



國立中央大學資訊管理學系 碩士論文口試

Vul LLaMA —

**透過微調 Code LLaMA 自注意力機制以提升
Android 程式碼行級別漏洞定位效能**

**Vul LLaMA: Enhancing Line-Level Vulnerability Localization Performance in
Android Code by Fine-Tuning the Self-Attention Mechanism of Code LLaMA**

研究生：朱珮瑜 指導教授：陳奕明博士 口試日期：2025/07/10



目錄

01

Introduction

02

Related Work

03

Methodology

04

Experiment

05

Conclusion

06

Future Work

01

Introduction

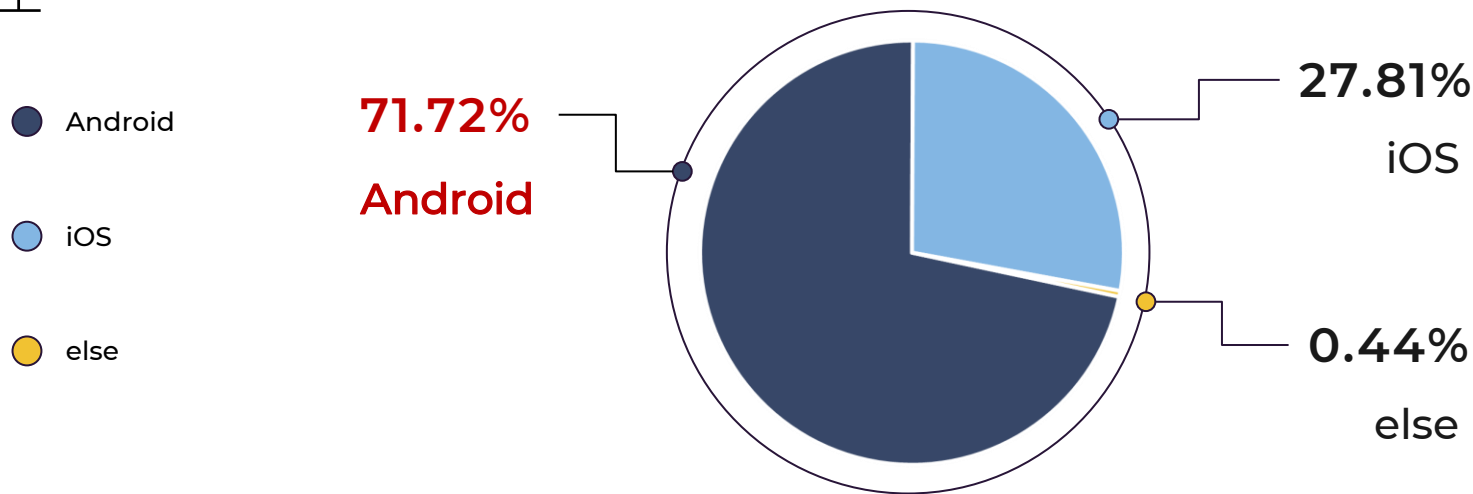
1.1 研究背景

1.2 動機與目的

1.3 研究貢獻

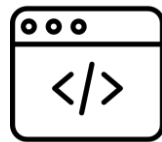
研究背景

- Android 市占率最高使其成為攻擊目標
- 精準的細粒度漏洞定位可以幫助開發者提升修復效率，提升行動裝置安全



研究背景

- 44% 企業認為「資料隱私與安全」是採用 LLM 的最大障礙
- OWASP Top 10 for LLMs 02 : Sensitive Information Disclosure

