unknown Linux-kernel vulnerabilities in the same study [46]. Devign subsequently applied gated GNNs to integrated program graphs [31]. More recent work combines structure with language models: GRACE incorporates graph information into in-context vulnerability detection [47], while Vul-LMGNNs and LLMxCPG couple graph-derived structure with pre-trained semantic representations [34, 35].

Graph construction, however, requires parsers and analysis pipelines for each language. Large graphs also increase computational cost and complicate long-range modeling. These constraints limit purely graph-based scaling and motivate pre-trained models that learn contextual information from large code corpora.

2.3.2 Pre-trained Code Language Models

CodeBERT brought bidirectional Transformer pre-training to paired natural language and source code, using masked language modeling and replaced-token detection [48]. GraphCodeBERT then incorporated data flow so that pre-training also captures variable definition–use relations [49], improving over the compared code-understanding baselines. These models provide contextual representations without constructing a task-specific graph for every sample.

UniXcoder provides a unified architecture for code understanding and generation [50]. It can operate in encoder-only, decoder-only, or encoder-decoder mode. Safdar et al. [13] evaluated UniXcoder-Base-Nine, a variant trained on C, C++, and C# code pairs. It achieved an F1 score of 88.9% on refined VulGate.

Token sequences, program graphs, and pre-trained Transformers are complementary rather than interchangeable. Their reported advantages depend on the downstream task and training setting, and none preserves every security-relevant property without additional cost. Once the input has been represented, a separate modeling choice determines how a repaired sequence is generated.

2.4 Masked Diffusion Language Models

Diffusion models generate data by reversing a gradual corruption applied to clean input x_0. The original formulation operates in continuous spaces, whereas text and code require corruption over discrete tokens. Iterative denoising is useful for code reconstruction because a prediction can condition on both sides of a corrupted position instead of following a fixed left-to-right order. The discussion below focuses on masked discrete diffusion, staged corruption, and diffusion-based code generation, including the assumptions that become restrictive for vulnerability repair.

2.4.1 Discrete Diffusion Models

Standard diffusion perturbs continuous data according to a noise schedule and learns the reverse process by optimizing a variational lower bound on log-likelihood (ELBO). Discrete sequences cannot use additive Gaussian noise directly. D3PM instead defines the forward process as a Markov chain governed by categorical transition matrices [51]. Its absorbing variant maps tokens to a special `[MASK]` state, from which the learned reverse process reconstructs the original token.

Under this absorbing formulation, each token position is independently kept unchanged or replaced by the mask state according to a time-dependent noise schedule:

$$q\left(x_t^{(i)} \mid x_0^{(i)}\right) = \begin{cases} \alpha_t, & x_t^{(i)} = x_0^{(i)}, \\ 1 - \alpha_t, & x_t^{(i)} = m, \end{cases} \tag{2.1}$$

where $x_0^{(i)}$ is the clean token at position i, m denotes the absorbing `[MASK]` state, and α_t

is a monotonically decreasing clean-token retention schedule. Once masked, a token remains masked until it is recovered by the learned reverse process. Although D3PM established a general framework for discrete diffusion modeling, its training objective remained computationally expensive and difficult to optimize in practice.

MDLM restricts the forward process to absorbing masks and derives a Rao-Blackwellized continuous-time ELBO [16]. This reduces the optimization cost of the general D3PM objective. Supervision is applied only where the current token is masked, yielding a weighted masked-language-modeling loss.

The resulting training objective can be written as:

$$\mathcal{L}_{\text{MDLM}}(\theta) = \mathbb{E}_{t,\, q(x_t|x_0)} \left[\frac{-\alpha_t'}{1-\alpha_t} \sum_{i:\, x_t^{(i)}=m} -\log \mu_\theta^{(i)}(x_t, t)\left[x_0^{(i)}\right] \right], \tag{2.2}$$

where $\mu_\theta^{(i)}(x_t, t)$ is the predicted clean-token distribution at position i. Since α_t decreases with time, $-\alpha_t'/(1-\alpha_t)$ is non-negative. Training therefore has the form of weighted masked language modeling and does not require a separate autoregressive decoder. Shi et al. [52] derived an equivalent simplified ELBO using the complementary masking parameterization.

MDLM further adopts the SUBS parameterization, which leaves unmasked tokens unchanged and predicts a clean-token distribution only when the current state is masked. For $0 \le s \le t$, the reverse transition is:

$$p_\theta(x_s \mid x_t) = \begin{cases} \text{Cat}(x_s;\, x_t)\,, & x_t \neq m, \\ \text{Cat}\left(x_s;\, \dfrac{(\alpha_s - \alpha_t)\mu_\theta(x_t, t) + (1-\alpha_s)\mathbf{e}_m}{1-\alpha_t}\right), & x_t = m, \end{cases} \tag{2.3}$$

where $\mathbf{e}_m$ is the one-hot vector of the absorbing mask state. An equivalent formulation uses the complementary masking probability notation

$$\gamma_t = 1 - \alpha_t, \qquad \frac{-\alpha_t'}{1-\alpha_t} = \frac{\gamma_t'}{\gamma_t}, \tag{2.4}$$

Here, the corruption probability increases with t. Under $\gamma_t = 1 - \alpha_t$, the weighting becomes

γ'_t/γ_t; this equivalence is a change of notation rather than a methodological modification.

Equations 2.1, 2.2, and 2.3 summarize the standard absorbing-state masked diffusion formulation. Within a single absorbing-state trajectory, corruption replaces token identities with masks but does not insert or delete sequence positions.

Beyond the reverse transition itself, a timestep representation supplies a denoising network with the current noise level. Sinusoidal basis functions provide one general construction derived from Transformer positional encoding [53]. D3PM defines discrete transition schedules but does not prescribe this particular timestep-embedding construction [51].

2.4.2 Curriculum and Masking Strategies

Fixed corruption is not required throughout training. Seed Diffusion uses a staged curriculum to address inefficient random-order learning signals and iterative denoising latency [19]. Mask-based corruption is used for the first 80% of training; insertion, deletion, and substitution are added for the final 20%. The method avoids carry-over unmasking because copying unmasked inputs teaches the model to regard them as correct and weakens self-correction at inference. It also suppresses the `[MASK]` logit to prevent masks from appearing in the output. This curriculum moves from masking to general edit corruption, but it does not distinguish tokens by security importance.

2.4.3 Diffusion-Based Code Generation and Modeling

Diffusion has also been applied directly to code generation, a task otherwise dominated by autoregressive language models [54]. Bidirectional denoising allows earlier token predictions to be reconsidered and supports partial parallelism. CodeFusion maps discrete tokens into a continuous latent space, applies Gaussian diffusion to the latent embeddings, and reconstructs code with a learned decoder [55]. Its evaluation concerns natural-language-to-code generation rather than vulnerability repair, and the diffusion process itself remains continuous.

Subsequent work shifted toward masked discrete diffusion directly over token sequences. Onan and Alhumyani [11] proposed CodeDiffuSe, which masks syntactic subtrees rather than in-

dividual tokens during the forward process. CodeDiffuSe also incorporates semantic-consistency regularization for syntactic validity and type consistency. Gong et al. [12] investigated DiffuCoder and showed that masked diffusion permits a non-strictly left-to-right and partly parallel generation order. DiffuCoder further introduced coupled-GRPO, a reinforcement-learning method designed to reduce training variance while improving generation quality.

The three code models do not use the same diffusion process. CodeFusion applies Gaussian diffusion in a continuous latent space, while CodeDiffuSe and DiffuCoder corrupt discrete tokens with masks. All three were evaluated on general code-generation tasks rather than vulnerability-repair benchmarks.

Existing diffusion models provide two useful capabilities for code repair: bidirectional token reconstruction and a training path from simple masks to richer edits. Two assumptions remain restrictive. Standard schedules assign corruption without regard to security importance, and an absorbing-state trajectory preserves the number of sequence positions. Sparse security-critical tokens and variable-length patches are therefore not handled by the standard formulation alone. Their treatment also depends on how the objective distributes learning signal across tokens, examples, and tasks.

2.5 Learning Objectives and Multi-Task Optimization

A diffusion process specifies corruption and recovery, but the loss determines where the gradient is concentrated. Detection separates security states at the sample level; repair predicts localized changes at token level, where security-critical positions are sparse. Shared parameters couple these signals and allow one objective to reinforce or interfere with another. Contrastive learning, imbalance-aware losses, and multi-task optimization address different parts of this training problem.

2.5.1 Contrastive Learning for Code

Contrastive learning trains representations by pulling semantically similar instances closer together while pushing dissimilar instances farther apart, without requiring explicit label prediction at every training step. Its core mechanism relies on constructing positive pairs, which should share similar representations, and negative pairs, whose representations should remain separable. Rather than directly optimizing a fixed classification boundary, contrastive objectives shape the geometry of the embedding space itself. Van den Oord et al. [23] introduced Contrastive Predictive Coding, which uses the InfoNCE objective to maximize a lower bound on the mutual information between a context representation and future observations.

The standard contrastive objective is the InfoNCE loss, which treats the positive pair as the correct class in a softmax classification over a set of negatives. For an anchor representation, InfoNCE increases similarity to positive samples while decreasing similarity to negatives in the same batch:

$$\mathcal{L}_{\text{InfoNCE}} = -\log \frac{\exp(\text{sim}(z_i, z_i^+)/\tau)}{\exp(\text{sim}(z_i, z_i^+)/\tau) + \sum_{z_j \in \mathcal{N}(i)} \exp(\text{sim}(z_i, z_j)/\tau)}, \tag{2.5}$$

where z_i is the anchor representation, z_i^+ is its positive representation, $\mathcal{N}(i)$ is the negative set, $\text{sim}(\cdot, \cdot)$ denotes cosine similarity, and $\tau > 0$ is the temperature controlling the concentration of the similarity distribution. Khosla et al. [24] extended this framework to supervised contrastive learning by allowing multiple same-class positives for each anchor. In batch-averaged form, the supervised contrastive objective is commonly written as:

$$\mathcal{L}_{\text{supcon}} = -\frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} \frac{1}{|P(i)|} \sum_{p \in P(i)} \log \frac{\exp(\text{sim}(z_i, z_p)/\tau)}{\sum_{a \neq i} \exp(\text{sim}(z_i, z_a)/\tau)}, \tag{2.6}$$

where $\mathcal{I}$ is the set of anchors in the batch, $P(i)$ is the set of same-label positives for anchor i, and a indexes all samples other than the anchor. This objective learns class-aware representation spaces.

Code-oriented contrastive methods differ mainly in how they construct pairs. ContraBERT treats semantics-preserving transformations of programming and natural language as positives

and reports improved robustness to the evaluated perturbations [56]. In vulnerability detection, Safdar et al. [13] define Hard-Negative samples as vulnerable and non-vulnerable function pairs with cosine similarity of at least 0.90. Such pairs place different security labels close together in the representation space and weaken a boundary learned only by classification.

Pair construction determines the value of a contrastive objective. Random negatives contribute little once they are already separated, whereas Hard-Negative samples place the learning signal near the decision boundary. Temperature controls how strongly this local structure is emphasized. Contrastive learning can therefore separate similar representations, but it does not specify the token edits needed to repair code.

2.5.2 Class-Imbalance-Aware Objectives

Class imbalance creates a separate optimization problem. Even with informative pairs, standard cross-entropy can be dominated by abundant examples that are already classified correctly. Focal Loss reduces their contribution through a confidence-dependent factor [57]:

$$\mathcal{L}_{\text{focal}} = -\alpha_f (1 - p_t)^{\gamma_f} \log(p_t), \tag{2.7}$$

where p_t denotes the model probability assigned to the correct class, α_f is the class-balancing coefficient, and γ_f is the focusing parameter controlling the suppression of well-classified examples.

The focusing factor shifts learning away from abundant easy examples. The same principle applies when security-relevant samples or token positions occupy only a small part of the data, although the weighting unit depends on the supervision level. Reweighting an individual loss does not, however, control its interaction with other objectives that update the same parameters.

2.5.3 Multi-Task Learning and Gradient Conflict

Multi-task optimization addresses this interaction directly. Yu et al. [58] associate detrimental interference with a combination of conflicting gradients, high curvature, and large dif-

ferences in gradient magnitude. Their PCGrad algorithm projects one task gradient when it conflicts with another instead of accepting the conflicting component in a naively aggregated update.

Beyond pairwise gradient projection, recent studies have revisited this issue from the perspective of gradient aggregation and task balancing. Achituve et al. [17] point out that multi-task optimization commonly computes task-specific gradients and aggregates them into a shared update direction, but the sensitivity and uncertainty of different gradient dimensions are often ignored. Their Bayesian aggregation framework shows that such uncertainty should be considered when updating shared representations. Zhou et al. [18] further argue that multi-task optimization should address not only directional gradient conflict but also task imbalance and dominance, where one objective can exert disproportionate influence on the shared model.

Parameter sharing does not guarantee compatible learning dynamics. Detection and repair may benefit from a common representation, but their supervision acts at different granularities and emphasizes different properties of the same code. A joint objective must account for both gradient direction and unequal task influence; a weighted sum alone does not ensure that either objective is preserved.

Four limitations recur across the reviewed work. Detection models remain sensitive to Hard-Negative samples; repair systems do not reliably close the gap between plausible and correct patches; standard absorbing diffusion preserves sequence positions and ignores token-specific security importance; and shared detection–repair training can produce unequal or conflicting gradients. These limitations link representation, variable-length generation, and optimization rather than treating them as independent problems. Chapter 3 presents the methodology used to study them in a unified framework.

TAIPEI TECH

Chapter 3 Methodology

This chapter describes the methodology of ThreatDiffusion, a tier-aware masked diffusion framework designed for joint vulnerability detection and repair in C/C++ source code. The design is motivated by the observation developed in the preceding chapters: vulnerable and repaired functions often remain lexically similar, yet their security distinction is determined by localized edits, API risk, guard conditions, and control-flow context. As illustrated in Figure 3.1, ThreatDiffusion therefore combines Security-Aware Data Preprocessing (SADP) with a masked diffusion backbone so that repair learning is guided by both token-level context and explicit structural threat signals.

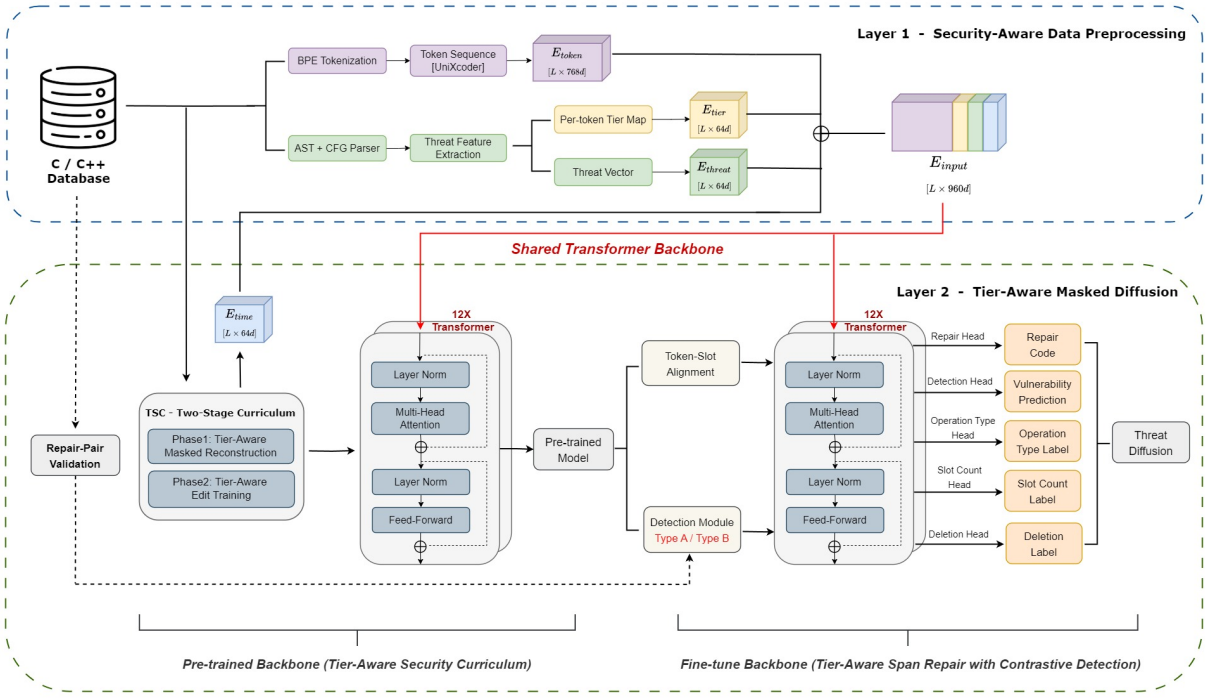


Figure 3.1: Two-layer modular architecture of ThreatDiffusion.

Figure 3.1 presents the framework in two layers. The upper layer constructs token, tier, threat, and timestep feature streams and fuses them as the input to the shared Transformer. The lower layer presents the training pipeline. TASC learns a security-aware initialization through tier-aware reconstruction and edit corruption. TASR-CD then uses Type A/Type B routing to jointly optimize aligned repair, vulnerability detection, and edit-structure prediction.

ThreatDiffusion builds on the BPE tokenizer and embedding module of UniXcoder [50],

the Pre-LN Transformer architecture [53, 59], the absorbing-state diffusion formulation of D3PM and MDLM [51, 16], and the two-phase curriculum of Seed Diffusion [19]. Based on these foundations, this study develops the AST/CFG-derived control-flow threat representation and per-token tier map, tier-aware corruption and reconstruction weighting, repair-pair validation, Token-Slot Alignment, and Type A/Type B joint training. Each adaptation is identified separately when it is introduced.

Sections 3.1.1–3.1.6 describe security-aware input construction. Section 3.2 then presents the shared diffusion backbone, TASC, and repair-pair routing, followed by TASR-CD in Section 3.2.4.

3.1 Security-Aware Data Preprocessing

3.1.1 VulGate Dataset

ThreatDiffusion is evaluated on VulGate [13], a C/C++ function-level dataset that preserves vulnerable and repaired functions extracted from security commits. It contains 236,663 samples from 7,600 CVE-related commits across 792 projects and 180 CWE categories, covering May 2002 to May 2025. Nearly 89% of its vulnerable–repaired pairs have cosine similarity of at least 0.90 [13]. Many patches therefore change only a few tokens. This property motivates the structural and tier-aware features introduced in Sections 3.1.3 and 3.1.5.

A representative case is CVE-2021-40568, where a single boundary-check condition is extended to cover negative indices:

$$\texttt{if (pps_id > 255)} \longrightarrow \texttt{if ((pps_id < 0) || (pps_id > 255))}. \tag{3.1}$$

The repaired version introduces validation logic for negative indices, preventing out-of-bounds memory access, yet the two versions share a nearly identical token sequence.

The experiments use the chronological 80/10/10 split, which has an approximately balanced class distribution. Tables 3.1 and 3.2 summarize the VulGate split and dataset statistics.

Table 3.1: Dataset split used in ThreatDiffusion.

Split	**Total**	**Vulnerable**	**Safe**
Train	189,330	94,035	95,295
Validation	23,666	11,671	11,995
Test	23,667	11,726	11,941
Total	236,663	117,432	119,231

Table 3.2: VulGate dataset statistics.

Parameter	**Value**	**Description**
Total samples	236,663	Aggregated from 792 projects and 7,600 CVE commits covering 180 CWE categories
Class ratio	≈ 49.6 / 50.4	Approximately balanced across the three partitions to limit majority-class bias
Hard negatives	≈89%	Vulnerable-repaired pairs with cosine similarity ≥0.90
Data split	80 / 10 / 10	Chronological split. Test commits postdate all training commits

Table 3.3 lists the fields used by the pipeline. The before/after functions provide repair pairs. Line and patch annotations locate edits. The `diff_func` field supports repair-pair validation. The vulnerability label provides detection supervision. CWE labels are used by the contrastive objective, while severity is retained as CVE metadata. Token tiers are derived separately from the AST/CFG analysis.

TAIPEI TECH

Table 3.3: Representative fields extracted from VulGate samples.

Field	Type	Role in ThreatDiffusion
`function_before`	string	Vulnerable source code x_{vuln} and primary model input for diffusion training
`function_after`	string	Repaired target x_{safe} used as the repair supervision target
`target`	binary int	Vulnerability label used for $\mathcal{L}_{\text{bce}}$
`flaw_line_index`	int list	Vulnerable line indices for `flaw_token_mask` construction
`flaw_lines`	string list	Fallback vulnerability spans when line indices are unavailable
`patch_line_index`	int list	Patched code spans used for Token-Slot Alignment
`diff_func`	string	Git diff determining structural validity of repair patch
`cwe`	string	CWE identifier used for contrastive sample grouping
`severity`	float	CVE severity metadata retained for valid scores in $[0, 10]$

3.1.2 BPE Tokenization

ThreatDiffusion uses the BPE tokenizer distributed with UniXcoder [50, 60]. Its subword vocabulary represents language constructs, API names, and project-specific identifiers without introducing a separate tokenizer.