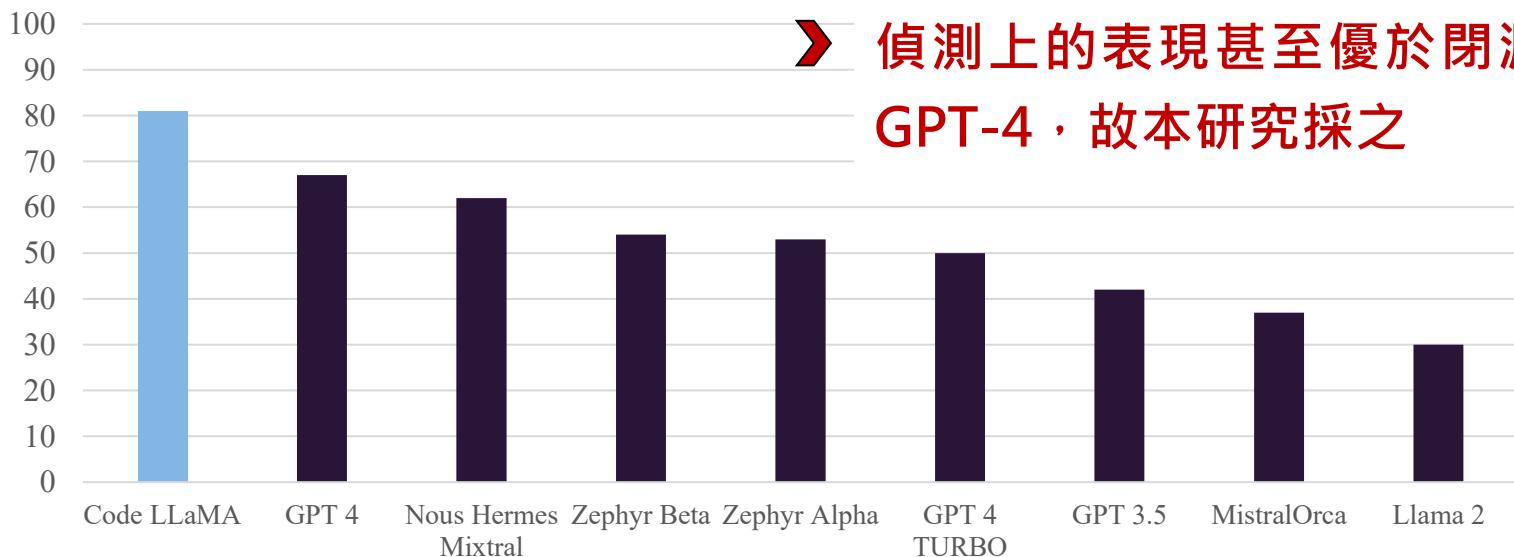


➤ 出於資料安全與外洩風險考量，企業傾向採用「可與外部連線完全隔離的地端 LLM」

Code LLaMA

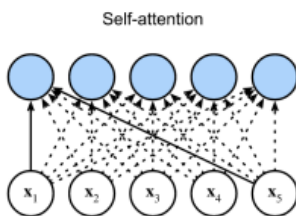
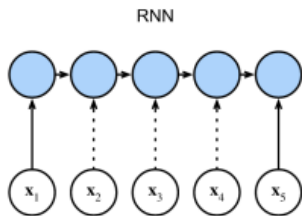
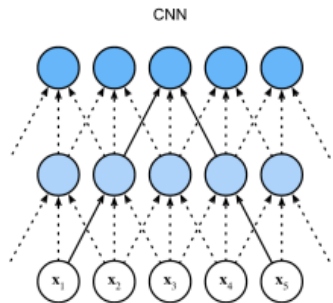
- 建構於 Meta 發布的 LLaMA 2 上，專為程式碼分析與生成任務而生的開源大型語言模型

研究顯示 Code LLaMA 在漏洞偵測上的表現甚至優於閉源的 GPT-4，故本研究採之



Transformer

- 完全依賴**自注意力機制 (Self-Attention)** 捕捉輸入與輸出之間的全局依賴關係
- 克服傳統方法難以兼顧長距離依賴關係與局部語意資訊的限制



雙向自注意力機制
(Bidirectional Attention)

單向自注意力機制
(Causal Attention)

混合自注意力機制
(Cross Attention)

Transformer

雙向自注意力

	<st>	I	am	fine
<st>	0.7	0.1	0.1	0.1
I	0.1	0.6	0.2	0.1
am	0.1	0.3	0.6	0.1
fine	0.1	0.3	0.3	0.3

作法

單向自注意力

	<st>	I	am	fine
<st>	0.7	0.1	0.1	0.1
I	0.1	0.6	0.2	0.1
am	0.1	0.3	0.6	0.1
fine	0.1	0.3	0.3	0.3

512~1024

輸入長度
(tokens)

4k~128k

➤ 有利處理
長程式碼



全局理解



➤ 微調改善

Encoder-Only : BERT

應用

Decoder-Only : GPT、LLaMA

可解釋性 AI (XAI) 技術

Explainability Techniques

Definition

Examples

Post-Hoc Explanations
事後解釋性

Provide explanations on model's output.

SHAP, LIME tools

Intrinsic Interpretability
內在可解釋性

Design LLMs to be inherently interpretable.

Transparent model architecture

Attention-based interpretability

Human-Centered Explanations
以人為本的解釋

Natural language explanations generated by LLMs.

Narrative-based explanations

Natural language generation

內在可解釋性 (Intrinsic Interpretability)

Transparent model architecture

模型本身的結構簡單、易於人類理解，其內部運作方式可以直接被觀察與解釋

Eg : Decision Tree、Linear Regression

Attention-based Interpretability

在使用注意力機制 (Attention Mechanism) 的模型中，觀察模型注意力分數，以了解模型在做出預測時「關注」了輸入的哪些部分

Eg : Transformer-based model

動機與目的

• 動機

- 現今缺乏針對 Android 程式碼進行細粒度漏洞定位的研究
- 雲端 LLM 有資料安全疑慮
- Code LLaMA 受限於單向注意力將影響行級漏洞標記準確度
- 大型語言模型表現優異，但黑盒決策過程缺乏可解釋性

• 目的

微調 Code LLaMA 自注意力機制，**增強全局上下文理解**，以提高行級別漏洞定位準確度，同時**引入可解釋性方法揭示模型決策依據**

研究貢獻

- 本研究提出 Vul LLaMA 架構，透過**兩階段注意力機制優化 Code LLaMA 全局理解**能力，最終針對 Android 程式碼達成更準確的函式級別漏洞檢測與行級別漏洞定位。
- 加入行級別注意力分數（Row-Level Attention Score）計算方法，透過計算並**輸出行級別注意力分數得知模型決策依據**，以達成**內在可解釋性**（Intrinsic Interpretability）。

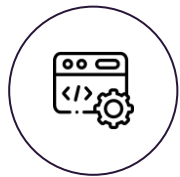
02

Related Work

2.1 現有檢測方法

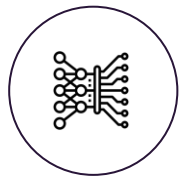
2.2 LineVul

Android 程式碼漏洞檢測方法



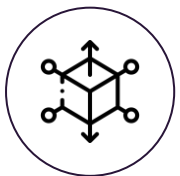
程式分析工具 ➤ 僅能檢測已知漏洞，難以應對未知攻擊

依靠預定義規則檢測程式碼中的潛在漏洞



基於機器學習/深度學習模型 ➤ 難以兼顧全局結構與語意資訊




輸入圖結構或序列特徵以訓練模型自動識別程式碼中的漏洞模式



基於 Transformer 架構 ➤ **Encoder-Only** 架構受輸入長度限制

透過自注意力機制兼顧全局與語意細節資訊

Android 程式碼漏洞檢測方法

		檢測未知漏洞	細粒度	可解釋	Android	代表研究
	靜態分析	✗	✓	N/A	✓	FlowDroid、Fortify
	動態分析	✗	✓	N/A	✓	TaintDroid、DroidScope
	基於圖特徵	✓	✗	✗	✗	Devign、Reveal、IVDetect
	基於序列特徵	✓	✗	✗	✗	Russel et al.、SySeVR、VulDeePecker
	Encoder Only	✓	✓	✓	✗	LineVul (SoTA)
	Decoder Only	✓	✓	✓	✓	Vul LLaMA