After BPE tokenization, each function is represented as $x = (x^{(1)}, \ldots, x^{(L)}) \in \mathcal{V}^L$, where $\mathcal{V}$ is the UniXcoder vocabulary, $V = |\mathcal{V}| = 51{,}416$, and $L = 1{,}024$ is the fixed sequence length after truncation and padding. ThreatDiffusion reuses the pre-trained UniXcoder embedding module, comprising word, position, and token-type embeddings followed by layer normalization:

$$E_{\text{token}} = \text{Embed}_{\text{UniX}}(x) \in \mathbb{R}^{L \times 768}, \tag{3.2}$$

where $\text{Embed}_{\text{UniX}}(\cdot)$ denotes only the UniXcoder embedding module and $E^{(i)}_{\text{token}} \in \mathbb{R}^{768}$ is the embedding at position i. No UniXcoder Transformer layer is included in this operation. Consequently, E_{token} is a non-contextual lexical and positional representation. Interactions between

tokens are modeled by the 12-layer Transformer introduced in Section 3.2.1. The embedding parameters are updated during TASC pre-training and frozen during TASR-CD fine-tuning, following the training configuration used in the implementation.

The matrix E_{token} forms the lexical stream used in Section 3.1.6.

3.1.3 AST-CFG Parser

ThreatDiffusion uses tree-sitter [61] to parse each function and derives intra-procedural control-flow features from the resulting AST. The preprocessing pipeline records API calls, enclosing guard and loop constructs, and local taint-related cues around security-relevant regions. These extracted AST/CFG indicators complement the lexical stream with an approximate view of control-flow context. Figure 3.2 illustrates the resulting analysis.

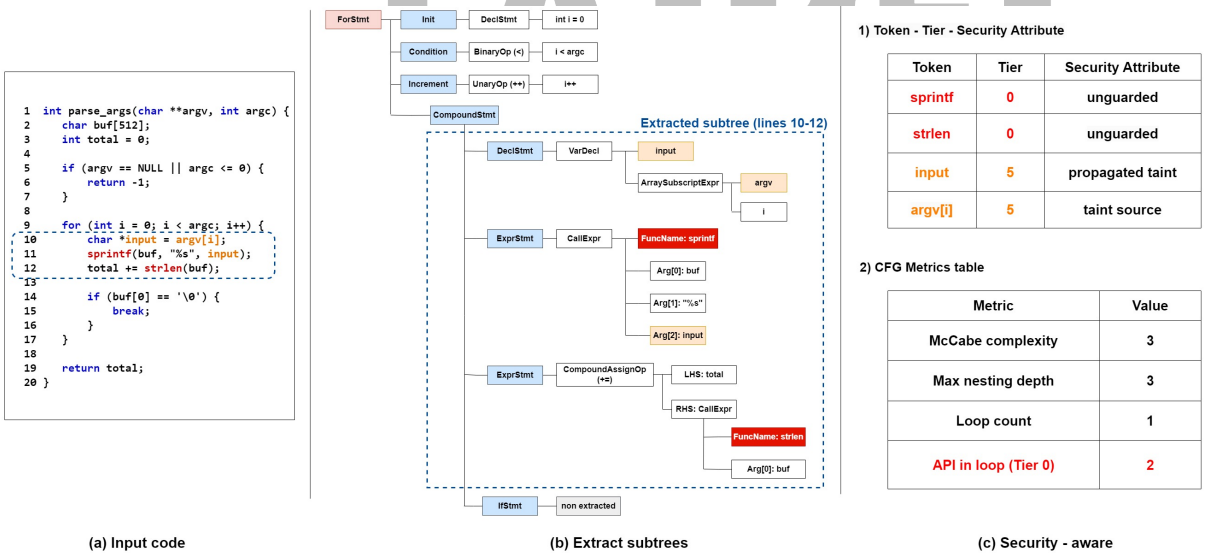


Figure 3.2: Security-Aware AST subtree extraction example.

In Figure 3.2, the selected code region (left) is mapped to its AST subtrees (center), from which the token-level tiers and function-level control-flow features are derived (right). The traversal retains `call_expression` nodes and their enclosing conditional or loop syntax while excluding statements outside the selected region. These features summarize the analyzed syntax and do not claim to recover every feasible runtime path.

Extracted API names are matched against an inventory derived from SEI CERT C/C++ rules [20]. ThreatDiffusion, rather than CERT, defines the numerical Tier 0/1/2/5 labels. If

multiple rules apply, the most severe assignment is kept. In Figure 3.2, `sprintf` and `strlen` are Tier 0 sink-side risks on an unguarded path. The source token `input` remains Tier 5 in this map, while its taint role is stored separately in S_{taint}.

An intra-procedural CFG is constructed from AST control nodes such as `for`, `if`, and `while` statements. The resulting graph provides local branch and loop context. Within this representation, the guard feature records whether a dangerous API argument has a recognized validation check in the analyzed local context. The guard condition is defined as:

$$\text{hasGuard}(\texttt{api}) = \begin{cases} \textit{true}, & \text{if Sanitized}(v,\, L_{\texttt{api}}) \text{ for some } v \in \text{Args}(\texttt{api}) \\ \textit{false}, & \text{otherwise} \end{cases}. \tag{3.3}$$

Here, $\text{Args}(\texttt{api})$ is the API argument set, and $\text{Sanitized}(v, L_{\texttt{api}})$ is true when v or an alias has a recognized check in the analyzed local context before the call. A guarded Tier 0 API is reduced to Tier 1 in Section 3.1.5. Loop-related syntax additionally supplies a loop-context indicator for calls.

3.1.4 Threat Vector

The extracted API, AST/CFG-derived control-flow, and taint-related results are aggregated into a 64-dimensional threat feature vector:

$$S = \left[S_{\text{API}} \parallel S_{\text{CFG}} \parallel S_{\text{taint}} \parallel S_{\text{ctx}}\right] \in \mathbb{R}^{64}, \tag{3.4}$$

where $S_{\text{API}} \in \mathbb{R}^{20}$ contains normalized API counts, loop participation, and guard coverage. $S_{\text{CFG}} \in \mathbb{R}^{15}$ contains normalized graph-structure statistics, including node, edge, and cyclomatic-complexity measures. $S_{\text{taint}} \in \mathbb{R}^{15}$ records taint sources, sanitized sources, and length aliases. $S_{\text{ctx}} \in \mathbb{R}^{14}$ records use-after-free, double-free, and unchecked arithmetic indicators. Unused coordinates within these groups are reserved as zero-valued extension slots. Thus, S is a function-level summary and has no sequence dimension.

A learned affine projection converts this summary to a 64-dimensional representation and broadcasts it across the token sequence:

$$E_{\text{threat}}^{(i)} = W_{\text{th}} S + b_{\text{th}} \in \mathbb{R}^{64}, \qquad i \in \{1, \ldots, L\}, \tag{3.5}$$

where $W_{\text{th}} \in \mathbb{R}^{64\times64}$ and $b_{\text{th}} \in \mathbb{R}^{64}$ are trainable parameters. Stacking the identical projected vector for all positions gives $E_{\text{threat}} \in \mathbb{R}^{L\times64}$, which forms the structural security feature stream used in Section 3.1.6.

3.1.5 Per-Token Tier Map

ThreatDiffusion assigns each token a tier $\tau^{(i)} \in \{0, 1, 2, 5\}$ from its API or arithmetic role and guard context. Tier 0 is the highest operational risk, Tiers 1 and 2 represent weaker risks, and Tier 5 is the background index. Indices 3 and 4 remain unused only for compatibility with the preprocessing index space.

The sets $\mathcal{A}_0$, $\mathcal{A}_1$, and $\mathcal{A}_2$ contain critical, conditional, and medium-risk APIs derived from SEI CERT guidance [20]. The set $\mathcal{O}_{\text{arith}}$ contains arithmetic operators that lack range validation. Tokens outside these sink and operator sets receive Tier 5. They may still carry source-side taint evidence through S_{taint}.

Let r_i denote the intrinsic risk level of token i:

$$r_i = \begin{cases} \text{high}, & \text{if } x^{(i)} \in \mathcal{A}_0 \\ \text{medium}, & \text{if } x^{(i)} \in \mathcal{A}_1 \ \vee \ x^{(i)} \in \mathcal{O}_{\text{arith}} \\ \text{low}, & \text{if } x^{(i)} \in \mathcal{A}_2 \\ \text{benign}, & \text{otherwise} \end{cases}. \tag{3.6}$$

Let $g_i \in \{0, 1\}$ denote the guard indicator for the API associated with token i, where $g_i = 1$ if $\text{hasGuard}(\text{api}_i) = \textit{true}$ as defined in Equation 3.3. For tokens that do not denote an API call, $g_i = 0$. The final tier assignment is:

$$\tau^{(i)} = \begin{cases} 0, & \text{if } r_i = \text{high} \ \wedge \ g_i = 0 \\ 1, & \text{if } (r_i = \text{high} \ \wedge \ g_i = 1) \ \vee \ (r_i = \text{medium} \ \wedge \ g_i = 0) \\ 2, & \text{if } (r_i = \text{medium} \ \wedge \ g_i = 1) \ \vee \ r_i = \text{low} \\ 5, & \text{otherwise} \end{cases}. \tag{3.7}$$

Thus, a guard reduces but does not erase API risk: a guarded Tier 0 API becomes Tier 1, while an unguarded arithmetic operator is assigned Tier 1 rather than the direct memory-corruption category Tier 0.

Each token's tier label is encoded through a trainable embedding lookup:

$$E_{\text{tier}}^{(i)} = \text{Embed}_{\text{tier}}\big(\tau^{(i)}\big) \in \mathbb{R}^{64}, \tag{3.8}$$

where $\text{Embed}_{\text{tier}} : \{0, 1, 2, 5\} \to \mathbb{R}^{64}$ is a trainable embedding table initialized with a vocabulary size of 6 to support the index space $\{0, 1, 2, 3, 4, 5\}$, of which only $\{0, 1, 2, 5\}$ correspond to active tier labels. The resulting $E_{\text{tier}} \in \mathbb{R}^{L\times 64}$ provides the per-token security tier feature stream used by the unified input construction in Section 3.1.6.

3.1.6 Input Representation

The input combines $E_{\text{token}}(x_t) \in \mathbb{R}^{L\times 768}$, $E_{\text{tier}} \in \mathbb{R}^{L\times 64}$, and $E_{\text{threat}} \in \mathbb{R}^{L\times 64}$. Only the token stream changes with the corrupted state x_t. The tier and threat streams remain fixed for the source function.

The timestep embedding $E_{\text{time}} \in \mathbb{R}^{L\times 64}$ encodes the current diffusion state and is introduced alongside the backbone architecture in Section 3.2.1. The four streams are fused through a trainable projection layer to produce the unified model input:

$$E_{\text{input}} = \text{LN}\big(\text{GELU}\big([E_{\text{token}}(x_t)\,;\, E_{\text{tier}}\,;\, E_{\text{threat}}\,;\, E_{\text{time}}]\, W_{\text{fuse}}^{\mathsf{T}} + 1_L b_{\text{fuse}}^{\mathsf{T}}\big)\big) \in \mathbb{R}^{L\times 960}, \tag{3.9}$$

where $[\cdot\,;\,\cdot]$ denotes feature-wise concatenation, giving $768+64+64+64 = 960$ channels. $W_{\text{fuse}} \in \mathbb{R}^{960\times960}$ and $b_{\text{fuse}} \in \mathbb{R}^{960}$ are the trainable affine-projection parameters, and $1_L \in \mathbb{R}^L$ broadcasts the bias over the sequence. GELU is the activation function and LN denotes Layer Normalization. The resulting E_{input} is passed to the Pre-LN Transformer in Section 3.2.1.

3.2 Tier-Aware Masked Diffusion

The fused representation $E_{\text{input}} \in \mathbb{R}^{L\times960}$ is processed as a masked-generation problem by a shared Pre-LN Transformer. TASC first learns tier-aware denoising and edit behavior. Repair-pair validation then routes reliable pairs to repair supervision. TASR-CD finally fine-tunes the same backbone for aligned repair and calibrated detection. Together, these stages address variable-length edits and the vulnerable-label bias of repair pairs.

3.2.1 Shared Transformer Backbone

TASC and TASR-CD use the same Transformer, and the selected TASC checkpoint initializes TASR-CD without changing the backbone structure. Their supervision differs only through the routing and objectives in Section 3.2.4.

The implemented graph contains 224,262,383 parameters, including 40,284,672 in the UniXcoder embedding module. This module is trainable during TASC and frozen during TASR-CD, leaving 183,977,711 trainable parameters. The unused UniXcoder encoder layers are not part of ThreatDiffusion.

- **Timestep Embedding.**

  ThreatDiffusion conditions the backbone on $t \in [0, 1]$ using sinusoidal timestep features followed by a two-layer MLP [53, 51]:

$$e_{\text{time}} = \text{MLP}(\text{SinEmb}(t)) \in \mathbb{R}^{64}, \tag{3.10}$$

  Here, $\text{SinEmb}(t) \in \mathbb{R}^{64}$ contains 32 sine and 32 cosine components. The MLP maps $64 \rightarrow$

$256 \to 64$ with GELU. Its output is broadcast over the sequence as $E_{\text{time}} = 1_L e_{\text{time}}^{\mathsf{T}} \in \mathbb{R}^{L\times 64}$. The base schedule maps $t = 0$ to a clean sequence and $t = 1$ to unit masking. After Equation 3.16 is applied, Tier 5 positions may still retain context near $t = 1$.

- **Pre-LN Transformer Backbone.**

  The shared backbone contains 12 Pre-LN Transformer layers [59, 62]. For $h_0 = E_{\text{input}}$, layer ℓ is defined as:

$$\begin{aligned} u_\ell &= h_{\ell-1} + \text{Attn}(\text{LN}(h_{\ell-1}))\,, \\ h_\ell &= u_\ell + \text{FFN}(\text{LN}(u_\ell))\,, \qquad \ell = 1, \ldots, 12, \end{aligned} \tag{3.11}$$

  Here, $h_0 = E_{\text{input}} \in \mathbb{R}^{L\times 960}$ is the initial input. The variable u_ℓ is the intermediate state after the attention residual at layer ℓ. The function $\text{Attn}(\cdot)$ denotes 12-head self-attention. The function $\text{FFN}(\cdot)$ denotes a two-layer feed-forward network with hidden dimension 3,840 and GELU activation. The final shared hidden representation is $H = h_{12} \in \mathbb{R}^{L\times 960}$. All prediction heads use H. Type A/Type B routing and the weighted objectives control their unequal supervision.

- **Masked Diffusion Forward and Reverse Process.**

  Following D3PM and MDLM [51, 16], ThreatDiffusion uses an absorbing `[MASK]` state. Chapter 2 reviews this process. The formulation is restated here only to define the notation used by the tier-aware adaptation.

  Let $x_0 \in \mathcal{V}^L$ denote a clean sequence and x_t its corrupted state at time $t \in [0, 1]$. For the standard, position-independent process reviewed in Chapter 2, each token is masked according to

$$q\left(x_t^{(i)} \mid x_0^{(i)}\right) = \begin{cases} 1 - \gamma_t, & x_t^{(i)} = x_0^{(i)}, \\ \gamma_t, & x_t^{(i)} = m \end{cases}, \tag{3.12}$$

  where $q(\cdot \mid \cdot)$ denotes the forward corruption kernel, $i \in \{1, \ldots, L\}$ is the token index, m is the absorbing `[MASK]` state, and $\gamma_t \in [0, 1]$ is a monotonically increasing corruption

probability with $\gamma_0 = 0$ and $\gamma_1 = 1$. ThreatDiffusion instantiates the base schedule as $\gamma_t = 1 - \cos(\pi t/2)$. This equation defines the MDLM foundation. Phase 1 replaces its uniform probability with the tier-scaled probability in Equation 3.16.

For $0 \le s \le t$ with $t > 0$, MDLM's SUBS parameterization leaves an observed token unchanged and predicts only from a masked state [16]:

$$p_\theta\big(x_s^{(i)} \mid x_t\big) = \begin{cases} \mathrm{Cat}\Big(x_s^{(i)}; e_{x_t^{(i)}}\Big), & x_t^{(i)} \neq m, \\ \mathrm{Cat}\left(x_s^{(i)}; \dfrac{(\gamma_t - \gamma_s)\mu_\theta^{(i)}(x_t, t) + \gamma_s e_m}{\gamma_t}\right), & x_t^{(i)} = m \end{cases}, \tag{3.13}$$

Here, p_θ is the reverse transition with trainable parameters θ. The expression $\mathrm{Cat}(x; \pi)$ evaluates a categorical distribution with probability vector π. The term $\mu_\theta^{(i)}(x_t, t) \in \Delta^{V-1}$ is the predicted clean-token distribution over a V-token vocabulary. The term e_a is the one-hot vector for token a. The masked coefficients sum to one, reducing to $\mu_\theta^{(i)}$ at $s = 0$ and e_m at $s = t$. This SUBS transition is adopted from MDLM.

Using the same corruption-probability notation, the continuous-time MDLM objective is

$$\mathcal{L}_{\mathrm{MDLM}}(\theta) = \mathbb{E}_{\substack{t\sim\mathcal{U}[0,1] \\ x_t\sim q(\cdot|x_0)}} \left[\frac{\gamma_t'}{\gamma_t} \sum_{i:\, x_t^{(i)} = m} -\log \mu_\theta^{(i)}(x_t, t)\big[x_0^{(i)}\big] \right], \tag{3.14}$$

where $\gamma_t' = d\gamma_t/dt$. In the retention notation of Equation 2.4, $\alpha_t = 1 - \gamma_t$ and $\gamma_t'/\gamma_t = -\alpha_t'/(1-\alpha_t)$. Thus, the change from α_t in Chapter 2 to γ_t in this chapter is only a change of notation. The tier-scaled masking and additional reconstruction weights introduced next constitute the methodological adaptation.

3.2.2 Tier-Aware Security Curriculum

TASC trains the shared Transformer in two phases. Its mask-to-edit sequence is inspired by Seed Diffusion [19], whereas the security-tier conditioning is introduced in ThreatDiffusion. As shown in Figure 3.3, Phase 1 applies tier-scaled absorbing-state masking [16], and Phase 2 shifts the learned representation toward structured edits at security-critical positions. The fully

masked $t = 1$ row is schematic: the implemented schedule remains position-dependent under Equation 3.16, so lower-risk tokens may retain context near the endpoint.

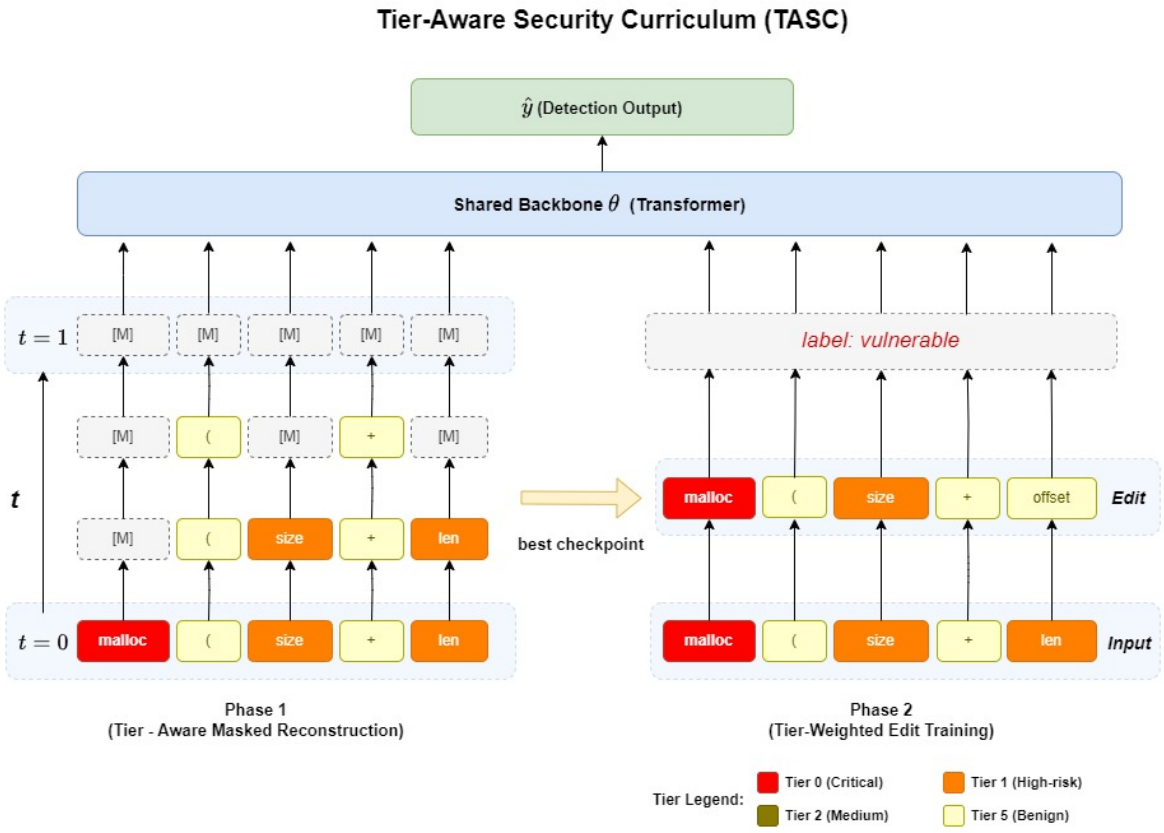


Figure 3.3: Two-phase structure of the Tier-Aware Security Curriculum (TASC).