➤ **Vul LLaMA** — 一個可以針對 **Android 程式碼** 進行 **細粒度** (行級別) 漏洞定位且具可解釋性的模型

SoTA 模型 — LineVul

	LineVul	Vul LLaMA
基礎模型	Code BERT	Code LLaMA - 7B
架構	Encoder-Only	Decoder-Only
模型參數量	110M	7B
輸入長度限制	512 tokens	4096 tokens
資料集主要語言	C/C++	Java
行級別漏洞定位	✓	✓
可解釋性	✓	✓

03

Methodology

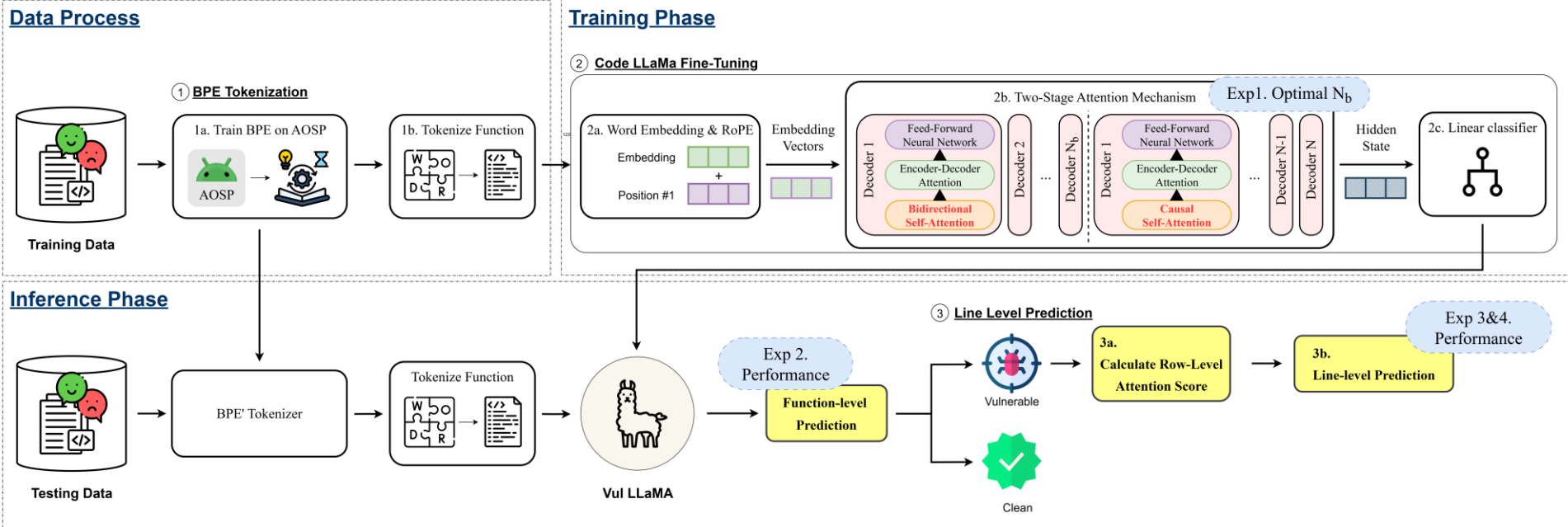
3.1 系統架構圖

3.2 資料處理階段

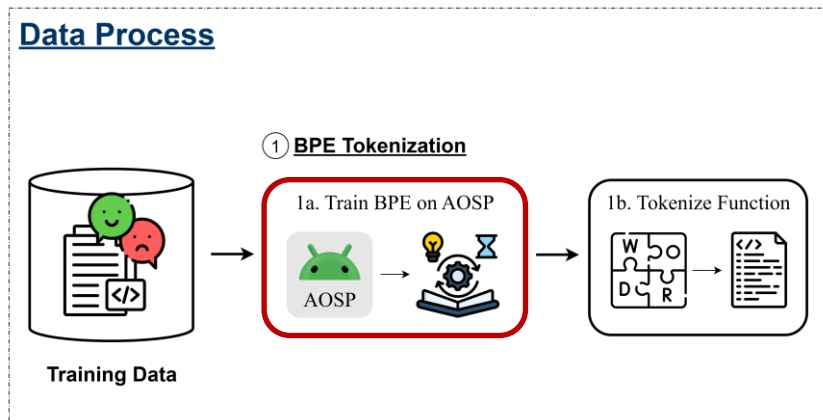
3.3 模型訓練階段

3.4 模型預測階段

系統架構圖



資料處理階段



1. BPE分詞 (Subword Tokenization)