- **Detection Head.**

  A Detection Head maps the shared sequence representation to a function-level vulnerability prediction. Let $\mathcal{I} = \{i : a^{(i)} = 1\}$ be the non-padding positions, where $a^{(i)}$ is the attention mask. The implementation mean-pools these positions and applies a two-layer MLP:

$$\begin{aligned} \bar{h} &= \frac{1}{|\mathcal{I}|} \sum_{i \in \mathcal{I}} H^{(i)}, \\ \hat{y} &= w_2^{\mathsf{T}} \mathrm{Drop}\big(\mathrm{GELU}\big(W_1 \bar{h} + b_1\big)\big) + b_2, \qquad \hat{p} = \sigma(\hat{y}), \end{aligned} \tag{3.15}$$

  Here, $W_1 \in \mathbb{R}^{256 \times 960}$, $b_1 \in \mathbb{R}^{256}$, $w_2 \in \mathbb{R}^{256}$, and $b_2 \in \mathbb{R}$ are trainable parameters. The function $\mathrm{Drop}(\cdot)$ denotes training-time dropout. The value $\hat{y}$ is the scalar logit, and $\hat{p}$ is the predicted vulnerability probability. The function $\sigma(\cdot)$ denotes the sigmoid function. For

a binary label $y \in \{0, 1\}$, the loss is weighted binary cross-entropy, $\mathcal{L}_{\text{bce}} = -\eta y \log \hat{p} - (1 - y) \log(1 - \hat{p})$, where $\eta > 0$ is the positive-class weight.

- **Phase 1: Tier-Aware Masked Reconstruction.**

  Phase 1 combines non-uniform tier masking with time- and token-dependent reconstruction weights.

  – **Tier-Aware Masking.**

  Phase 1 scales each token's masking probability by a tier-dependent multiplier rather than applying the global noise schedule γ_t uniformly:

$$p_{\text{mask}}^{(i)} = \min\big(\gamma_t \cdot \mu_{\tau^{(i)}},\ 1\big), \qquad \mu_0 > \mu_1 > \mu_2 > \mu_5, \tag{3.16}$$

  where $\tau^{(i)} \in \{0, 1, 2, 5\}$ is the security tier assigned to token i, and $\mu_{\tau^{(i)}}$ is a tier-specific masking multiplier. The reported configuration uses $(\mu_0, \mu_1, \mu_2, \mu_5) = (1.3, 1.1, 1.0, 0.8)$. High-risk tokens are therefore masked more frequently than benign context tokens, requiring the backbone to reconstruct security-critical API regions from surrounding structural evidence.

  – **MDLM Timestep Weight in TASC.**

  The ratio γ_t'/γ_t is the standard MDLM weight written in the corruption-probability notation of this chapter. It is not a modified weighting rule. For $\gamma_t = 1 - \cos(\pi t/2)$, the implementation uses the numerically stabilized and capped form

$$\bar{w}(t) = \min\left(\frac{(\pi/2)\sin(\pi t/2)}{1 - \cos(\pi t/2) + 10^{-8}},\ 10\right). \tag{3.17}$$

  Here, $\bar{w}(t)$ denotes the implemented timestep weight, 10^{-8} prevents division by zero, and the upper bound 10 prevents very small nonzero timesteps from dominating a batch update. Before capping, the MDLM ratio places greater weight on low-noise states. ThreatDiffusion retains this standard time dependence in Phase 1. Its contribution is instead the tier-dependent corruption in Equation 3.16 and the tier, mask, flaw, and focal factors in the following objective.

TAIPEI TECH

– **Phase 1 Training Objective.**

The implementation supervises all valid positions rather than only the masked subset. Let $\mathcal{B}$ denote a mini-batch and $\mathcal{I}_b$ the non-padding positions of sample b. Its normalized Phase 1 objective is

$$\mathcal{L}_{\text{phase1}}(\theta) = \frac{\sum_{b\in\mathcal{B}}\sum_{i\in\mathcal{I}_b} \kappa_{b,i}\,\ell_{b,i}}{\sum_{b\in\mathcal{B}}\sum_{i\in\mathcal{I}_b} \kappa_{b,i}}, \qquad \kappa_{b,i} = \bar{w}(t_b) w_{\text{tier}}^{(b,i)} w_{\text{mask}}^{(b,i)} w_{\text{flaw}}^{(b,i)}, \tag{3.18}$$

Here, $t_b \sim \mathcal{U}[0,1]$ is the timestep of sample b and $\bar{w}(t_b)$ is defined in Equation 3.17. The tier factor is a fixed lookup, $w_{\text{tier}}^{(b,i)} = W_{\text{base}}[\tau^{(b,i)}]$, where $W_{\text{base}} = (2.5, 2.2, 1.8, 0, 0, 1.0)$ for tier indices 0 through 5. The optional learnable tier scaling is disabled in the reported configuration, so this lookup is used directly. $w_{\text{mask}}^{(b,i)}$ equals 2 at positions sampled for masking and 1 otherwise. When an exact `flaw_token_mask` annotation is available, $w_{\text{flaw}}^{(b,i)}$ equals 1.5 on its marked positions and 1 elsewhere.

Let $p_{b,i} = \mu_\theta^{(b,i)}(x_{t_b}^{(b)}, t_b)[x_0^{(b,i)}]$ be the probability assigned to the clean target at position i of sample b, where $x_{t_b}^{(b)}$ is its corrupted sequence, and let $\text{CE}_\epsilon(\mu, x)$ denote cross-entropy with label smoothing $\epsilon = 0.1$. The per-position loss is

$$\ell_{b,i} = \begin{cases} (1-p_{b,i})^{\gamma_f}\,\text{CE}_\epsilon\left(\mu_\theta^{(b,i)}, x_0^{(b,i)}\right), & \tau^{(b,i)} \in \{0,1,2\}, \\ \text{CE}_\epsilon\left(\mu_\theta^{(b,i)}, x_0^{(b,i)}\right), & \tau^{(b,i)} = 5 \end{cases}, \tag{3.19}$$

where $\gamma_f = 2$ is the Focal Loss focusing parameter [57]. Explicitly, $\text{CE}_\epsilon(\mu, x) = -(1-\epsilon)\log\mu[x] - (\epsilon/V)\sum_{v\in\mathcal{V}}\log\mu[v]$. Because Equation 3.18 combines tier-dependent corruption, supervision on unmasked positions, and additional token weights, it is a tier-aware surrogate denoising objective derived from, but not identical to, the standard MDLM likelihood bound in Equation 3.14. A bound derived directly for the non-uniform process would instead require the per-position ratio $(\gamma_t^{(i)})'/\gamma_t^{(i)}$.

Phase 1 retains weak full-sequence supervision while emphasizing security-critical positions.

- **Phase 2: Tier-Weighted Edit Training.**

  Phase 2 continues from Phase 1 with structured edits around security-critical positions.

  – **Edit-Based Perturbation.**

  At each training step, edit positions are sampled from a tier-weighted distribution:

$$p_{\text{edit}}^{(i)} = \frac{w_{\text{edit}}\big(\tau^{(i)}\big)}{\sum_{j\in\mathcal{I}} w_{\text{edit}}\big(\tau^{(j)}\big)}, \qquad i \in \mathcal{I}, \tag{3.20}$$

  where $\mathcal{I}$ denotes the non-padding positions of the current sequence.

  The implemented edit weights are $w_{\text{edit}}(0) = 0.7$, $w_{\text{edit}}(1) = 0.5$, $w_{\text{edit}}(2) = 0.3$, and $w_{\text{edit}}(5) = 0.1$. For $t \sim \mathcal{U}[0,1]$, the number of edit draws is $k_t = \max\{1, \lfloor 0.1Lt \rfloor\}$. The implementation draws these positions from p_{edit} with replacement. A position may therefore receive more than one sequential edit. At each draw, substitution, insertion, or deletion is selected with probability $1/3$. Insertions shift the suffix right and drop the final position, whereas deletions shift the suffix left and append `[PAD]`, so the Phase 2 corruption always preserves length L. This appended value is inactive tail padding for the fixed-length corruption process. It is distinct from the active `[PAD]` slots introduced later by Token-Slot Alignment. This edit corruption follows the mask-to-edit curriculum pattern of Seed Diffusion [19]. The tier-dependent location distribution is the adaptation introduced in TASC.

  – **Phase 2 Training Objective.**

  The Phase 2 loss applies normalized tier-weighted cross-entropy over valid positions and jointly preserves function-level detection:

$$\mathcal{L}_{\text{phase2}}(\theta) = \frac{\sum_{b\in\mathcal{B}}\sum_{i\in\mathcal{I}_b} w_{\text{tier}}^{(b,i)} w_{\text{edit-pos}}^{(b,i)} w_{\text{flaw}}^{(b,i)} \text{CE}\Big(\mu_\theta^{(b,i)}, x_0^{(b,i)}\Big)}{\sum_{b\in\mathcal{B}}\sum_{i\in\mathcal{I}_b} w_{\text{tier}}^{(b,i)} w_{\text{edit-pos}}^{(b,i)} w_{\text{flaw}}^{(b,i)}} + \lambda_{\text{bce}}\mathcal{L}_{\text{bce}}. \tag{3.21}$$

  Here, $\mathcal{B}$ is the mini-batch and $\mathcal{I}_b$ contains the non-padding positions of sample b.

The factor $w_{\text{edit-pos}}^{(b,i)}$ equals 2 at a sampled edit position and 1 otherwise. The factor $w_{\text{flaw}}^{(b,i)}$ uses the same optional 1.5 annotation boost as Phase 1. The factor $w_{\text{tier}}^{(b,i)}$ is the same tier lookup used in Equation 3.18. $\text{CE}(\mu_\theta^{(b,i)}, x_0^{(b,i)}) = -\log \mu_\theta^{(b,i)}[x_0^{(b,i)}]$, and $\lambda_{\text{bce}} = 0.20$ controls the contribution of the detection loss. In this stage, the positive-class weight in $\mathcal{L}_{\text{bce}}$ is $\eta = 1.5$. The denominator stabilizes the scale across sequences with different edit patterns.

Phase 2 refines the Phase 1 backbone toward edit-aware repair behavior and prepares the representation for span-level fine-tuning.

3.2.3 Repair-Pair Validation and Type Routing

VulGate contains 9,118 paired `diff_normal` training samples, but automated extraction may yield truncated bodies, duplicated signatures, identical pairs, or non-code outputs. Such samples may retain valid detection labels but cannot provide reliable repair targets.

Repair-Pair Validation therefore separates samples according to whether their repair targets can support Token-Slot Alignment. The clean repair set $\mathcal{D}_{\text{clean}}$ is defined over the repair-pair subset $\mathcal{D}_{\text{diff}}$ as:

$$\mathcal{D}_{\text{clean}} = \left\{ (x_{\text{before}},\, x_{\text{after}}) \in \mathcal{D}_{\text{diff}} \;\middle|\; \neg r_{\text{trunc}} \wedge \neg r_{\text{dup}} \wedge \neg r_{\text{same}} \wedge \neg r_{\text{noncode}} \right\}, \tag{3.22}$$

where $\mathcal{D}_{\text{diff}}$ denotes the complete set of `diff_normal` training samples, and r_{trunc}, r_{dup}, r_{same}, r_{noncode} are Boolean rejection flags. The validation rules are listed in Table 3.4.

Table 3.4: Validation rules for corrupted repair pairs.

Rule	Description
r_{trunc}	`function_after` contains fewer than 20% of the lines in `function_before` and lacks a closing brace, indicating that only a partial function body was extracted.
r_{dup}	`function_after` opens with a duplicated function signature, a known artifact of repeated function headers in diff context extraction.
r_{same}	`function_before` and `function_after` are character-identical despite the sample being labeled vulnerable, providing no valid repair signal.
r_{noncode}	`function_after` contains no braces, semicolons, or C-language keywords, indicating non-code content was extracted rather than source code.

Applying these conditions excludes 1,036 pairs (≈11.4%) from repair supervision and retains $|\mathcal{D}_{\text{clean}}| = 8{,}082$ clean repair pairs. Table 3.5 reports the validation statistics. All retained repair pairs subsequently pass Token-Slot Alignment during span construction.

Table 3.5: Repair-pair validation statistics.

Category	Excluded	Remaining
Original `diff_normal` samples	—	9,118
r_{trunc}: truncated_after	312	—
r_{dup}: duplicate_signature	198	—
r_{same}: identical_vuln_safe	401	—
r_{noncode}: non_code_after	125	—
After validation	1,036	8,082

TAIPEI TECH

The validated repair pairs are designated as Type A samples and receive full repair supervision. The excluded repair pairs, together with all non-repair training samples, form Type B (181,248 samples total). Type B samples retain valid vulnerability labels for detection training but contribute no repair loss. This routing prevents corrupted repair targets from affecting span generation while preserving their detection signal for the detection-calibration path in TASR-CD. The contrastive objective remains confined to Type A vulnerable–repair pairs.

3.2.4 Tier-Aware Span Repair with Contrastive Detection

TASR-CD loads the TASC checkpoint and uses the Type A/Type B routing from Section 3.2.3. Token-Slot Alignment converts variable-length patches into fixed-coordinate targets, edit heads model their structure, and the Type B path preserves detection calibration while Type A provides repair supervision.

- **Token-Slot Alignment.**

  Variable-length insertions, deletions, and replacements do not directly match the Transformer's positionwise output. Token-Slot Alignment resolves this mismatch by assigning fixed coordinates to edit spans. Figure 3.4 shows a CWE-377 example, following the edit visualization of Edit Flow [63]: the vulnerable call `fopen(file, "w")` becomes `open(file, O_CREAT, 0644)`. Here, `[PAD]` denotes an active empty slot for insertion. It keeps a valid attention entry and receives an operation label and a token target. Inactive tail padding uses the same symbol but is excluded by the attention mask. In contrast, `[MASK]` is used only for diffusion corruption.

TAIPEI TECH

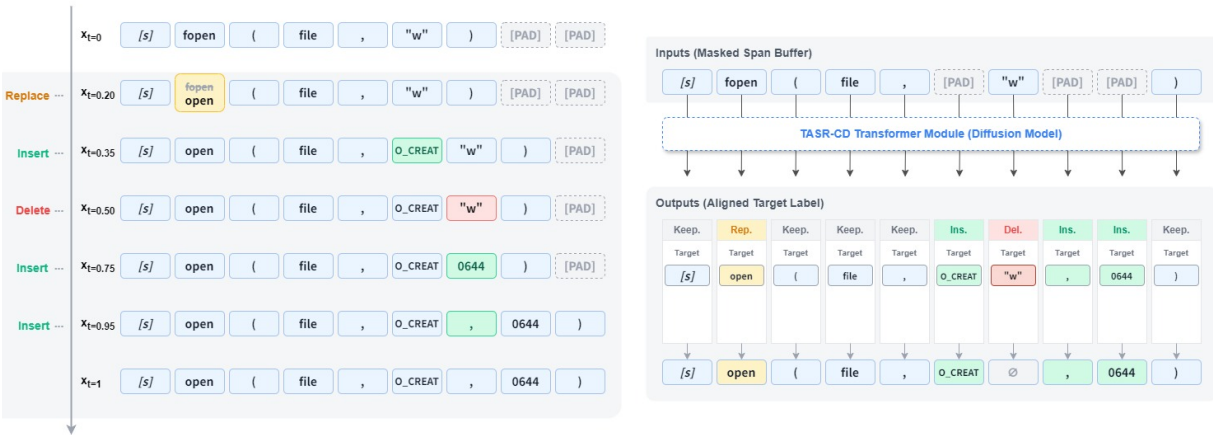


Figure 3.4: Illustration of the Token-Slot Alignment mechanism and the TASR-CD transformer's end-to-end prediction for an insecure file creation vulnerability patch.

Given x_{vuln} and x_{safe}, a `difflib`-based procedure extracts contiguous `equal`, `replace`, `insert`, and `delete` spans. Equal spans are copied. Replacement spans retain available vulnerable tokens and use active `[PAD]` slots when the repaired side is longer. Insertion spans use active `[PAD]` slots throughout. Deletion spans retain the source tokens at active positions and are marked by the delete operation label. The separate before-sequence deletion labels provide complementary source-coordinate supervision. Unused tail capacity is inactive tail padding. The resulting buffer preserves aligned tokens, edit types, and the distinction between an empty insertion slot and a deleted source token.

For an edit span S_k, the allocated buffer capacity C_k is determined by the edit operation:

$$C_k = \begin{cases} |S_k^{\text{vuln}}|, & \text{if } o_k = \texttt{equal}, \\ \min\left(\max\left(|S_k^{\text{vuln}}|, |S_k^{\text{safe}}|\right), K_{\max}\right), & \text{if } o_k = \texttt{replace}, \\ \min\left(|S_k^{\text{safe}}|, K_{\max}\right), & \text{if } o_k = \texttt{insert}, \\ \min\left(|S_k^{\text{vuln}}|, K_{\max}\right), & \text{if } o_k = \texttt{delete} \end{cases}, \tag{3.23}$$

Here, S_k^{vuln} and S_k^{safe} are the vulnerable- and safe-side token subsequences of span k. The notation $|\cdot|$ denotes token count. The variable $o_k \in \{\texttt{equal}, \texttt{replace}, \texttt{insert}, \texttt{delete}\}$ denotes the diff operation. The value $K_{\max} = 16$ is the maximum number of aligned positions allocated to one edit span. For replacement, the capacity is large enough to

retain the available vulnerable tokens and safe-side targets, subject to this cap. A deletion therefore retains its source tokens in the aligned buffer, but its repair-token loss is disabled and its edit decision is supervised by the operation and Deletion Heads. The corresponding source indices are also recorded in `del_label` $\in \{0,1\}^L$ in the original before-sequence coordinate.

At aligned position i, $x^{(i)}_{\text{span_safe}}$ is the repair target for keep, replacement, and insertion positions. A deletion position retains its vulnerable token as an alignment reference, but it is removed by the predicted delete operation and is excluded from the repair-token loss. Replacement inputs retain a vulnerable token when available and otherwise use active `[PAD]`. Insertion inputs use active `[PAD]` throughout. Their attention value is one, whereas inactive tail padding has attention value zero. Deletions remain in the original before-sequence coordinate.

Token-Slot Alignment is constructed for the 8,082 pairs that passed Repair-Pair Validation. These samples constitute Type A. The K_{max} cap truncates an individual aligned edit span but does not trigger an additional Type A-to-Type B reclassification in the implemented pipeline. Type B is therefore the mutually exclusive set defined in Section 3.2.3.

- **Detection Module.**

  Type A repair samples all carry $y = 1$. Repeated joint updates on these samples may therefore bias the detector toward vulnerable predictions. This potential high-recall shift is termed Recall Drift in this work. ThreatDiffusion uses Type A for joint repair–detection learning and Type B, with its original labels, for detection-only calibration.

- **Multi-Task Prediction Heads.**

  Five heads operate on $H \in \mathbb{R}^{L\times 960}$. Repair and detection reuse the pre-training heads. The operation type, slot count, and deletion heads provide edit-structure supervision.

  1. **Repair Head.**

     The Repair Head projects each contextual state over the UniXcoder vocabulary:

$$
\begin{aligned}
Z_{\text{rep}} &= HW_{\text{rep}}^{\mathsf{T}} + 1_L b_{\text{rep}}^{\mathsf{T}}, \\
\widehat{P}_{\text{safe}} &= \text{softmax}(Z_{\text{rep}}) \in \mathbb{R}^{L \times V}.
\end{aligned}
\tag{3.24}
$$

Here, $W_{\text{rep}} \in \mathbb{R}^{V \times 960}$ and $b_{\text{rep}} \in \mathbb{R}^{V}$ are trainable, Z_{rep} contains vocabulary logits, and $\widehat{P}_{\text{safe}}^{(i,v)}$ is the probability of token v at aligned position i. The `[MASK]` logit is suppressed before softmax so that a mask cannot be emitted as a repair token.

The position-weighted repair loss is

$$
\mathcal{L}_{\text{repair}} = \frac{\displaystyle\sum_{i=1}^{L_{\text{span}}} \rho^{(i)} \left[-\log \widehat{P}_{\text{safe}}^{\left(i,\, x_{\text{span_safe}}^{(i)}\right)} \right]}{\displaystyle\sum_{i=1}^{L_{\text{span}}} \rho^{(i)}},
\tag{3.25}
$$

If all active aligned positions are deletion positions, the denominator is zero and $\mathcal{L}_{\text{repair}}$ is set to zero for that sample. Here, L_{span} is the padded aligned-sequence length, $x_{\text{span_safe}}^{(i)}$ is the target token, and

$$
\rho^{(i)} = \begin{cases}
0, & a_{\text{span}}^{(i)} = 0 \text{ or } i \in \mathcal{B}_{\text{del}}, \\
\rho_{\text{span}}, & i \in \mathcal{B}_{\text{span}}, \\
\rho_{\text{noise}}, & i \in \mathcal{B}_{\text{noise}} \setminus \mathcal{B}_{\text{span}}, \\
\rho_{\text{ctx}}, & \text{otherwise.}
\end{cases}
\tag{3.26}
$$

The variable $a_{\text{span}}^{(i)}$ is the aligned attention mask. The set $\mathcal{B}_{\text{span}}$ contains the valid positions selected by the repair-span mask, including replacement and insertion positions and any retained keep positions within the aligned edit region. The set $\mathcal{B}_{\text{del}}$ contains retained deletion positions. The set $\mathcal{B}_{\text{noise}}$ contains the additional positions masked by the repair-noise process. The aligned attention mask keeps active `[PAD]` insertion slots in the loss while excluding inactive tail padding. Retained deletion positions remain available to the operation and deletion heads but receive zero repair-token weight. The implementation uses $\rho_{\text{span}} = 5$, $\rho_{\text{noise}} = 1$, and $\rho_{\text{ctx}} = 0.1$, concentrat-

TAIPEI TECH

ing the principal repair gradient on the aligned edit while retaining weak contextual supervision.