使用 AOSP (Android Open Source Project) 官方 API 做為語料庫，
訓練 BPE 分詞器 ➤ *BPE'*

➤ 使 *BPE'* 比預設分詞器更適合 Android 程式碼分詞

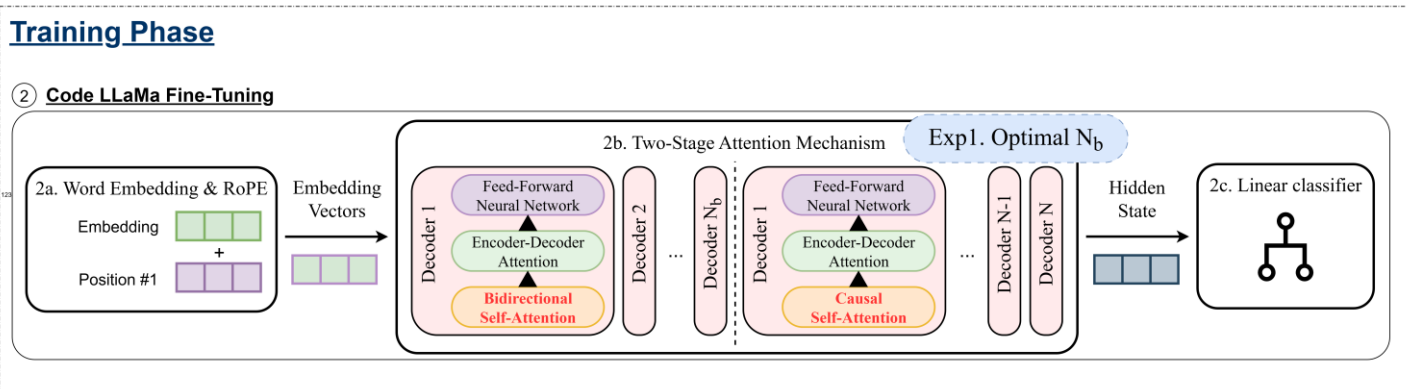
未經訓練的 BPE vs 經過訓練的 BPE'

```
NotificationChannel channel = new NotificationChannel(CHANNEL_ID, "Channel
name", NotificationManager.IMPORTANCE_HIGH);
```

Android 系統中定義通知重要程度，會影響通知的顯示行為的參數

	CHANNEL_ID						IMPORTANCE_HIGH							
BPE	CH	AN	N	EL	_	ID	I	MP	ORT	ANCE	_	H	IG	H
BPE'	CHANNEL		_		ID		IMPORTANCE			_			HIGH	

模型訓練階段

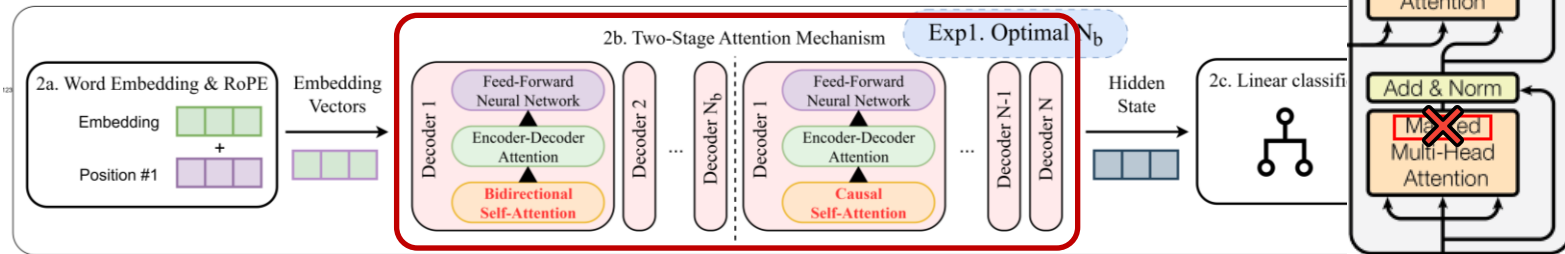


2. 微調 Code LLaMA ➤ *Vul LLaMA*

- 詞嵌入與位置編碼
- 兩階段自注意力機制
- 線性分類器

模型訓練階段

2b. 兩階段自注意力機制



Code LLaMA-7B 中共 32 層 Decoder，將前 N_b 層的遮罩 (Mask) 取消 (即 雙向自注意力)，後 $N - 32$ 層保留 單向自注意力