2. **Detection Head.**

   The Detection Head retains the $960 \to 256 \to 1$ MLP in Equation 3.15. During TASR-CD, Type A samples optimize $\mathcal{L}_{\text{bce}}$ together with the repair objectives, whereas Type B samples optimize only $\mathcal{L}_{\text{bce}}$. The head architecture is shared. Only the losses activated by each sample type differ.

3. **Operation Type Head.**

   A $960 \to 256 \to 4$ MLP with GELU and dropout produces $P_{\text{op}}^{(i)} \in \Delta^3$ over $\{\texttt{equal}, \texttt{replace}, \texttt{insert}, \texttt{delete}\}$. Let $\mathcal{I}_{\text{align}} = \{i : a_{\text{span}}^{(i)} = 1\}$ and let $o^{(i)}$ be the operation label generated by `difflib.SequenceMatcher`. The loss is

$$\mathcal{L}_{\text{op}} = -\frac{1}{|\mathcal{I}_{\text{align}}|} \sum_{i \in \mathcal{I}_{\text{align}}} \log P_{\text{op}}^{(i,o^{(i)})}. \tag{3.27}$$

   Deletion positions retain their vulnerable tokens and therefore provide positive `delete` targets in the aligned coordinate. The Deletion Head supplies complementary before-sequence supervision, so the two heads describe the same edit from aligned and source-coordinate perspectives.

4. **Slot Count Head.**

   A second $960 \to 256 \to 17$ MLP predicts $P_{\text{slot}}^{(i)}$ over counts $\{0, \ldots, K_{\max}\}$, where $K_{\max} = 16$. Let $\mathcal{I}_{\text{ins}} = \{i : o^{(i)} = \texttt{insert}\}$ and let $c^{(i)}$ be the number of slots assigned to that insertion. The loss is

$$\mathcal{L}_{\text{slot}} = -\frac{1}{|\mathcal{I}_{\text{ins}}|} \sum_{i \in \mathcal{I}_{\text{ins}}} \log P_{\text{slot}}^{(i,c^{(i)})}, \tag{3.28}$$

   and is set to zero when $\mathcal{I}_{\text{ins}} = \varnothing$. The head therefore models insertion length up to the explicit 16-slot cap rather than assuming unbounded output length.

5. **Deletion Head.**

   The Deletion Head operates in the original vulnerable before-sequence coordinate

and predicts

$$\hat{p}_{\text{del}}^{(i)} = \sigma\left(w_{\text{del}}^{\mathsf{T}} H_{\text{vuln}}^{(i)} + b_{\text{del}}\right) \in [0,1], \qquad i \in \{1,\ldots,L\}, \tag{3.29}$$

Here, $w_{\text{del}} \in \mathbb{R}^{960}$ and $b_{\text{del}} \in \mathbb{R}$ are trainable. The term $H_{\text{vuln}}^{(i)}$ is the hidden state at source position i. The label $y_{\text{del}}^{(i)} = 1$ means that the source token is absent from x_{safe}. With $\mathcal{I}_{\text{vuln}}$ denoting non-padding source positions,

$$\mathcal{L}_{\text{del}} = -\frac{1}{|\mathcal{I}_{\text{vuln}}|} \sum_{i \in \mathcal{I}_{\text{vuln}}} \left[y_{\text{del}}^{(i)} \log \hat{p}_{\text{del}}^{(i)} + (1 - y_{\text{del}}^{(i)}) \log(1 - \hat{p}_{\text{del}}^{(i)}) \right]. \tag{3.30}$$

This source-coordinate prediction complements the aligned delete operation. The retained source token remains visible to the operation and deletion heads, while the final edit application removes it from the repaired sequence.

- **Training Objectives and Optimization.**
  - **Noise Schedule for Repair.**

    During span repair, each sample draws $t \sim \mathcal{U}[t_{\text{min}}, t_{\text{max}}]$ with $t_{\text{min}} = 0.05$ and $t_{\text{max}} = 0.30$. Positions in $\mathcal{B}_{\text{span}} \cup \mathcal{B}_{\text{del}}$ retain the aligned inputs defined by Token-Slot Alignment. A replacement slot contains its vulnerable-side token when available, an insertion slot contains an active `[PAD]` value, and a deletion slot contains its vulnerable-side token. These edit slots receive no additional stochastic masking so that their alignment semantics are preserved. Inactive tail padding is also excluded. For every remaining valid position i, the masking probability is $\min(\gamma_t \nu_{\tau^{(i)}}, 1)$ with $(\nu_0, \nu_1, \nu_2, \nu_5) = (3.0, 2.0, 1.2, 0.3)$. The truncated range and lower Tier 5 multiplier preserve enough context for aligned repair, while the larger Tier 0 and Tier 1 multipliers expose the model more often to uncertain security-critical context.

  - **Contrastive and Multi-Task Objectives.**

    For Type A samples, this work introduces a CWE-conditioned contrastive objective to improve the separation between vulnerable and repaired code representations. The objective adopts the normalized similarity and temperature scaling of

InfoNCE [23], while its use of multiple same-class positives is motivated by supervised contrastive learning [24]. It is therefore an adapted objective for the paired repair setting rather than the original SupCon loss. Let $z_v^{(k)} = \text{normalize}(\text{pool}(H_{\text{vuln}}^{(k)}))$ and $z_s^{(k)} = \text{normalize}(\text{pool}(H_{\text{safe}}^{(k)}))$ denote the normalized vulnerable and repaired sequence representations, respectively. Here, $H_{\text{vuln}}^{(k)}$ and $H_{\text{safe}}^{(k)}$ are the final hidden-state matrices for the vulnerable and repaired members of pair k, $\text{pool}(\cdot)$ denotes mean pooling over valid sequence positions, and $\text{normalize}(\cdot)$ denotes ℓ_2 normalization.

For an anchor k, $\mathcal{P}_k$ contains the other vulnerable samples in the mini-batch that share its CWE label, and $\mathcal{K}$ contains the anchors for which this set is non-empty. If the entire mini-batch contains no same-CWE pair, the implementation uses all other vulnerable samples as fallback positives. Repaired representations are used only as negatives. The resulting objective is

$$\mathcal{L}_{\text{contra}} = -\frac{1}{|\mathcal{K}|} \sum_{k \in \mathcal{K}} \frac{1}{|\mathcal{P}_k|} \sum_{j \in \mathcal{P}_k} \log \frac{\exp(z_v^{(k)} \cdot z_v^{(j)} / \tau_c)}{\exp(z_v^{(k)} \cdot z_v^{(j)} / \tau_c) + \sum_{i=1}^{B} \exp(z_v^{(k)} \cdot z_s^{(i)} / \tau_c)}, \tag{3.31}$$

where B denotes the mini-batch size, $z_v^{(k)}$ and $z_s^{(i)}$ are unit-normalized vulnerable and repaired representations, respectively, and $\tau_c = 0.07$ is the contrastive temperature. The dot product is therefore cosine similarity. The fallback rule relaxes the CWE constraint only when no valid same-CWE pair exists in the complete mini-batch. Otherwise, anchors without a same-CWE positive are excluded from $\mathcal{K}$.

The TASR-CD objective is separated into two optimization paths according to sample type. In addition to the losses defined above, $\mathcal{L}_{\text{bce}}$ denotes the binary cross-entropy detection loss. The non-negative coefficients λ_{repair}, λ_{contra}, λ_{op}, λ_{slot}, λ_{del}, and λ_{bce} control the contribution of the six tasks. For Type A samples, all repair and detection objectives are jointly optimized:

$$\mathcal{L}_A = \lambda_{\text{repair}}\mathcal{L}_{\text{repair}} + \lambda_{\text{contra}}\mathcal{L}_{\text{contra}} + \lambda_{\text{op}}\mathcal{L}_{\text{op}} + \lambda_{\text{slot}}\mathcal{L}_{\text{slot}} + \lambda_{\text{del}}\mathcal{L}_{\text{del}} + \lambda_{\text{bce}}\mathcal{L}_{\text{bce}}. \quad (3.32)$$

For Type B samples, only the detection objective is optimized:

$$\mathcal{L}_B = \lambda_{\text{bce}}\mathcal{L}_{\text{bce}}. \quad (3.33)$$

The coefficients are epoch-dependent in the reported configuration. In epochs 1–3, $(\lambda_{\text{repair}}, \lambda_{\text{contra}}, \lambda_{\text{op}}, \lambda_{\text{slot}}, \lambda_{\text{del}}, \lambda_{\text{bce}}) = (0.60, 0, 0.045, 0.018, 0, 0.14)$. From epoch 4 onward, they take their target values $(0.65, 0.05, 0.05, 0.02, 0.03, 0.15)$. The TASR-CD detection loss uses positive-class weight $\eta = 0.8$. Thus, the repair objective remains the primary optimization signal, while the auxiliary heads provide lightweight structural supervision over edit behavior. Although gradient-surgery methods are reviewed in Chapter 2, TASR-CD does not apply PCGrad or project task gradients. Each Type A update uses the weighted sum in Equation 3.32, whereas a Type B update uses Equation 3.33.

– **Type-A/Type-B Training Procedure.**

Training interleaves the two paths. Each epoch samples 20,000 Type B cases. Type A and Type B use batch sizes 8 and 16, respectively. Type B batches are distributed across the Type A pass, with any remainder processed afterward. This schedule supplies detection-only calibration throughout the epoch without interrupting Type A span-repair learning.

In summary, Security-Aware Data Preprocessing constructs the model input, TASC shapes pre-training with token tiers, and TASR-CD aligns validated repairs while preserving a detection-calibration path. ThreatDiffusion contributes this integration and its tier-conditioned objectives, repair-pair routing, Token-Slot Alignment, and joint repair–detection training. The underlying embeddings, Transformer, diffusion process, and mask-to-edit curriculum remain cited foundations.

TAIPEI TECH

Chapter 4 Implementation

We implemented ThreatDiffusion as a reproducible training and evaluation pipeline following the methodology defined in Chapter 3. Security-Aware Data Preprocessing converts raw C/C++ functions into token, tier, and structural feature streams, while Repair-Pair Validation records whether each sample can provide reliable repair supervision. We then pre-train the shared Transformer backbone with Tier-Aware Security Curriculum (TASC) and fine-tune it for joint vulnerability detection and repair using Tier-Aware Span Repair with Contrastive Detection (TASR-CD).

Rather than repeating the derivations in Chapter 3, the discussion records the data flow, implementation constraints, hyperparameters, and evaluation protocols used in our experiments. Pseudocode is included for the procedures whose control flow directly affects reproducibility: feature construction and supervision routing, Token-Slot Alignment, and TASR-CD fine-tuning.

4.1 Environment Setup

We separated CPU-bound static analysis from GPU-bound Transformer training. Static analysis, tokenization, and span alignment were completed before model training, after which TASC and TASR-CD were trained on a single CUDA-enabled workstation. Table 4.1 lists the hardware environment used for the main experiments.

Table 4.1: Hardware specifications used in the experiments.

Component	**Specification**
Machine	Single-node workstation/server
Processor	Multi-core Intel CPU for preprocessing and data loading
GPU	NVIDIA RTX A4000 class CUDA GPU with 16 GB memory
Operating system	Ubuntu 22.04 LTS / containerized Linux environment
Memory	64 GB system memory
Storage	NVMe SSD storage for dataset, checkpoints, and logs

We implemented the pipeline in Python 3.11.9 with PyTorch 2.5.1. The UniXcoder tokenizer and embedding module are loaded through the Hugging Face Transformers interface. We reuse only its word, position, and token-type embeddings with layer normalization; the UniXcoder Transformer encoder is not executed. Static code analysis uses `tree-sitter`, and token-level repair alignment uses Python's `difflib.SequenceMatcher`. All major training scripts use a fixed random seed of 42. Table 4.2 lists the software components that are most relevant to reproducing the experiments.

Table 4.2: Software components used by ThreatDiffusion.

Component	**Version**	**Purpose**
Python	3.11.9	Main implementation language
PyTorch	2.5.1	Transformer training, mixed precision, and checkpointing
Transformers	4.44.2	UniXcoder tokenizer and pre-trained embedding-module loading
tree-sitter	0.21.3	C/C++ AST parsing for security-aware feature extraction
difflib	Python standard	Token-level diff extraction and span alignment
pandas / NumPy	2.0.2 / 1.24.3	Dataset loading, Repair-Pair Validation, and tensor preparation
scikit-learn	1.2.2	Metric computation and auxiliary evaluation utilities
TensorBoard	2.13.0	Training log visualization and metric tracking

4.2 Security-Aware Data Preprocessing

We convert raw VulGate samples into model-ready records containing token sequences, security tiers, structural threat features, and vulnerability labels. Token-level and structural features are constructed for every valid function. Repair-Pair Validation is performed as a supervision-routing step after feature construction: it does not prevent a valid function from entering tokenization or AST/CFG feature extraction, but only changes whether the sample is eligible to provide repair targets for TASR-CD. The resulting records are serialized once and

used by both TASC and TASR-CD.

4.2.1 Dataset Loading, Feature Construction, and Sample Routing

We load the chronological VulGate splits described in Chapter 3 and first convert each valid function into token-level and structural representations. If the sample contains a candidate repair pair, the pair is then validated to determine whether it can provide repair supervision. Samples with structurally valid vulnerable–repaired pairs are assigned to Type A. Samples without valid repair supervision, including repair pairs excluded by validation and ordinary detection-only samples, are assigned to Type B. Type A samples are used for repair and detection objectives, whereas Type B samples are used only for detection calibration.

Table 4.3: Training sample routing after Repair-Pair Validation.

Route	Count	Role
Type A	8,082	Clean vulnerable–repaired pairs with full repair, detection, and edit-structure supervision
Type B	181,248	Detection-only samples, including validation-excluded repair pairs and non-repair training samples

4.2.2 Tokenization and Static Feature Extraction

For each function, we use the UniXcoder tokenizer to produce a BPE sequence with a maximum length of 1,024 and parse the same source function with `tree-sitter`. The AST is traversed to identify API calls, arithmetic operators, taint sources, and guard contexts. An intra-procedural CFG derived from AST control nodes records local branch and loop context together with recognized validation checks in the analyzed local context. These features provide a conservative structural view and do not claim to recover every feasible runtime path. The per-token tier labels are aligned with the BPE sequence. The 64-dimensional threat vector remains a function-level summary that is serialized once per sample and is projected and broadcast across

sequence positions when the unified input is constructed.

Algorithm 1 records the implemented flow for feature construction and supervision routing. For each sample, we load the source fields, tokenize the vulnerable function, and construct the available security-aware features. Candidate repaired functions then undergo Repair-Pair Validation and Token-Slot Alignment before their supervision route is recorded. Samples without eligible repair supervision remain on the Type B detection path. The routing statistics and processed record are updated after each sample. The algorithm is executed independently for the training, validation, and test splits to preserve chronological separation.

TAIPEI TECH

Algorithm 1: Feature Construction and Supervision Routing

Input: VulGate split $\mathcal{D}$, UniXcoder tokenizer, CERT API sets

Output: Processed records $\mathcal{R}$

$\mathcal{R} \leftarrow \emptyset$

$stats \leftarrow$ InitializeCounters()

foreach *sample* $s \in \mathcal{D}$ **do**

  $(x_b, x_a, y) \leftarrow$ LoadFields(s)

  $x \leftarrow$ BPETokenize(x_b)

  $(AST, CFG) \leftarrow$ ParseGraph(x_b)

  if $AST = \emptyset$ **then**

    $R \leftarrow$ TypeBRecord$(x, 5\mathbf{1}_L, \mathbf{0}_{64}, y)$

    $stats \leftarrow$ UpdateCounter$(stats, R.\text{type})$

    $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$

    continue

  $(\tau, \mathbf{S}) \leftarrow$ AnalyzeThreat(x, AST, CFG)

  if $x_a \neq \emptyset \wedge$ ValidateRepairPair(x_b, x_a) **then**

    $x_a^{\text{tok}} \leftarrow$ BPETokenize(x_a)

    $A \leftarrow$ TokenSlotAlign(x, x_a^{tok})

    if $A \neq \emptyset$ **then**

      $R \leftarrow$ RouteRepair$(x, \tau, \mathbf{S}, y, A)$

    else

      $R \leftarrow$ TypeBRecord$(x, \tau, \mathbf{S}, y)$

  else

    $R \leftarrow$ TypeBRecord$(x, \tau, \mathbf{S}, y)$

  $stats \leftarrow$ UpdateCounter$(stats, R.\text{type})$

  $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$

SaveStatistics$(stats)$

return $\mathcal{R}$

TAIPEI TECH

4.2.3 Unified Input Serialization

Each processed record stores the components needed to construct the unified input representation from Chapter 3: the UniXcoder token ids, attention masks, per-token tier labels, one function-level 64-dimensional threat vector, and the vulnerability label. During model input construction, the threat vector is projected and broadcast across the sequence rather than being independently aligned to individual BPE tokens. For Type A records, additional repair fields are stored, including the repaired token target, span masks, operation labels, slot-count labels, and deletion labels. For Type B records, these repair-specific fields are disabled and only the detection label is used during TASR-CD.

4.3 Tier-Aware Security Curriculum Implementation

We pre-train the ThreatDiffusion backbone with the two-phase TASC procedure defined in Chapter 3: Phase 1 performs tier-aware masked reconstruction, and Phase 2 continues training with tier-weighted edit perturbations. Both phases operate on the same shared Transformer backbone and produce a checkpoint used to initialize TASR-CD.

4.3.1 Backbone Initialization

We use a 12-layer Pre-LN Transformer with 12 attention heads and hidden dimension 960. The dimension corresponds to the concatenation of 768-dimensional outputs from the UniXcoder embedding module, 64-dimensional tier embeddings, 64-dimensional structural threat embeddings, and 64-dimensional timestep embeddings. The concatenated representation is processed by the learned $960 \rightarrow 960$ fusion projection, GELU, and layer normalization defined in Section 3.2.1. The UniXcoder embedding module is trainable during TASC. The maximum sequence length is fixed at 1,024. The same architecture is reused during TASR-CD fine-tuning.

Table 4.4: Shared Transformer backbone configuration.

Parameter	Value	Description
Hidden dimension	960	Concatenated lexical, tier, threat, and time features
Transformer layers	12	Shared Pre-LN Transformer layers
Attention heads	12	Multi-head self-attention heads
Dropout	0.1	Regularization during training
Vocabulary size	51,416	UniXcoder vocabulary size
Maximum sequence length	1,024	Fixed input length for all model stages

4.3.2 Two-Phase TASC Training

TASC trains one shared backbone across two consecutive phases. Phase 1 uses tier-aware masked reconstruction. For each sample in a batch, a continuous timestep is sampled, converted into a cosine masking probability, and then scaled by the token's security tier. As a result, Tier 0 and Tier 1 positions are corrupted more often than benign context tokens. The reconstruction objective applies the capped MDLM timestep weight together with tier, mask-position, and flaw-token weights. Focal reweighting is applied to Tier 0–2 positions, and label smoothing is used in the token objective. Their mathematical definitions are given in Equations 3.17–3.19.

For Phase 2, we continue from the Phase 1 checkpoint and introduce tier-weighted edit perturbations. Edit positions are sampled from valid non-padding positions according to the tier-weighted distribution defined in Chapter 3. Each selected position undergoes substitution, insertion, or deletion. The Phase 2 objective combines tier-weighted token reconstruction with function-level vulnerability detection. This phase exposes the backbone to localized repair behavior while retaining the denoising ability learned in Phase 1. The final TASC checkpoint is selected after an early warmup period, using validation loss together with a detection-performance constraint to avoid selecting a checkpoint that reconstructs tokens well but weakens vulnerability discrimination. The security-specific tier multipliers, reconstruction weights, edit weights, and corruption schedules used in both phases follow the definitions accompanying Equations 3.16–3.21 in Chapter 3.

Table 4.5: TASC training configuration.

Parameter	Value	Description
Phase 1 epochs	35	Tier-Aware masked reconstruction
Phase 2 epochs	15	Tier-Weighted edit training
Batch size	8	Mini-batch size before gradient accumulation
Gradient accumulation	16	Effective batch size 128
Optimizer	AdamW	Weight decay 0.01
Peak learning rate	3.0×10^{-4}	TASC Phase 1 learning rate
Phase 2 learning rate	8.0×10^{-5}	Lower learning rate for edit training
Scheduler	WSD	Warmup-Stable-Decay schedule
Maximum gradient norm	1.0	Gradient clipping threshold

4.4 Token-Slot Alignment Implementation

Token-Slot Alignment converts each repair pair that passes Repair-Pair Validation into fixed-coordinate supervision for the shared Pre-LN Transformer. The implementation uses `difflib.SequenceMatcher` to extract `equal`, `replace`, `insert`, and `delete` spans. Each span is allocated according to Equation 3.23.

Equal spans retain the original tokens. Replacement spans retain available vulnerable-side tokens and use active `[PAD]` only when the repaired span is longer. The additional repaired-side positions are labeled as insertions, whereas additional vulnerable-side positions are retained and labeled for deletion. Insertion spans use active `[PAD]` inputs, while deletion spans retain their source tokens and receive zero repair-token weight. Active positions have an attention value of one. Inactive tail padding has an attention value of zero. Token-Slot Alignment does not introduce `[MASK]`, which is reserved for diffusion corruption.

Algorithm 2 records the corresponding implementation. The span cap $K_{\max} = 16$ truncates an individual edit span without changing its supervision route. Therefore, all 8,082 repair pairs that pass Repair-Pair Validation remain Type A.

Algorithm 2: Token-Slot Alignment Construction

Input: Vulnerable tokens x_{vuln}, repaired tokens x_{safe}, span cap $K_{\max} = 16$, and aligned length L_{span}

Output: Span-aligned Type A repair record

$Ops \leftarrow \text{SequenceMatcher}(x_{\text{vuln}}, x_{\text{safe}})$

$B \leftarrow \emptyset; \quad Y_{\text{del}} \leftarrow \mathbf{0}^{|x_{\text{vuln}}|}$

foreach $(o_k, S_k^{vuln}, S_k^{safe}) \in Ops$ **do**

- $C_k \leftarrow \text{Capacity}(o_k, S_k^{\text{vuln}}, S_k^{\text{safe}}, K_{\max})$ according to Equation 3.23
- **if** $o_k = \texttt{equal}$ **then**
  - append copied inputs, targets, and `equal` labels to B
- **else if** $o_k = \texttt{replace}$ **then**
  - retain vulnerable-side tokens and append active `[PAD]` for repaired-side surplus
  - assign `replace` to paired positions, `insert` to repaired-side surplus, and `delete` to vulnerable-side surplus
  - derive insertion-slot labels and mark deleted source indices in Y_{del}
- **else if** $o_k = \texttt{insert}$ **then**
  - append C_k active `[PAD]` inputs with repaired-token targets
  - assign `insert` labels and derive insertion-slot labels
- **else**
  - retain C_k vulnerable-side tokens and assign `delete` labels
  - set their repair-token weights to zero and mark their source indices in Y_{del}

$(X_{\text{span}}, Y_{\text{repair}}, Y_{\text{op}}, Y_{\text{slot}}, a_{\text{span}}) \leftarrow \text{Materialize}(B, L_{\text{span}})$

return $\text{TypeARecord}(X_{span}, Y_{repair}, Y_{op}, Y_{slot}, Y_{del}, a_{span})$

The materialization step retains at most L_{span} active positions and fills the unused tail with inactive `[PAD]`. The returned record provides the aligned repair, operation, slot-count, and deletion supervision used during TASR-CD. Deletion positions remain visible to the Transformer but are excluded from repair-token loss according to Equation 3.26.

TAIPEI TECH

4.5 TASR-CD Fine-Tuning Implementation

For TASR-CD fine-tuning, we reuse the TASC backbone together with its Repair Head and Detection Head. We newly initialize the Operation Type, Slot Count, and Deletion Heads for the edit-structure supervision defined in Chapter 3. The UniXcoder embedding module is frozen during fine-tuning, while the shared Pre-LN Transformer and prediction heads are optimized jointly. Type A batches provide span repair, edit-structure, contrastive, and detection supervision. Type B batches provide detection-only calibration alongside the Type A updates.

4.5.1 Prediction Heads

TASR-CD uses five prediction heads. The Repair and Detection Heads are inherited from TASC, whereas the Operation Type, Slot Count, and Deletion Heads are newly initialized. The new-head learning-rate multiplier applies only to these three edit-structure heads. Table 4.6 lists their implementation roles.

Table 4.6: Prediction heads implemented in TASR-CD.

Head	Supervision	Role
Repair Head	Type A	Predicts repaired tokens in span-aligned coordinates
Detection Head	Type A, Type B	Predicts vulnerable or safe function labels
Operation Type Head	Type A	Predicts equal, replace, insert, and delete operations
Slot Count Head	Type A	Predicts the number of inserted repair tokens
Deletion Head	Type A	Predicts deleted positions in vulnerable-sequence space

4.5.2 Type-A and Type-B Batch Scheduling

We maintain separate loaders for Type A and Type B samples. Type A samples provide repair, detection, and edit-structure supervision. Type B samples retain their vulnerability labels and are used only for detection training. At the beginning of each epoch, 20,000 Type B samples are selected to form the detection pool. One Type B batch is placed after each Type A batch while

samples remain in the pool. Any remaining Type B batches are processed after the Type A loader is exhausted.

We start from the TASC checkpoint, retaining the shared backbone, Repair Head, and Detection Head while initializing the Operation Type, Slot Count, and Deletion Heads for TASR-CD. A learning-rate multiplier of 5.0 is applied only to these three new heads. The UniXcoder embedding module is frozen throughout fine-tuning. Algorithm 3 gives the resulting training loop.

TAIPEI TECH

Algorithm 3: TASR-CD Fine-Tuning

Input: TASC checkpoint θ^{TASC} and loaders $\mathcal{D}_A, \mathcal{D}_B$

Output: Selected raw and EMA checkpoints $\mathcal{C}^*$

$\theta \leftarrow \text{LoadCheckpoint}(\theta^{\text{TASC}})$

Reuse the TASC Repair and Detection Heads

Initialize the Operation Type, Slot Count, and Deletion Heads with a learning-rate multiplier of 5.0

Freeze the UniXcoder embedding module

$(Opt, Sch, AMP, EMA) \leftarrow \text{InitializeTrainingState}(\theta)$

$A \leftarrow 8; \quad \mathcal{C}^* \leftarrow \emptyset$

$\lambda^{\text{warm}} \leftarrow (0.60, 0, 0.045, 0.018, 0, 0.14)$

$\lambda^{\text{target}} \leftarrow (0.65, 0.05, 0.05, 0.02, 0.03, 0.15)$

for $epoch \leftarrow 1$ **to** 50 **do**

  $\lambda \leftarrow \lambda^{\text{warm}}$ if $epoch \leq 3$, otherwise λ^{target}

  $P_B \leftarrow \text{SampleTypeBPool}(\mathcal{D}_B, 20000)$

  $Q \leftarrow \text{InterleaveThenAppend}(\mathcal{D}_A, P_B)$

  $\text{ZeroGrad}(Opt)$

  for $j \leftarrow 1$ **to** $|Q|$ **do**

    $(B, r) \leftarrow Q_j$

    $\mathcal{L} \leftarrow \text{SelectLoss}(r, \theta, B; \lambda)$

    $\text{AMPBackward}(\mathcal{L}/A)$

    if $j \bmod A = 0$ ***or*** $j = |Q|$ **then**

      $\text{OptimizerStep}(Opt, Sch, EMA, \theta, 1.0)$

  $(m_{\text{raw}}, m_{\text{EMA}}) \leftarrow \text{ValidateRawAndEMA}(\theta, EMA)$

  $\mathcal{C}^* \leftarrow \text{RetainByConfiguredCriteria}(\mathcal{C}^*, \theta, EMA, m_{\text{raw}}, m_{\text{EMA}})$

return $\mathcal{C}^*$

TAIPEI TECH

In Algorithm 3, $\mathcal{D}_A$ and $\mathcal{D}_B$ denote the Type A and Type B loaders. P_B is the Type B pool sampled for the current epoch, and Q is the ordered batch sequence after interleaving. The route variable r determines whether SelectLoss applies the joint Type A objective or the detection-only Type B objective. The loss vector follows the order $(\lambda_{\text{repair}}, \lambda_{\text{contra}}, \lambda_{\text{op}}, \lambda_{\text{slot}}, \lambda_{\text{del}}, \lambda_{\text{bce}})$.

The accumulation factor A is set to eight. OptimizerStep clips the gradient norm at 1.0, updates AdamW and the cosine scheduler, updates the EMA weights, and clears the accumulated gradients. At the end of each epoch, m_{raw} and m_{EMA} denote the validation measurements of the raw and EMA models. These measurements are used to update the retained checkpoint set $\mathcal{C}^*$.

Table 4.7: TASR-CD fine-tuning configuration.

Parameter	**Value**	**Description**
Epochs	50	Fine-tuning epochs
Batch size	8	Type A mini-batch size
Type B batch size	16	Detection-only mini-batch size
Type B samples per epoch	20,000	Type B pool sampled in each epoch
Gradient accumulation	8	Micro-batches per optimizer update
Learning rate	5.0×10^{-5}	Base fine-tuning learning rate
New-head multiplier	5.0	Applied to the Operation Type, Slot Count, and Deletion Heads
Weight decay	0.03	AdamW weight decay
Maximum gradient norm	1.0	Gradient-clipping threshold
EMA decay	0.9995	Exponential moving-average decay
Detection threshold	0.30	Fixed decision threshold used during evaluation

For the first three epochs, we use the warm-up weights shown in Table 4.8. The contrastive and deletion losses are disabled during this period. From the fourth epoch onward, training proceeds with the complete joint objective.

TAIPEI TECH

Table 4.8: TASR-CD loss weights.

Loss term	Epochs 1–3	Epoch 4 onward	Applied to
$\mathcal{L}_{\text{repair}}$	0.60	0.65	Type A span-repair reconstruction
$\mathcal{L}_{\text{contra}}$	0	0.05	Type A contrastive representation learning
$\mathcal{L}_{\text{op}}$	0.045	0.05	Type A operation-type prediction
$\mathcal{L}_{\text{slot}}$	0.018	0.02	Type A insertion slot-count prediction
$\mathcal{L}_{\text{del}}$	0	0.03	Type A deletion prediction
$\mathcal{L}_{\text{bce}}$	0.14	0.15	Type A and Type B vulnerability detection

4.6 Evaluation Protocol

We evaluate vulnerability detection and repair quality separately. Detection uses the complete test split, including both Type A and Type B samples. Repair is evaluated only on Type A samples because they contain paired repaired targets. This separation prevents detection-only samples from being treated as repair failures.

4.6.1 Inference and Repair Reconstruction

At evaluation time, the selected EMA checkpoint performs a single forward pass for each sample. Detection uses the function-level probability produced by the Detection Head. For Type A repair samples, token predictions are obtained by greedy argmax at active aligned positions. The operation, slot-count, and deletion predictions are then applied to the aligned record to reconstruct the full-function output. Inactive tail padding is excluded. The top-K results reported in Chapter 5 are oracle diagnostics and are not used in the main inference procedure.

4.6.2 Detection Metrics

The Detection Head outputs a vulnerability probability for each function, which is thresholded at 0.30 during TASR-CD evaluation. The reported detection metrics are F1-score, precision, recall, accuracy, and the confusion matrix. In addition, a Hard-Negative F1-score is computed on samples whose vulnerable and repaired versions have cosine similarity greater than or equal to 0.90, following the definition in Section 3.1.1. This subset is important because lexical similarity alone is insufficient for distinguishing vulnerable and safe behavior.

4.6.3 Repair Metrics

We evaluate repair predictions from complementary perspectives. Span Token Accuracy measures token correctness at the evaluated repair-span positions. Aligned-record percent perfect prediction (%PP) measures the percentage of Type A samples for which all valid positions in the aligned repair record match the reference targets. Inactive tail padding is excluded from this calculation. Full Token Accuracy measures token correctness across the reconstructed full function, while Full Sequence %PP requires exact full-function reconstruction. Operation Type Accuracy measures the auxiliary head's predictions over the equal, replace, insert, and delete labels at evaluated aligned positions, as also reported in Chapter 5. Because these metrics cover different evaluation scopes, the span-position, aligned-record, and full-function results are reported separately.

TAIPEI TECH

Table 4.9: Evaluation metrics used in ThreatDiffusion.

Metric	Description
F1-score	Harmonic mean of detection precision and recall
Precision	Fraction of predicted vulnerable samples that are truly vulnerable
Recall	Fraction of vulnerable samples correctly detected
Hard-Negative F1	Detection F1 on vulnerable–repaired pairs with cosine similarity $\geq$ 0.90
Span Token Accuracy	Token accuracy restricted to span-aligned repair positions
Aligned-record %PP	Exact match over all valid aligned-record positions
Full Token Accuracy	Token accuracy over reconstructed full-function output
Full Sequence %PP	Strict full-function exact-match repair rate
Operation Type Accuracy	Accuracy of equal, replace, insert, and delete predictions at evaluated aligned positions

4.6.4 Checkpoint Selection

We save TASC checkpoints according to validation loss and detection performance. TASR-CD retains several checkpoints because detection and repair objectives do not always peak at the same epoch. The implementation records the best overall, detection, aligned-record %PP, and span-accuracy checkpoints, together with their EMA-smoothed variants. The final evaluation uses the EMA checkpoint selected by validation aligned-record %PP. All detection and repair results reported in Chapter 5 are obtained from this checkpoint.

4.6.5 Ablation and Baseline Evaluation

We implement the ablations as diagnostic training configurations associated with the major ThreatDiffusion components. The ablation labels denote changes to the corresponding training conditions, while the evaluation protocol remains unchanged. The main ablation dimensions correspond to TASC, tier embedding, structural threat features, TASR-CD, and edit auxiliary su-

pervision. The precise changes associated with each configuration are described in Section 5.6.

The repair evaluation does not include an external baseline. Existing reported results use input–output formats and evaluation protocols that are not directly comparable with the span-aligned setting used in this work. Repair performance is therefore evaluated using the repair metrics, internal ablations, and token-level top-K oracle analysis reported in Chapter 5.

4.7 Implementation Summary

The resulting pipeline connects Security-Aware Data Preprocessing and serialized supervision routing with TASC pre-training, Token-Slot Alignment, TASR-CD fine-tuning, checkpoint management, and evaluation. Chapter 5 reports the resulting model performance and the empirical contribution of its major components.

TAIPEI TECH

Chapter 5 Results and Analysis

This chapter reports the experimental results for ThreatDiffusion. Function-level vulnerability detection and span-level repair are evaluated separately because they use different prediction units and error criteria. We first examine the optimization behavior of TASC and TASR-CD, then analyze test performance and component ablations under the protocol defined in Chapters 3 and 4.

5.1 Evaluation Metrics

5.1.1 Vulnerability Detection Metrics

For vulnerability detection, each function-level sample is treated as either vulnerable or safe. The prediction outcome can therefore be described by four basic cases:

- **True Positive (TP)**: a vulnerable function is correctly predicted as vulnerable.
- **True Negative (TN)**: a safe function is correctly predicted as safe.
- **False Positive (FP)**: a safe function is incorrectly predicted as vulnerable. In vulnerability detection, this corresponds to a false alarm.
- **False Negative (FN)**: a vulnerable function is incorrectly predicted as safe, meaning the underlying vulnerability fails to be detected.

Based on these definitions, this chapter reports Accuracy, Precision, Recall, and F1-score for function-level vulnerability detection. As shown in Equation 5.1, Accuracy measures the proportion of correctly classified samples among all predictions:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \tag{5.1}$$

Precision measures the reliability of positive predictions. As shown in Equation 5.2, it is the proportion of truly vulnerable samples among all samples predicted as vulnerable:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{5.2}$$

Recall measures the coverage of vulnerable samples. As shown in Equation 5.3, it is the proportion of actual vulnerable samples that are detected:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{5.3}$$

F1-score is the harmonic mean of Precision and Recall. As shown in Equation 5.4, it provides a balanced measure when false alarms and missed vulnerabilities must both be considered:

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \tag{5.4}$$

The detection threshold is fixed at 0.30 in the TASR-CD evaluation configuration and is applied to the test split without additional threshold scanning. Following the evaluation concerns raised by Risse et al. [14], this chapter also reports Hard-Negative F1 (HN-F1). It uses the same F1 definition on the subset with cosine similarity greater than or equal to 0.90, as defined in Sections 3.1.1 and 4.6.2. This subset contains 7,602 test samples and measures whether the detector can reject highly similar vulnerable–safe pairs.

5.1.2 Vulnerability Repair Metrics

For vulnerability repair, following VulRepair and subsequent methodological studies of automated program repair [9, 64], the evaluation focuses on whether the model can predict the reference repair tokens in the localized edit region. To evaluate repair performance from token-level agreement to exact reference-sequence match, this chapter adopts five specific metrics:

- **Span Token Accuracy (SpanAcc)**: evaluates token-level correctness at the localized repair-span positions.
- **Full Token Accuracy (FullTokenAcc)**: evaluates token-level correctness over the reconstructed full function.

- **Aligned-record Percentage of Perfect Prediction (Aligned-record %PP)**: measures the proportion of Type A samples for which all valid positions in the aligned repair record match the reference targets.

- **Full Sequence Percentage of Perfect Prediction (Full Seq. %PP)**: measures the proportion of samples for which the entire reconstructed function matches the reference safe function.

- **Operation Type Accuracy (Op Type Acc)**: evaluates the equal, replace, insert, and delete labels predicted by the auxiliary head at aligned positions.

The detailed mathematical formulations of these metrics are defined as follows. Span Token Accuracy measures correctness only within aligned repair spans. Given a set of evaluated repair positions Ω_{span}, the metric is defined as

$$\text{SpanAcc} = \frac{\sum_{i \in \Omega_{\text{span}}} \mathbb{I}\left[\hat{y}_i = y_i\right]}{|\Omega_{\text{span}}|}. \tag{5.5}$$

where y_i denotes the target repair token and $\hat{y}_i$ denotes the predicted token at the same aligned position. Full Token Accuracy uses the same token-level comparison, but over the reconstructed full function:

$$\text{FullTokenAcc} = \frac{\sum_{i \in \Omega_{\text{full}}} \mathbb{I}\left[\hat{f}_i = f_i^{\text{safe}}\right]}{|\Omega_{\text{full}}|}. \tag{5.6}$$

where $\hat{f}_i$ is the predicted token at reconstructed position i, f_i^{safe} is the corresponding token in the safe target function, and Ω_{full} is the set of valid positions in the reconstructed function.

Since most tokens in a function are unchanged, Full Token Accuracy is useful for verifying reconstruction consistency, but it may overstate repair quality. Span Token Accuracy is therefore treated as the main localized repair accuracy metric.

This chapter also reports the percentage of perfect prediction (%PP), an exact-match metric used in vulnerability repair [9, 64]. For a set of evaluated samples $\mathcal{D}$, %PP is defined as the proportion of samples whose predicted sequence exactly matches the target sequence:

$$\%\text{PP} = \frac{1}{|\mathcal{D}|} \sum_{j=1}^{|\mathcal{D}|} \mathbb{I}\left[\hat{\mathbf{y}}^{(j)} = \mathbf{y}^{(j)}\right]. \tag{5.7}$$

Aligned-record %PP applies this definition to all valid positions in the aligned repair record, excluding inactive tail padding. Full Sequence %PP instead applies it to the reconstructed full function. The former requires exact agreement with the reference targets in the aligned representation, whereas the latter requires the reconstructed function to match the reference safe function.

For ablation analysis, component impact is measured by the signed relative change from the full model:

$$\Delta_m(c) = \frac{m(\text{w/o } c) - m(\text{Full})}{m(\text{Full})} \times 100\%, \tag{5.8}$$

where m is the metric of interest and c denotes the evaluated ablation condition, indicated by a minus sign '$-$' in the tables. For metrics where higher values are better, a more negative $\Delta_m(c)$ indicates a larger performance decrease under that condition.

5.2 TASC Pre-training Analysis

TASC trains the shared backbone before TASR-CD fine-tuning rather than serving as the final repair model. We therefore examine its reconstruction behavior and the transition from masked reconstruction to joint edit and detection supervision.

Figure 5.1 shows the training dynamics across the two curriculum phases.

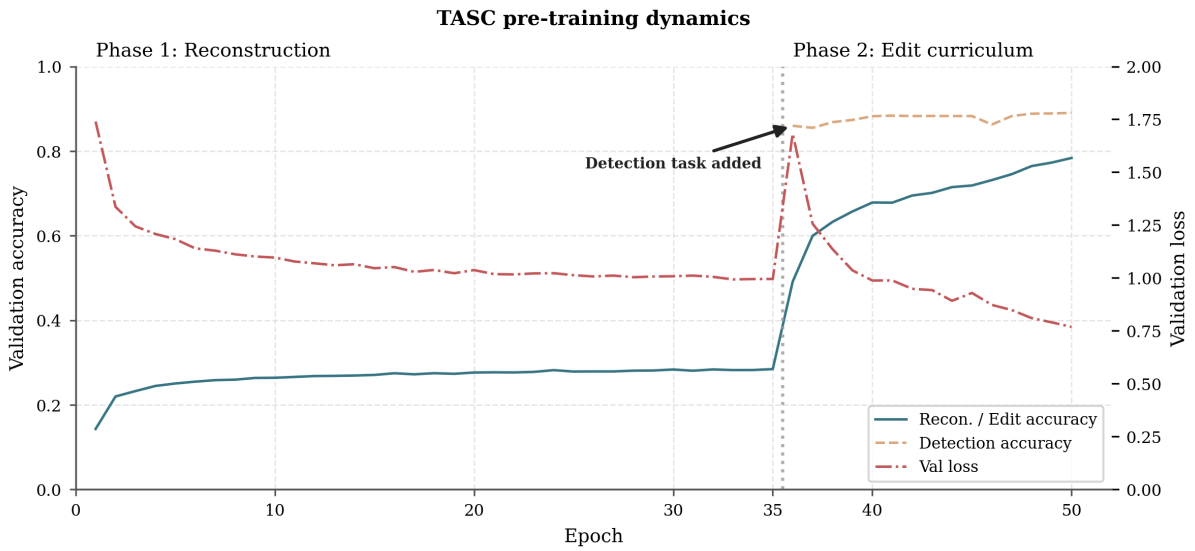


Figure 5.1: TASC pre-training dynamics.

During Phase 1 (Epochs 1–35), the model is trained with the tier-aware masked reconstruction objective without detection supervision. The validation loss decreases steadily from approximately 1.75 to 1.0, while reconstruction accuracy gradually improves from 0.15 to 0.28. This trend indicates that the backbone learns a stable denoising representation over general source-code tokens before any detection-related supervision is introduced.

At Epoch 36, the curriculum enters Phase 2, where edit-oriented supervision and the auxiliary vulnerability detection task are added. The validation loss shows a short spike immediately after the transition, which reflects the increased optimization difficulty introduced by the new objectives. After this point, the loss quickly recovers and continues to decrease. The plotted detection accuracy rises rapidly to approximately 0.86 and gradually approaches 0.90, while reconstruction accuracy continues to improve to approximately 0.78. Phase 1 accuracy is computed only over masked positions, whereas Phase 2 accuracy is computed over all valid positions. The values on the two sides of the curriculum transition are therefore not directly comparable, and the apparent increase partly reflects the change in evaluation scope. Nevertheless, the subsequent curves show that the joint Phase 2 objective remains stable after the transition.

Table 5.1 compares tier-aware and uniform masking in the TASC pre-training evaluation.

TAIPEI TECH

Table 5.1: Masking-policy comparison in the TASC pre-training evaluation.

Configuration	Token Acc.	Detection F1	Precision	Recall
Tier-aware masking	77.45%	89.13%	86.94%	91.44%
Uniform masking	63.24%	88.22%	86.78%	89.71%

The largest difference appears in token accuracy. Tier-aware masking achieves 77.45%, compared with 63.24% under uniform masking. This result is consistent with more effective reconstruction under tier-aware corruption. The detection metrics remain close across the two settings. The tier-aware and uniform configurations obtain detection F1-scores of 89.13% and 88.22%, respectively, with smaller differences in precision and recall. These results support the use of tier-aware masking for reconstruction while maintaining comparable vulnerability-detection performance. This comparison evaluates the masking policy rather than the complete TASC training condition.

The stable Phase 2 behavior and higher token accuracy support using the tier-aware TASC checkpoint to initialize TASR-CD.

5.3 TASR-CD Fine-tuning Analysis

After TASC pre-training, TASR-CD jointly optimizes localized repair with Type A samples and function-level detection with Type A and Type B samples. The curves are examined separately because the two objectives converge at different rates.

Figures 5.2, 5.3, and 5.4 report the loss, detection, and repair curves over 50 fine-tuning epochs.

Figure 5.2 shows the optimization behavior of the joint objective. The total loss decreases over training, while the repair, contrastive, and BCE components remain bounded and follow a generally downward trend. This pattern does not by itself establish the absence of task interference, but it is consistent with the intended design in which repair generation and vulnerability detection are trained under separate loss terms and calibrated through Type A/Type B sample routing.

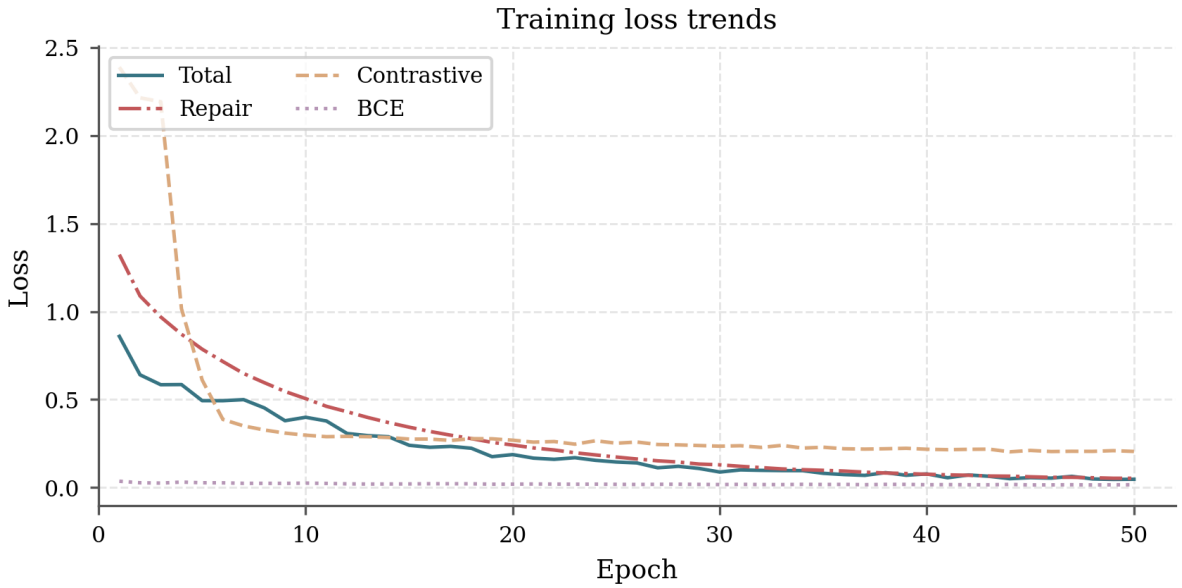


Figure 5.2: Training loss trends during TASR-CD fine-tuning.

The validation curves show an asymmetric convergence pattern. In Figure 5.3, the detection metrics enter a stable region early and remain close to the final test-level behavior throughout most of the fine-tuning process. Function-level vulnerability detection therefore stabilizes earlier than the localized repair objective in this run.

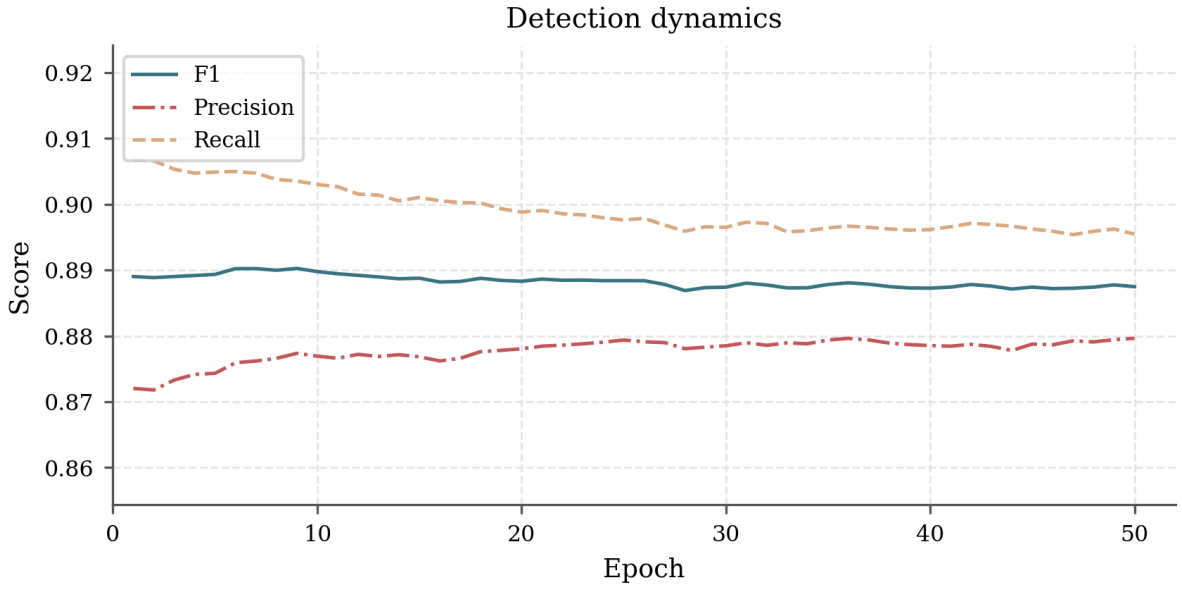


Figure 5.3: TASR-CD detection dynamics.

Unlike detection, the span-level repair metrics in Figure 5.4 continue to improve over later epochs. The span token accuracy reaches optimal validation performance near Epoch 20 and

then plateaus, while Aligned-record %PP follows a similar stable trajectory. This pattern indicates that the model continues to refine localized token choices after the detection boundary has become relatively stable. Full Token Accuracy starts near 80% and quickly approaches 100%. Because unchanged context dominates this metric, the curve is consistent with stable full-function reconstruction but does not by itself establish localized repair quality.

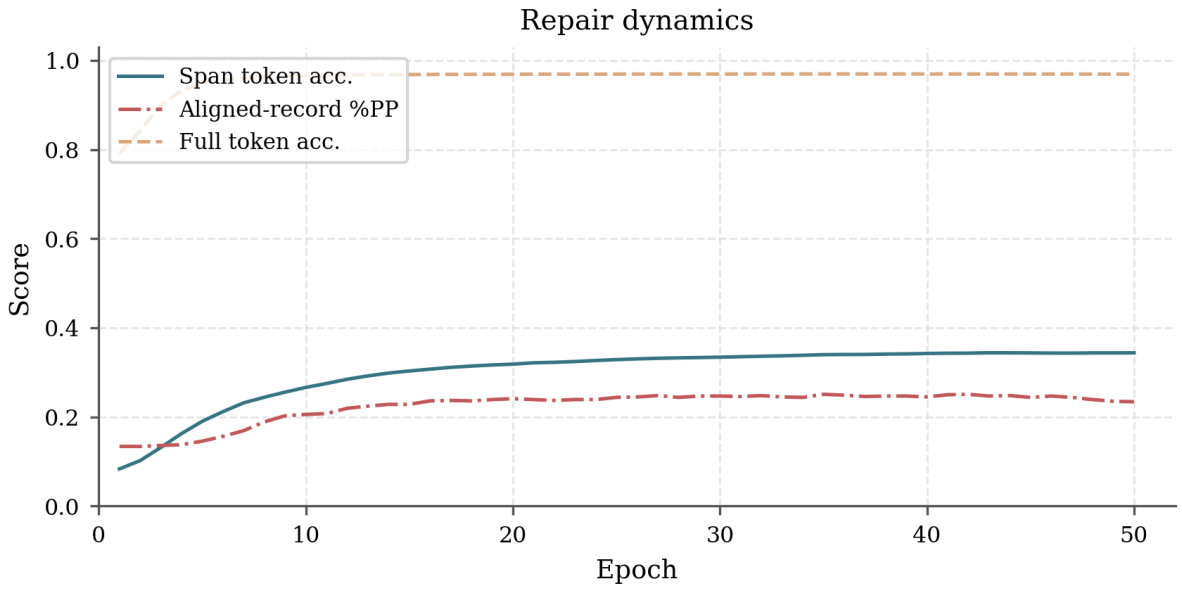


Figure 5.4: TASR-CD repair dynamics.

Although the two tasks are optimized jointly, their validation metrics reach their best values at different epochs.Detection depends on a calibrated function-level boundary, whereas repair requires exact token choices within aligned edit spans. The final evaluation therefore uses the EMA checkpoint selected by validation Aligned-record %PP, following the implementation protocol described in Chapter 4.

5.4 Overall Performance Evaluation

5.4.1 Detection and Repair Performance of ThreatDiffusion

Table 5.2 reports the overall test performance of ThreatDiffusion. All detection and repair metrics are obtained from the same EMA checkpoint selected by validation aligned-record %PP. Detection is evaluated using standard and Hard-Negative metrics, while repair is evaluated at the span and full-function levels. Span-level metrics focus on edited regions, whereas full-function accuracy is dominated by unchanged tokens.

Table 5.2: Overall test performance of ThreatDiffusion.

Task	Metric	Result
Detection	F1-score	88.97%
Detection	Precision	88.24%
Detection	Recall	89.71%
Detection	Hard-Negative F1	81.35%
Detection	Accuracy	88.98%
Repair	Span Token Accuracy	34.59%
Repair	Aligned-record %PP	26.53%
Repair	Full Token Accuracy	97.13%
Repair	Full Sequence %PP	15.22%

The repair metrics in Table 5.2 are computed on all 999 Type A samples in the test split.

The detection F1-score is 88.97%, while HN-F1 falls to 81.35% on highly similar vulnerable–safe pairs. For repair, the gap between 97.13% Full Token Accuracy and 34.59% Span Token Accuracy shows why unchanged context cannot be used as the sole measure of repair quality. The lower exact-match rates further indicate that reproducing the complete target patch remains more difficult than token-level reconstruction. These metrics measure agreement with the reference repair tokens and do not assess compilation, functional correctness, or security validity.

5.4.2 Baseline Comparison on Vulnerability Detection

To contextualize the detection result, Table 5.3 presents five baseline results reported by Safdar et al. [13]: Cppcheck [65], Infer [27], CodeBERT [48], UniXcoder-Base, and UniXcoder-Base-Nine [50]. These values are taken from the VulGate study rather than rerun in the Threat-Diffusion pipeline and are therefore treated as external benchmark references.

UniXcoder-Base-Nine reports an F1-score of 88.9%, which is comparable to the 88.97% obtained by ThreatDiffusion. These values come from different evaluation runs and do not establish a meaningful detection advantage; ThreatDiffusion is evaluated here additionally on span-level repair.

Table 5.3: Detection comparison on the VulGate dataset.

Model / Tool	**F1-score**	**Precision**	**Recall**
Cppcheck [65]	29.0%	54.0%	20.0%
Infer [27]	42.0%	26.0%	98.0%
CodeBERT [48]	85.9%	83.2%	89.0%
UniXcoder-Base [50]	87.0%	85.4%	88.7%
UniXcoder-Base-Nine [50]	88.9%	87.7%	**90.0%**
ThreatDiffusion (Ours)	**88.97%**	**88.24%**	89.71%

5.5 Detection and Repair Analysis

The following analysis separates confusion-matrix behavior, tier-wise detection, span-level repair, and token-level top-K oracle results.

Figure 5.5 is interpreted together with the reported precision and recall.

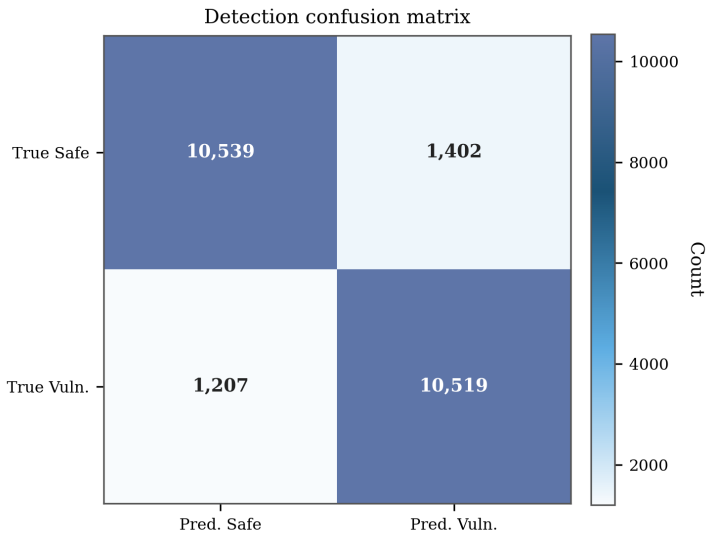


Figure 5.5: Confusion matrix of vulnerability detection on the test set.

The number of true positives and true negatives shows that the model correctly classifies many vulnerable and benign functions. Analyzing false predictions is equally important in the context of automated code review: false positives misguide developers into inspecting safe code, while false negatives fail to catch actual code-level flaws before deployment.

Tier-wise detection groups each function by the highest-risk tier among its valid tokens: Tier 0 takes precedence over Tier 1, followed by Tier 2 and Tier 5.

Table 5.4: Tier-wise vulnerability detection performance.

Tier	F1-score	Precision	Recall
Tier 0 (Critical API)	94.36%	92.86%	95.91%
Tier 1 (High-risk)	87.59%	84.24%	91.22%
Tier 2 (Medium-risk)	85.20%	85.59%	84.82%
Tier 5 (Background)	79.88%	77.60%	82.29%

As shown in Table 5.4, higher-risk tiers obtain higher F1-scores than Tier 5. Tier 5 remains more difficult because it contains functions without recognized Tier 0–2 token signals, making the local security cues provided by the tier map weaker. The ordering is consistent with stronger local security signals in the higher-risk groups, although it does not isolate the causal effect of the tier representation.

Repair performance is examined at several granularities. Span Token Accuracy measures token correctness at the repair-span positions. Aligned-record %PP requires all valid positions in the aligned repair record to match the reference targets. Full Token Accuracy measures reconstruction consistency over the full function, but unchanged context tokens dominate its denominator.

Table 5.5 reports a token-level top-K oracle over positions selected by the span mask, rather than sequence-level beam decoding.

Table 5.5: Token-level top-K oracle over repair-span positions.

Candidate Set	Oracle Exact Match
Greedy@1	24.12%
Top-5 oracle	35.15%
Top-50 oracle	46.87%

Greedy@1 and the larger candidate sets are evaluated only at positions selected by the span mask. The oracle subset contains 879 Type A samples with a non-empty span mask. For $K > 1$, a sample is counted as correct when the target token appears among the top-K predictions at every selected position. In contrast, the 26.53% Aligned-record %PP in Table 5.2 applies greedy exact match to all valid positions in the aligned repair record. The two values use different position sets and are therefore not expected to be identical. The top-K results should be interpreted as an oracle diagnostic rather than deployment performance.

No external repair baseline uses the same span-aligned supervision and output protocol. Repair quality is therefore assessed through the reported metrics, internal ablations, and this top-K oracle.

5.6 Ablation Study

Tables 5.6 and 5.7 report the effects of the major ablation conditions on detection and repair performance. The TASC-to-uniform condition changes both the pre-trained initialization and the fine-tuning noise policy. TASR-CD affects how the learned representation is converted

into span-aligned repair behavior. The tier, structural, and edit-head components provide more localized security and edit-structure signals.

The ablation labels denote changes to training conditions rather than the physical removal of every corresponding module. In the TASC $\rightarrow$ uniform configuration, the TASC checkpoint is replaced by a checkpoint obtained from uniform pre-training. The tier-dependent noise multipliers used during TASR-CD are also set to 1.0. Pre-training is therefore retained in this configuration.

In the –TASR-CD configuration, Type A repair and BCE updates are retained. Type B calibration, contrastive supervision, and the operation, slot-count, and deletion losses are disabled. In –Edit auxiliary heads, the three head modules remain available for diagnostic evaluation, but their loss weights are set to zero. The resulting near-random Operation Type Accuracy reflects the absence of edit-operation supervision.

Table 5.6: Detection performance ablation.

Configuration	**Det. F1**	**Precision**	**Recall**	**HN-F1**	**Det. F1 Δ**
Full Model	**88.97%**	**88.24%**	**89.71%**	**81.35%**	**–**
– TASC	74.51%	60.92%	95.89%	39.06%	−16.2%
– TASR-CD	82.65%	71.73%	97.50%	59.55%	−7.1%
– Tier embedding	88.79%	87.54%	90.07%	80.46%	−0.2%
– Structural threat features	88.77%	87.53%	90.04%	80.29%	−0.2%
– Edit auxiliary heads	88.86%	87.67%	90.09%	80.35%	−0.1%

TAIPEI TECH

Table 5.7: Repair performance ablation.

Configuration	Span Acc.	Full Seq. %PP	Span Acc. Δ	Full Seq. %PP Δ	Op Type Acc.
Full Model	**34.59%**	**15.22%**	–	–	**95.87%**
− TASC	21.46%	5.91%	−38.0%	−61.2%	96.05%
− TASR-CD	30.11%	12.91%	−13.0%	−15.2%	5.44%
− Tier embedding	33.06%	13.71%	−4.4%	−9.9%	96.05%
− Structural threat features	32.81%	14.81%	−5.1%	−2.7%	96.05%
− Edit auxiliary heads	26.06%	8.61%	−24.7%	−43.4%	5.23%

- **Impact of the Tier-Aware TASC Condition.**

  Removing the tier-aware components of TASC replaces the TASC-derived checkpoint and tier-aware fine-tuning corruption with their uniform counterparts while retaining the pre-training procedure. This condition produces the largest performance decline across both tasks. Detection F1 decreases to 74.51%, while HN-F1 falls to 39.06%. Span Accuracy and Full Seq. %PP decrease by 38.0% and 61.2%, respectively. Op Type Accuracy remains at 96.05%, indicating that operation-type prediction is not the main source of the repair degradation.

  Because the initialization and fine-tuning corruption are changed together, this ablation measures their combined contribution. The results show that the complete tier-aware training condition provides an important foundation for both detection and repair. Table 5.1 separately reports the effect of the masking policy during pre-training.

- **Impact of TASR-CD Fine-tuning.**

  Removing TASR-CD reduces Detection F1 by 7.1% to 82.65% and HN-F1 by 21.8 percentage points to 59.55%. Precision decreases to 71.73%, while Recall increases to 97.50%. This configuration disables Type B calibration, contrastive supervision, and the edit auxiliary losses together. The resulting Precision–Recall shift is consistent with the Recall Drift pattern described in Chapter 3. Because these components are disabled simultaneously, the result reflects the combined TASR-CD condition rather than the isolated effect

of Type B routing. Op Type Accuracy also falls to 5.44%, consistent with the edit auxiliary losses being disabled.

- **Impact of Edit Auxiliary Heads.**

  Removing edit auxiliary heads leaves Detection F1 nearly unchanged (-0.1%) but reduces Span Accuracy by 24.7% and Full Seq. %PP by 43.4%, with Op Type Accuracy falling from 95.87% to 5.23%. Because the head modules are retained while their losses are set to zero, the result isolates the contribution of edit auxiliary supervision rather than the architectural presence of the heads.

- **Impact of Tier Embedding and Structural Threat Features.**

  Removing either component changes Detection F1 by only 0.2% and Span Accuracy by 4–5% under the current evaluation metrics. These changes are modest relative to the TASC and edit-supervision ablations. The present metrics therefore provide limited evidence for their individual contribution, even though the components encode different token-level and function-level signals. Given the single seed, the small detection differences should not be treated as a reliable ordering between the two components.

The largest changes occur under the –TASC and –TASR-CD conditions and when edit auxiliary supervision is disabled. Tier embedding and structural threat features produce smaller changes under the current evaluation. Overall, the ablation results support the staged architecture, but do not show a clear difference between the effects of tier embedding and structural threat features.

TAIPEI TECH

Chapter 6 Conclusion and Future Work

This chapter summarizes the findings of this thesis and outlines directions for future research. The central problem addressed in this work is the difficulty of performing vulnerability detection and repair on C/C++ source code when vulnerable and repaired functions remain highly similar. In this setting, a model must not only identify whether a function is vulnerable, but also preserve fine-grained security distinctions and predict localized repair edits. The proposed ThreatDiffusion framework was designed to address this problem through security-aware data representation, tier-aware pre-training, span-aligned repair supervision, and joint detection-repair calibration.

6.1 Conclusion

This thesis investigated a tier-aware masked diffusion framework for joint vulnerability detection and repair. The study was motivated by the limitations of existing SSDLC tools: static analysis tools often provide limited contextual repair capability, while neural repair models may overlook local security features such as API risk, boundary conditions, and control-flow structure. The empirical results show that modeling these security-related signals together with localized repair supervision provides a practical basis for vulnerability analysis and localized repair prediction in C/C++ code.

The main findings of this thesis are summarized as follows:

1. **Security-Aware Data Representation.**

   This thesis constructed a preprocessing pipeline that converts raw VulGate samples into model-ready records containing token context, security tier labels, vulnerability labels, repair supervision, and AST/CFG-derived threat features. This representation allows the model to use more than surface token patterns. By incorporating API risk, taint-related indicators, guard conditions, and control-flow context, the framework provides a structured basis for distinguishing code fragments that may appear lexically similar but differ in security behavior.

2. **Tier-Aware Security Curriculum.**

   The TASC pre-training stage addresses the sparsity of security-critical tokens by directing masking and edit-oriented corruption toward higher-risk regions. The TASC evaluation results (Section 5.2) show that tier-aware masking improves token reconstruction compared with uniform masking while maintaining comparable vulnerability-detection performance. The –TASC condition also produces the largest decline in detection and repair, especially on Hard-Negative samples. Because this condition changes both the initialization and fine-tuning noise, the decline cannot be attributed to tier-aware masking alone.

3. **Span-Aligned Repair with Contrastive Detection.**

   This thesis introduced TASR-CD to connect localized repair prediction with vulnerability detection. Through Token-Slot Alignment, variable-length repairs involving replacement, insertion, and deletion are represented in a fixed aligned coordinate system. This design makes it possible to supervise span-level repair while preserving the encoder-only diffusion architecture. The TASR-CD results (Section 5.3) show that the model obtains measurable repair performance under both token-level and exact-match metrics, including Span Token Accuracy, Aligned-record %PP, Full Token Accuracy, and Full Sequence %PP. The gap between Full Token Accuracy and Span Token Accuracy also shows why localized repair metrics are needed when most surrounding tokens remain unchanged.

4. **Hard-Negative-Aware Joint Detection and Repair Validation.**

   The Type A/Type B sample-routing mechanism provides separate repair-oriented and detection-only training paths during fine-tuning. Hard-Negative evaluation is important because VulGate contains many vulnerable–repaired pairs with cosine similarity of at least 0.90. Chapter 5 reports an overall detection F1-score of 88.97% and an HN-F1 of 81.35% on this subset. The ablation results show lower HN-F1 under the –TASC and –TASR-CD conditions. These results are consistent with contributions from security-aware pre-training and the combined TASR-CD fine-tuning objective.

Overall, this thesis shows that vulnerability detection and repair should not be treated as isolated tasks when vulnerable and repaired code are highly similar. Detection requires a calibrated boundary between security states, whereas repair requires precise token choices inside localized edit regions. ThreatDiffusion connects these two requirements through a shared security-aware representation and a span-aligned fine-tuning procedure. Although exact patch generation remains challenging, the reported exact-match metrics measure agreement with reference repair tokens rather than semantic correctness. The results support further study of security tier information, structured program features, and repair-aware diffusion training for Shift Left vulnerability analysis and localized repair prediction. All reported results are obtained from a single training run with a fixed random seed and should be interpreted accordingly.

6.2 Future Work

Although ThreatDiffusion provides a unified framework for vulnerability detection and localized repair, its repair quality is currently assessed through internal ablations rather than direct comparison with external repair systems, and several directions remain open for future research.

- **Feedback-Guided Repair Generation and Standardized Evaluation.**

  Future work can incorporate additional feedback signals to improve the reliability of generated patches. Compiler diagnostics, static analysis results, unit tests, and security-specific validators could be used to rerank or filter candidate repairs produced by the model. This direction is also important for evaluation. The Aligned-record %PP used in this thesis measures exact agreement across all valid positions in the aligned repair record, excluding inactive tail padding. Its evaluation scope differs from the full-patch exact-match metrics commonly used by sequence-to-sequence repair models. Future studies can establish more standardized repair evaluation protocols, including tokenization normalization, full-function reconstruction, formatting normalization, compilation checks, and test-based validation. Such protocols would make results more comparable across

different repair paradigms and would better reflect the practical reliability of generated patches.

- **Long-Term Generalization to Emerging CVEs and Coding Practices.**

  This thesis evaluates ThreatDiffusion on VulGate, a large-scale C/C++ vulnerability dataset collected from real security commits. However, vulnerability patterns, library usage, and coding practices continue to evolve over time. Future work can therefore evaluate the framework on newly disclosed CVEs, recently introduced APIs, emerging vulnerability classes, and changing project-specific coding styles. Such long-term evaluation would help determine whether the learned security representations remain stable when the model is exposed to future vulnerability patterns rather than only historical data. It would also provide a basis for updating tier mappings, API risk sets, and structural threat features as software ecosystems change.

- **Developer-Centered SSDLC Integration and Explainability.**

  Another important direction is the integration of ThreatDiffusion into developer-facing SSDLC workflows. The framework could be extended into IDE plugins, code review assistants, CI/CD checks, or pre-commit hooks, where vulnerability detection and repair suggestions are provided before deployment. For practical use, the model should also provide interpretable evidence rather than only a vulnerability label or a patch. The tier labels, API risk scores, AST/CFG-derived features, and localized repair spans used in this thesis can serve as the basis for explanations that identify high-risk tokens, missing guard conditions, or control-flow contexts related to the predicted vulnerability. Such explanations would help developers inspect model outputs and decide whether a generated repair should be accepted, revised, or rejected.

In summary, future work should move ThreatDiffusion from benchmark evaluation toward feedback-guided repair, long-term security generalization, and developer-centered deployment. These directions would further connect automated vulnerability repair with practical Shift Left security workflows.

TAIPEI TECH

References

[1] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. *Commun. ACM*, 68, 2, (Jan. 2025), 96–105. https://doi.org/10.1145/3610721.

[2] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. 2024. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Softw. Engg.*, 30, 2, (Dec. 2024). https://doi.org/10.1007/s10664-024-10590-1.

[3] Rafiq Ahmad Khan, Siffat Ullah Khan, Habib Ullah Khan, and Muhammad Ilyas. 2022. Systematic literature review on security risks and its practices in secure software development. *IEEE Access*, 10, 5456–5481. https://doi.org/10.1109/ACCESS.2022.3140181.

[4] Gareth Bennett, Tracy Hall, Steve Counsell, Emily Winter, and Thomas Shippey. 2024. Do developers use static application security testing (SAST) tools straight out of the box? a large-scale empirical study. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (ESEM '24). Association for Computing Machinery, Barcelona, Spain, 454–460. ISBN: 9798400710476. https://doi.org/10.1145/3674805.3690750.

[5] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. An empirical study of static analysis tools for secure code review. (2024). https://arxiv.org/abs/2407.12241 arXiv: 2407.12241 [cs.SE].

[6] Lan Zhang, Qingtian Zou, Anoop Singhal, Xiaoyan Sun, and Peng Liu. 2024. Evaluating large language models for real-world vulnerability repair in C/C++ code. In *Proceedings of the 10th ACM International Workshop on Security and Privacy Analytics* (IWSPA '24). Association for Computing Machinery, Porto, Portugal, 49–58. ISBN: 9798400705564. https://doi.org/10.1145/3643651.3659892.

[7] Lucas Bastos Germano, Ronaldo Ribeiro Goldschmidt, Ricardo Choren Noya, and Julio Cesar Duarte. 2025. A systematic review on detection, repair, and explanation of vulnerabilities in source code using large language models. *IEEE Access*, 13, 192263–192293. https://doi.org/10.1109/ACCESS.2025.3631363.

[8] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2* (NIPS'14). MIT Press, Montreal, Canada, 3104–3112. https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.

[9] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (ESEC/FSE 2022). Association for Computing Machinery, Singapore, Singapore, 935–947. ISBN: 9781450394130. https://doi.org/10.1145/3540250.3549098.

TAIPEI TECH

[10] Weishi Wang, Yue Wang, Shafiq Joty, and Steven C. H. Hoi. 2023. RAP-Gen: retrieval-augmented patch generation with CodeT5 for automatic program repair. (2023). https://arxiv.org/abs/2309.06057 arXiv: 2309.06057 [cs.SE].

[11] Aytuğ Onan and Hesham A. Alhumyani. 2025. CodeDiffuSe: a masked diffusion framework for structure-aware code completion and repair. *Journal of King Saud University Computer and Information Sciences*, 37, 230. https://doi.org/10.1007/s44443-025-00237-6.

[12] Shansan Gong, Ruixiang Zhang, Huangjie Zheng, Jiatao Gu, Navdeep Jaitly, Lingpeng Kong, and Yizhe Zhang. 2025. DiffuCoder: understanding and improving masked diffusion models for code generation. (2025). https://arxiv.org/abs/2506.20639 arXiv: 2506.20639 [cs.CL].

[13] Rijha Safdar, Danyail Mateen, Syed Taha Ali, M. Umer Ashfaq, and Wajahat Hussain. 2025. Data and context matter: towards generalizing AI-based software vulnerability detection. (2025). https://arxiv.org/abs/2508.16625 arXiv: 2508.16625 [cs.CR].

[14] Niklas Risse and Marcel Böhme. 2024. Uncovering the limits of machine learning for automatic vulnerability detection. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (SEC '24). USENIX Association, Philadelphia, PA, USA. ISBN: 978-1-939133-44-1. https://www.usenix.org/conference/usenixsecurity24/presentation/risse.

[15] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability detection with code language models: how far are we? In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (ICSE '25). IEEE Press, Ottawa, Ontario, Canada, 1729–1741. ISBN: 9798331505691. https://doi.org/10.1109/ICSE55347.2025.00038.

[16] Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. 2024. Simple and effective masked diffusion language models. (2024). https://arxiv.org/abs/2406.07524 arXiv: 2406.07524 [cs.CL].

[17] Idan Achituve, Idit Diamant, Arnon Netzer, Gal Chechik, and Ethan Fetaya. 2024. Bayesian uncertainty for gradient aggregation in multi-task learning. In *Proceedings of the 41st International Conference on Machine Learning* (Proceedings of Machine Learning Research). Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp, (Eds.) Vol. 235. PMLR, (21–27 Jul 2024), 117–134. https://proceedings.mlr.press/v235/achituve24a.html.

[18] Zhipeng Zhou, Liu Liu, Peilin Zhao, and Wei Gong. 2025. Injecting imbalance sensitivity for multi-task learning. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*. James Kwok, (Ed.) Main Track. International Joint Conferences on Artificial Intelligence Organization, (Aug. 2025), 7236–7244. https://doi.org/10.24963/ijcai.2025/805.

TAIPEI TECH

[19] Yuxuan Song, Zheng Zhang, Cheng Luo, Pengyang Gao, Fan Xia, Hao Luo, Zheng Li, Yuehang Yang, Hongli Yu, Xingwei Qu, Yuwei Fu, Jing Su, Ge Zhang, Wenhao Huang, Mingxuan Wang, Lin Yan, Xiaoying Jia, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Yonghui Wu, and Hao Zhou. 2025. Seed diffusion: a large-scale diffusion language model with high-speed inference. (2025). https://arxiv.org/abs/2508.02193 arXiv: 2508.02193 [cs.CL].

[20] Software Engineering Institute. 2016. SEI CERT C Coding Standard. Carnegie Mellon University. Accessed: 2026-06-01. (2016). https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

[21] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA. ISBN: 0321486811.

[22] Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*. Association for Computing Machinery, Urbana-Champaign, Illinois, 1–19. ISBN: 9781450373869. https://doi.org/10.1145/800028.808479.

[23] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2019. Representation learning with contrastive predictive coding. (2019). https://arxiv.org/abs/1807.03748 arXiv: 1807.03748 [cs.LG].

[24] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised contrastive learning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (NIPS '20). Curran Associates Inc., Vancouver, BC, Canada. ISBN: 9781713829546. https://proceedings.neurips.cc/paper/2020/hash/d89a66c7c80a29b1bdbab0f2a1a94af8-Abstract.html.

[25] David A. Wheeler. 2024. Flawfinder. Accessed: 2026-06-01. (2024). https://github.com/david-a-wheeler/flawfinder.

[26] GitHub. 2024. CodeQL documentation. Accessed: 2026-06-01. (2024). https://codeql.github.com/docs/.

[27] Meta Platforms, Inc. 2024. Infer static analyzer. Accessed: 2026-06-01. (2024). https://github.com/facebook/infer.

[28] Rebecca L. Russell, Louis Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated vulnerability detection in source code using deep representation learning. (2018). https://arxiv.org/abs/1807.04320 arXiv: 1807.04320 [cs.LG].

[29] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: a deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium (NDSS)* (NDSS 2018). Internet Society. https://doi.org/10.14722/ndss.2018.23158.

TAIPEI TECH

[30] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2022. SySeVR: a framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19, 04, (July 2022), 2244–2258. https://doi.ieeecomputersociety.org/10.1109/TDSC.2021.3051525.

[31] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Curran Associates Inc., Red Hook, NY, USA, 10197–10207. https://papers.neurips.cc/paper_files/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html.

[32] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: are we there yet? *IEEE Transactions on Software Engineering*, 48, 9, 3280–3296. https://doi.org/10.1109/TSE.2021.3087402.

[33] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: a transformer-based line-level vulnerability prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, 608–620. https://doi.org/10.1145/3524842.3528452.

[34] Ruitong Liu, Yanbin Wang, Haitao Xu, Jianguo Sun, Fan Zhang, Peiyue Li, and Zhenhao Guo. 2025. Vul-LMGNNs: fusing language models and online-distilled graph neural networks for code vulnerability detection. (2025). https://doi.org/10.1016/j.inffus.2024.102748 arXiv: 2404.14719 [cs.CR].

[35] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: context-aware vulnerability detection through code property graph-guided large language models. In *Proceedings of the 34th USENIX Conference on Security Symposium* (SEC '25). USENIX Association, Seattle, WA, USA. ISBN: 978-1-939133-52-6. https://www.usenix.org/conference/usenixsecurity25/presentation/lekssays.

[36] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: a generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38, 1, 54–72. https://doi.org/10.1109/TSE.2011.104.

[37] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. 2023. APR4Vul: an empirical study of automatic program repair techniques on real-world java vulnerabilities. *Empirical Softw. Engg.*, 29, 1, (Dec. 2023). https://doi.org/10.1007/s10664-023-10415-7.

[38] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 691–701. https://doi.org/10.1145/2884781.2884807.

[39] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. Is self-repair a silver bullet for code generation? In *Proceedings of the Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=y0GJXRungR.

TAIPEI TECH

[40] Quanjun Zhang, Chunrong Fang, Bowen Yu, Weisong Sun, Tongke Zhang, and Zhenyu Chen. 2024. Pre-trained model-based automated software vulnerability repair: how far are we? *IEEE Transactions on Dependable and Secure Computing*, 21, 4, 2507–2525. https://doi.org/10.1109/TDSC.2023.3308897.

[41] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2023). Association for Computing Machinery, Seattle, WA, USA, 1282–1294. ISBN: 9798400702211. https://doi.org/10.1145/3597926.3598135.

[42] Xin Zhou, Kisub Kim, Bowen Xu, DongGyun Han, and David Lo. 2024. Out of sight, out of mind: better automatic vulnerability repair by broadening input ranges and sources. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 1071–1083. https://doi.org/10.1145/3597503.3639222.

[43] Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. 2024. Unifying the perspectives of NLP and software engineering: a survey on language models for code. *Transactions on Machine Learning Research*. https://openreview.net/forum?id=hkNnGqZnpa.

[44] Qiushi Sun, Zhirui Chen, Fangzhi Xu, Kanzhi Cheng, Chang Ma, Zhangyue Yin, Jianing Wang, Chengcheng Han, Renyu Zhu, Shuai Yuan, Qipeng Guo, Xipeng Qiu, Pengcheng Yin, Xiaoli Li, Fei Yuan, Lingpeng Kong, Xiang Li, and Zhiyong Wu. 2025. A survey of neural code intelligence: paradigms, advances and beyond. (2025). https://arxiv.org/abs/2403.14734 arXiv: 2403.14734 [cs.SE].

[45] Mayank Agarwal, Yikang Shen, Bailin Wang, Yoon Kim, and Jie Chen. 2024. Structured code representations enable data-efficient adaptation of code language models. (2024). https://arxiv.org/abs/2401.10716 arXiv: 2401.10716 [cs.CL].

[46] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, 590–604. https://doi.org/10.1109/SP.2014.44.

[47] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212, 112031. https://www.sciencedirect.com/science/article/pii/S0164121224000748.

[48] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: a pre-trained model for programming and natural languages. (2020). https://arxiv.org/abs/2002.08155 arXiv: 2002.08155 [cs.CL].

[49] 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. 2021. GraphCodeBERT: pre-training code representations with data flow. (2021). https://arxiv.org/abs/2009.08366 arXiv: 2009.08366 [cs.SE].

TAIPEI TECH

[50] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: unified cross-modal pre-training for code representation. (2022). https://arxiv.org/abs/2203.03850 arXiv: 2203.03850 [cs.CL].

[51] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. 2021. Structured denoising diffusion models in discrete state-spaces. In *Proceedings of the 35th International Conference on Neural Information Processing Systems* (NIPS '21). Curran Associates Inc., Red Hook, NY, USA. ISBN: 9781713845393. https://proceedings.neurips.cc/paper_files/paper/2021/hash/958c530554f78bcd8e97125b70e6973d-Abstract.html.

[52] Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K. Titsias. 2024. Simplified and generalized masked diffusion for discrete data. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (NIPS '24). Curran Associates Inc., Vancouver, BC, Canada. ISBN: 9798331314385. https://proceedings.neurips.cc/paper_files/paper/2024/hash/bad233b9849f019aead5e5cc60cef70f-Abstract-Conference.html.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (NIPS'17). Curran Associates Inc., Long Beach, California, USA, 6000–6010. ISBN: 9781510860964. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[54] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, 35, 2, (Jan. 2026). https://doi.org/10.1145/3747588.

[55] Mukul Singh, José Cambronero, Sumit Gulwani, Vu Le, Carina Negreanu, and Gust Verbruggen. 2023. CodeFusion: a pre-trained diffusion model for code generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Houda Bouamor, Juan Pino, and Kalika Bali, (Eds.) Association for Computational Linguistics, Singapore, (Dec. 2023), 11697–11708. https://aclanthology.org/2023.emnlp-main.716/.

[56] Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023. ContraBERT: enhancing code pre-trained models via contrastive learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2476–2487. https://doi.org/10.1109/ICSE48619.2023.00207.

[57] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2018. Focal loss for dense object detection. (2018). https://arxiv.org/abs/1708.02002 arXiv: 1708.02002 [cs.CV].

[58] Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. 2020. Gradient surgery for multi-task learning. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (NIPS '20). Curran Associates Inc., Vancouver, BC, Canada. ISBN: 9781713829546.

TAIPEI TECH

[59] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. 2020. On layer normalization in the transformer architecture. In *Proceedings of the 37th International Conference on Machine Learning* (ICML'20). JMLR.org. https://proceedings.mlr.press/v119/xiong20b.html.

[60] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Katrin Erk and Noah A. Smith, (Eds.) Association for Computational Linguistics, Berlin, Germany, (Aug. 2016), 1715–1725. https://aclanthology.org/P16-1162/.

[61] Max Brunsfeld et al. 2024. tree-sitter/tree-sitter. Software repository. Accessed: 2026-06-01. (2024). https://github.com/tree-sitter/tree-sitter.

[62] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2024. Code Llama: open foundation models for code. (2024). https://arxiv.org/abs/2308.12950 arXiv: 2308.12950 [cs.CL].

[63] Marton Havasi, Brian Karrer, Itai Gat, and Ricky T. Q. Chen. 2025. Edit flows: flow matching with edit operations. (2025). https://arxiv.org/abs/2506.09018 arXiv: 2506.09018 [cs.LG].

[64] Boyang Yang, Zijian Cai, Fengling Liu, Bach Le, Lingming Zhang, Tegawendé F. Bissyandé, Yang Liu, and Haoye Tian. 2025. A survey of LLM-based automated program repair: taxonomies, design paradigms, and applications. (2025). https://arxiv.org/abs/2506.23749 arXiv: 2506.23749 [cs.SE].

[65] D. Marjamäki. 2024. Cppcheck: a tool for static C/C++ code analysis. Accessed: 2026-06-01. (2024). http://cppcheck.sourceforge.net/.

TAIPEI TECH

Appendix A: Mathematical Notation

This appendix summarizes the mathematical notation used in Chapters 2, 3, and 5.

A.1 Notation Used in Chapter 2

Notation	Definition	Eq.
$x_0^{(i)},\ x_t^{(i)}$	*Token states.* Clean token and its corrupted state at position i and time t.	2.1
$q(x_t^{(i)} \mid x_0^{(i)})$	*Token-level forward kernel.* Absorbing-state forward distribution for the token at position i.	2.1
$q(x_t \mid x_0)$	*Sequence-level forward kernel.* Absorbing-state forward distribution for the complete sequence.	2.1
m	*Mask state.* Absorbing `[MASK]` state used by masked discrete diffusion.	2.1
$\alpha_t,\ \alpha_s$	*Retention schedule.* Clean-token retention probability at times t and s.	2.1
α'_t	*Schedule derivative.* Derivative of the retention schedule with respect to time.	2.2
$\mathcal{L}_{\text{MDLM}}(\theta)$	*MDLM loss.* Rao–Blackwellized continuous-time masked-diffusion objective parameterized by θ.	2.2
$\mu_\theta^{(i)}(x_t, t)[x_0^{(i)}]$	*Token prediction.* Probability assigned to the clean target token at position i.	2.2

Continued.

Continued.

Notation	Definition	Eq.
$p_\theta(x_s \mid x_t)$	*Reverse transition.* Learned SUBS transition from state x_t to x_s, where $0 \le s \le t$.	2.3
$\mu_\theta(x_t, t)$	*Token distribution.* Model-predicted clean-token probability vector used when the current state is masked.	2.3
$\mathbf{e}_m$	*Mask one-hot vector.* One-hot vector associated with the absorbing mask state.	2.3
γ_t	*Corruption schedule.* Complementary masking or corruption probability.	2.4
$\gamma'_t,\ \gamma'_t/\gamma_t$	*Corruption weight.* Derivative of the corruption schedule and the corresponding MDLM timestep weight.	2.4
$\mathcal{L}_{\text{InfoNCE}}$	*InfoNCE loss.* Contrastive objective that favors the positive representation over the negative set.	2.5
$z_i,\ z_i^+$	*Contrastive pair.* Anchor representation and its positive representation.	2.5
$z_j,\ \mathcal{N}(i)$	*Negative set.* Negative representation and the negative set associated with anchor i.	2.5
τ	*Contrastive temperature.* Temperature controlling the concentration of the similarity distribution.	2.5

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
$\mathcal{L}_{\text{supcon}}$	*Supervised contrastive loss.* Batch-averaged contrastive objective using multiple same-label positives.	2.6
$z_i,\ z_p,\ z_a$	*Sample representations.* Anchor, positive, and non-anchor comparison representations.	2.6
$\mathcal{I},\ \mathcal{P}(i),\ a$	*Supervised contrastive indices.* Anchor set, same-label positive set for anchor i, and comparison index over samples other than the anchor.	2.6
$\mathcal{L}_{\text{focal}}$	*Focal loss.* Confidence-reweighted classification loss.	2.7
p_t	*Class probability.* Model probability assigned to the correct class.	2.7
α_f	*Balance coefficient.* Coefficient used to balance classes in Focal Loss.	2.7
γ_f	*Focusing parameter.* Exponent controlling the suppression of well-classified examples.	2.7

A.2 Notation Used in Chapter 3

Notation	Definition	Eq.
$x,\ \mathcal{V},\ V,\ L$	*Token sequence.* BPE-tokenized function, UniXcoder vocabulary, vocabulary size $V = \lvert\mathcal{V}\rvert = 51{,}416$, and fixed sequence length $L = 1{,}024$.	3.2

Continued.

Continued.

Notation	Definition	Eq.
E_{token}	*Token embedding stream.* Non-contextual lexical and positional representation produced by the UniXcoder embedding module; $E_{\text{token}} \in \mathbb{R}^{L\times 768}$.	3.2
$\text{Embed}_{\text{UniX}}(\cdot)$	*UniXcoder embedding.* Word, position, and token-type embedding module with layer normalization.	3.2
hasGuard(`api`)	*Guard predicate.* Boolean indicating whether an API argument has a recognized validation check.	3.3
Args(`api`)	*API arguments.* Set of arguments of the analyzed API call.	3.3
Sanitized($v, L_{\texttt{api}}$)	*Sanitization predicate.* True when argument v, or an alias, is checked before the API call at $L_{\texttt{api}}$.	3.3
S	*Threat vector.* Function-level concatenation of API, CFG, taint, and contextual security summaries; $S \in \mathbb{R}^{64}$.	3.4
S_{API}	*API features.* Normalized API counts, loop participation, and guard coverage; $S_{\text{API}} \in \mathbb{R}^{20}$.	3.4
S_{CFG}	*CFG features.* Normalized node, edge, and cyclomatic-complexity statistics; $S_{\text{CFG}} \in \mathbb{R}^{15}$.	3.4
S_{taint}	*Taint features.* Taint sources, sanitized sources, and length aliases; $S_{\text{taint}} \in \mathbb{R}^{15}$.	3.4

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
S_{ctx}	*Context features.* Use-after-free, double-free, and unchecked arithmetic indicators; $S_{\text{ctx}} \in \mathbb{R}^{14}$.	3.4
$E^{(i)}_{\text{threat}}$	*Threat embedding.* Projected function-level threat summary broadcast to token position i; $E^{(i)}_{\text{threat}} \in \mathbb{R}^{64}$.	3.5
$W_{\text{th}},\ b_{\text{th}}$	*Threat projection parameters.* Trainable affine parameters used to project the threat vector.	3.5
r_i	*Token risk.* Intrinsic risk category assigned from API and arithmetic-operator membership.	3.6
$\mathcal{A}_0,\ \mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{O}_{\text{arith}}$	*Risk sets.* Critical-, conditional-, and medium-risk API sets and arithmetic operators lacking range validation.	3.6
g_i	*Guard indicator.* Indicator that the API associated with token i has a recognized guard.	3.7
$\tau^{(i)}$	*Security tier.* Per-token tier derived from intrinsic risk and guard status.	3.7
$E^{(i)}_{\text{tier}}$	*Tier embedding.* Trainable 64-dimensional embedding of the tier assigned to position i.	3.8
E_{tier}	*Tier stream.* Stacked per-token tier embeddings; $E_{\text{tier}} \in \mathbb{R}^{L\times 64}$.	3.8
E_{input}	*Unified input.* Fused token, tier, threat, and timestep streams; $E_{\text{input}} \in \mathbb{R}^{L\times 960}$.	3.9

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
W_{fuse}	*Fusion projection.* Trainable 960×960 projection matrix.	3.9
$b_{\text{fuse}},\ 1_L$	*Fusion bias.* Trainable bias and the length-L all-ones vector used to broadcast it over the sequence.	3.9
GELU, LN	*Fusion operations.* GELU activation and Layer Normalization.	3.9
t	*Diffusion time.* Continuous diffusion timestep; zero is clean and one is unit masking in the base schedule.	3.10
$e_{\text{time}},\ E_{\text{time}}$	*Timestep representations.* The 64-dimensional MLP output and its sequence-broadcast form $E_{\text{time}} \in \mathbb{R}^{L\times 64}$.	3.10
SinEmb(t), MLP	*Timestep functions.* Sinusoidal encoding of t and the $64 \to 256 \to 64$ projection network.	3.10
$h_0,\ u_\ell,\ h_\ell,\ H$	*Transformer states.* Initial input, post-attention residual state, layer output, and final shared hidden representation.	3.11
Attn, FFN, ℓ	*Transformer operations.* Multi-head self-attention, feed-forward network, and Transformer-layer index.	3.11
$x_0,\ x_t,\ x_s$	*Sequence states.* Clean sequence and its states at diffusion times t and s, where $0 \le s \le t$.	3.12
$q(x_t^{(i)} \mid x_0^{(i)})$	*Forward kernel.* Position-independent absorbing-state corruption kernel.	3.12

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
m	*Mask state.* Absorbing `[MASK]` state used only for diffusion corruption.	3.12
$\gamma_t,\ \gamma_s,\ \gamma'_t$	*Corruption schedule.* Corruption probabilities at times t and s, and the derivative with respect to time.	3.12
$p_\theta(x_s^{(i)} \mid x_t)$	*Reverse transition.* SUBS reverse transition parameterized by θ.	3.13
$\mu_\theta^{(i)}(x_t, t)$	*Token distribution.* Predicted clean-token distribution over the vocabulary at position i.	3.13
$\mathrm{Cat}(x;\pi),\ \ e_a,\ \ e_m$	*Categorical terms.* Categorical distribution evaluation and one-hot vectors for token a and the mask state.	3.13
Δ^{V-1}	*Probability simplex.* Set of categorical probability vectors over the V-token vocabulary.	3.13
$\mathcal{L}_{\mathrm{MDLM}}(\theta)$	*MDLM objective.* Continuous-time MDLM loss under uniformly sampled diffusion time and forward corruption.	3.14
γ'_t/γ_t	*MDLM timestep ratio.* Standard continuous-time weight in corruption-probability notation.	3.14
$\mathcal{I},\ \ a^{(i)}$	*Valid positions.* Non-padding position set and attention-mask value at position i.	3.15
$\bar{h}$	*Pooled representation.* Mean-pooled hidden representation over non-padding positions.	3.15

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
$W_1,\ b_1,\ w_2,\ b_2$	*Detection parameters.* Trainable parameters of the $960 \to 256 \to 1$ detection MLP.	3.15
$\hat{y},\ \hat{p},\ \sigma$	*Detection output.* Vulnerability logit, predicted probability, and sigmoid function.	3.15
$\mathcal{L}_{\text{bce}},\ \ y,\ \eta$	*Detection loss.* Weighted binary cross-entropy, binary vulnerability label, and positive-class weight.	3.15
$p_{\text{mask}}^{(i)}$	*Mask probability.* Probability of masking token i, capped at one.	3.16
$\mu_0,\ \mu_1,\ \mu_2,\ \mu_5$	*Tier multipliers.* Tier-dependent masking multipliers $(1.3, 1.1, 1.0, 0.8)$.	3.16
$\bar{w}(t)$	*Capped timestep weight.* Stabilized MDLM timestep weight capped at 10.	3.17
$\mathcal{L}_{\text{phase1}}(\theta)$	*Phase-1 objective.* Normalized weighted reconstruction loss.	3.18
$\mathcal{B},\ \mathcal{I}_b,\ \ t_b$	*Phase-1 indices.* Mini-batch, non-padding positions of sample b, and its sampled timestep.	3.18
$\kappa_{b,i},\ \ell_{b,i}$	*Phase-1 terms.* Combined position weight and per-position reconstruction loss.	3.18
$w_{\text{tier}}^{(b,i)},\ w_{\text{mask}}^{(b,i)},\ w_{\text{flaw}}^{(b,i)}$	*Phase-1 factors.* Tier, selected-mask-position, and flaw-annotation weights.	3.18

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
$p_{b,i},\ \mathrm{CE}_\epsilon,\ \epsilon,\ \gamma_f$	*Token-loss terms.* Target-token probability, label-smoothed cross-entropy, smoothing coefficient, and Focal Loss focusing parameter.	3.19
$p_{\text{edit}}^{(i)}$	*Edit probability.* Tier-normalized probability of selecting valid position i for structured edit corruption.	3.20
$w_{\text{edit}}(\tau^{(i)})$	*Edit weight.* Tier-dependent weight used in the edit-position distribution.	3.20
$\mathcal{I}$	*Valid edit positions.* Non-padding positions eligible for edit-position sampling.	3.20
$\mathcal{L}_{\text{phase2}}(\theta)$	*Phase-2 objective.* Tier- and edit-position-weighted reconstruction loss with detection BCE.	3.21
$w_{\text{tier}}^{(b,i)},\ w_{\text{edit-pos}}^{(b,i)}$	*Phase-2 weights.* Tier and selected-edit-position weights.	3.21
$w_{\text{flaw}}^{(b,i)},\ \lambda_{\text{bce}}$	*Phase-2 weights.* Flaw-annotation weight and detection-loss coefficient.	3.21
$\mathrm{CE}(\mu_\theta^{(b,i)}, x_0^{(b,i)})$	*Token cross-entropy.* Negative log-probability assigned to the clean target token.	3.21
$\mathcal{D}_{\text{clean}}$	*Clean repair set.* Repair-pair subset for which all rejection flags are false.	3.22

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
$\mathcal{D}_{\text{diff}}$	*Repair candidates.* Complete set of `diff_normal` training samples considered by Repair-Pair Validation.	3.22
x_{before}, x_{after}	*Repair pair.* Vulnerable function and its corresponding repaired function.	3.22
r_{trunc}, r_{dup}	*Rejection flags.* Flags for truncated functions and duplicated signatures.	3.22
r_{same}, r_{noncode}	*Rejection flags.* Flags for identical repair pairs and non-code records.	3.22
C_k	*Span capacity.* Number of aligned positions allocated to diff span k.	3.23
S_k^{vuln}, S_k^{safe}	*Span tokens.* Vulnerable- and safe-side token subsequences of span k.	3.23
o_k	*Diff operation.* Equal, replace, insert, or delete operation assigned to span k.	3.23
$K_{\max}$	*Span cap.* Maximum aligned capacity of one edit span; $K_{\max} = 16$.	3.23
`[PAD]`, `[MASK]`	*Special tokens.* Active `[PAD]` denotes an insertion slot, inactive tail padding is excluded by the attention mask, and `[MASK]` is used only for diffusion corruption.	3.23
Z_{rep}	*Repair logits.* Vocabulary logits produced by the Repair Head.	3.24
$\widehat{P}_{\text{safe}}$	*Repair distribution.* Softmax repair-token probabilities; $\widehat{P}_{\text{safe}} \in \mathbb{R}^{L \times V}$.	3.24
W_{rep}, b_{rep}	*Repair parameters.* Trainable vocabulary projection and bias.	3.24

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
$\widehat{P}_{\text{safe}}^{(i,v)}$	*Repair probability.* Probability assigned to vocabulary token v at aligned position i.	3.24
$\mathcal{L}_{\text{repair}}$	*Repair loss.* Position-weighted negative log-likelihood over the aligned record.	3.25
L_{span}	*Aligned length.* Padded length of the aligned repair record.	3.25
$x_{\text{span_safe}}^{(i)}$	*Repair target.* Reference repair token at aligned position i.	3.25
$\rho^{(i)}$	*Repair weight.* Position-dependent weight used in the repair-token objective.	3.25
$a_{\text{span}}^{(i)}$	*Aligned attention.* One for active aligned positions and zero for inactive tail padding.	3.26
$\mathcal{B}_{\text{span}}$, $\mathcal{B}_{\text{del}}$, $\mathcal{B}_{\text{noise}}$	*Repair sets.* Repair-span positions, retained deletion positions, and additional positions selected by repair noise.	3.26
ρ_{span}, ρ_{noise}, ρ_{ctx}	*Repair weights.* Weights for aligned repair positions, additional noise positions, and remaining context.	3.26
$P_{\text{op}}^{(i)}$	*Operation distribution.* Four-class distribution over equal, replace, insert, and delete.	3.27
$\mathcal{I}_{\text{align}}$	*Aligned positions.* Set of active positions in the aligned record.	3.27
$o^{(i)}$, $\mathcal{L}_{\text{op}}$	*Operation supervision.* Operation label and mean cross-entropy over active aligned positions.	3.27

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
$P_{\text{slot}}^{(i)}$	*Slot-count distribution.* Distribution over insertion counts $\{0, \ldots, K_{\max}\}$.	3.28
$\mathcal{I}_{\text{ins}},\ c^{(i)}$	*Insertion supervision.* Insert-labeled positions and the corresponding insertion-slot count.	3.28
$\mathcal{L}_{\text{slot}}$	*Slot-count loss.* Mean insertion-count cross-entropy.	3.28
$\hat{p}_{\text{del}}^{(i)}$	*Deletion probability.* Predicted deletion probability at vulnerable-source position i.	3.29
$w_{\text{del}},\ b_{\text{del}}$	*Deletion parameters.* Trainable deletion-head weight and bias.	3.29
$H_{\text{vuln}}^{(i)}$	*Vulnerable hidden state.* Hidden representation at source position i in the original vulnerable sequence.	3.29
$y_{\text{del}}^{(i)}$	*Deletion label.* Binary source-coordinate label; one indicates removal from the reference repaired sequence.	3.30
$\mathcal{I}_{\text{vuln}}$	*Valid source positions.* Non-padding positions in the original vulnerable sequence.	3.30
$\mathcal{L}_{\text{del}}$	*Deletion loss.* Mean binary cross-entropy over valid vulnerable-source positions.	3.30
$\mathcal{L}_{\text{contra}}$	*Contrastive loss.* CWE-conditioned temperature-scaled objective for valid vulnerable anchors.	3.31

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
$z_v^{(k)}$, $z_s^{(i)}$	*Sequence representations.* Unit-normalized vulnerable and repaired sequence representations.	3.31
$\mathcal{K}$, $\mathcal{P}_k$	*Contrastive sets.* Valid anchor set and same-CWE positive set for anchor k.	3.31
B, τ_c	*Contrastive parameters.* Mini-batch size and contrastive temperature.	3.31
$\mathcal{L}_A$	*Type-A objective.* Weighted sum of repair, contrastive, operation, slot-count, deletion, and detection losses.	3.32
$\mathcal{L}_B$	*Type-B objective.* Detection-only weighted objective for Type B samples.	3.33
λ_{repair}, λ_{contra}, λ_{op}	*Loss coefficients.* Weights for repair, contrastive, and operation-type supervision.	3.32
λ_{slot}, λ_{del}, λ_{bce}	*Loss coefficients.* Weights for slot-count, deletion, and vulnerability-detection supervision.	3.32

A.3 Notation Used in Chapter 5

Notation	Definition	Eq.
TP, TN, FP, FN	*Detection outcomes.* Numbers of true positives, true negatives, false positives, and false negatives.	5.1–5.4

Continued.

Continued.

Notation	Definition	Eq.
Accuracy	*Detection accuracy.* Proportion of correctly classified samples among all predictions.	5.1
Precision	*Detection precision.* Proportion of truly vulnerable samples among all vulnerable predictions.	5.2
Recall	*Detection recall.* Proportion of vulnerable samples correctly detected.	5.3
F1	*F1-score.* Harmonic mean of Precision and Recall.	5.4
SpanAcc	*Span Token Accuracy.* Token accuracy over evaluated repair-span positions.	5.5
Ω_{span}	*Span positions.* Set of evaluated aligned repair-span positions.	5.5
$y_i,\ \hat{y}_i$	*Span tokens.* Reference repair token and predicted token at aligned position i.	5.5
$\mathbb{I}[\cdot]$	*Indicator function.* Returns one when the stated token or sequence equality holds and zero otherwise.	5.5
FullTokenAcc	*Full Token Accuracy.* Token accuracy over valid positions of the reconstructed full function.	5.6
Ω_{full}	*Full positions.* Set of valid positions in the reconstructed function.	5.6
$\hat{f}_i,\ \ f_i^{\text{safe}}$	*Full-function tokens.* Predicted reconstructed token and corresponding reference safe-function token.	5.6

Continued.

TAIPEI TECH

Continued.

Notation	Definition	Eq.
%PP	*Perfect prediction.* Proportion of evaluated samples whose predicted sequence exactly matches its reference target.	5.7
$\mathcal{D}$	*Evaluation set.* Set of samples used to calculate perfect prediction.	5.7
$\hat{\mathbf{y}}^{(j)},\ \mathbf{y}^{(j)}$	*Sequence pair.* Predicted and reference target sequences of sample j.	5.7
Aligned-record %PP	*Aligned-record exact match.* Exact agreement over all valid positions in the aligned repair record, excluding inactive tail padding.	5.7
Full Sequence %PP	*Full-function exact match.* Exact agreement between the reconstructed function and the reference safe function.	5.7
$\Delta_m(c)$	*Signed ablation change.* Signed percentage change in metric m under ablation condition c.	5.8
$m,\ \ c$	*Metric and condition.* Metric of interest and evaluated ablation condition.	5.8
$m(\text{w/o } c),\ \ m(\text{Full})$	*Ablation values.* Metric value under condition c and the corresponding full-model value.	5.8

TAIPEI TECH