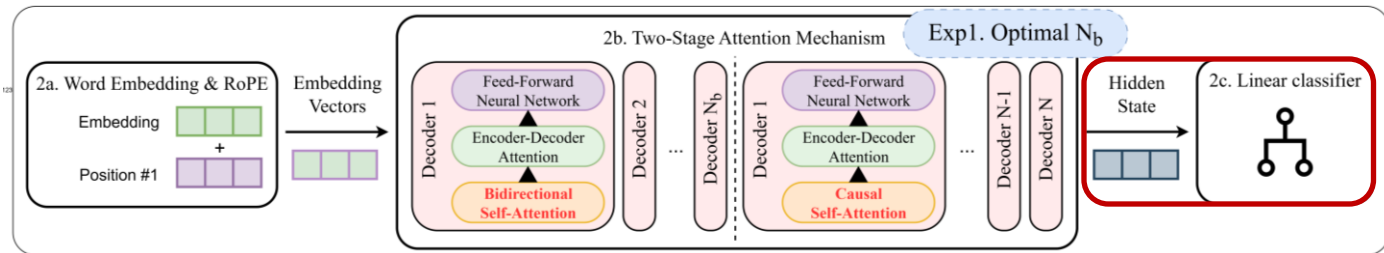
➤ 恢復全局理解能力

➤ 節省計算量

➤ 後續將設計消融實驗確認最佳 N_b 值

模型訓練階段

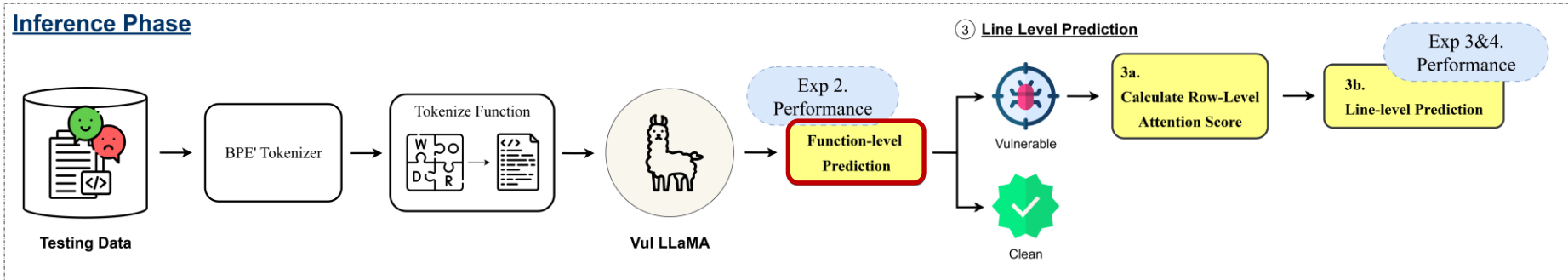
2c. 線性分類器



使用線性分類器 (Linear Classifier) 將 Hidden State H 與模型學習到的權重矩陣 W 相乘，即得二維 logits 分數

$$\text{logits} = W \times H$$

模型預測階段

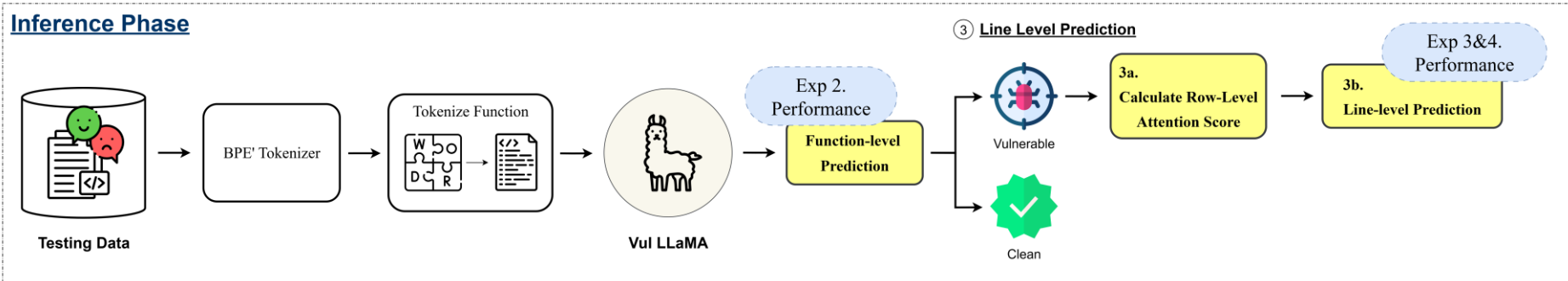


函式級別漏洞檢測

使用 Sigmoid function 正規化模型輸出之二維 logits，即得模型預測該函式為漏洞或非漏洞的機率值

$$Prob = Sigmoid(logits)$$

模型預測階段

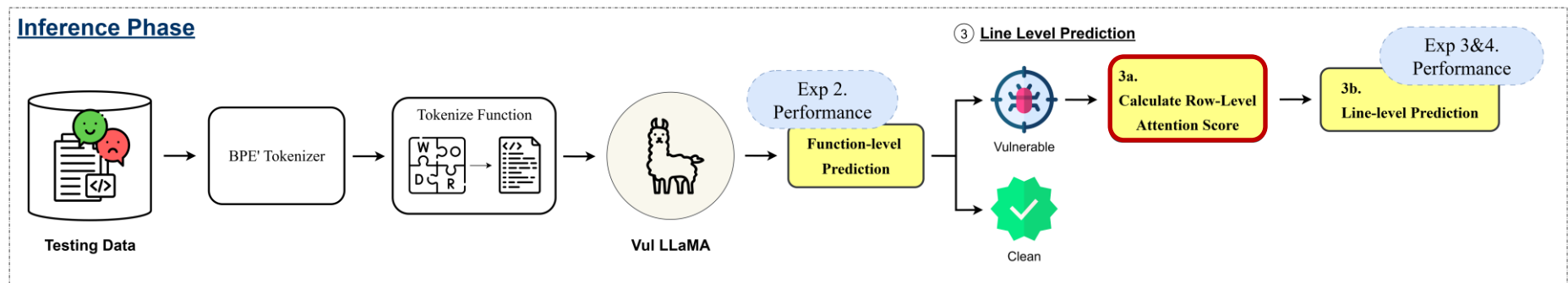


3. 行級別漏洞預測

- 計算行級別注意力分數
- 漏洞行定位

模型預測階段

3a. 計算行級別注意力分數



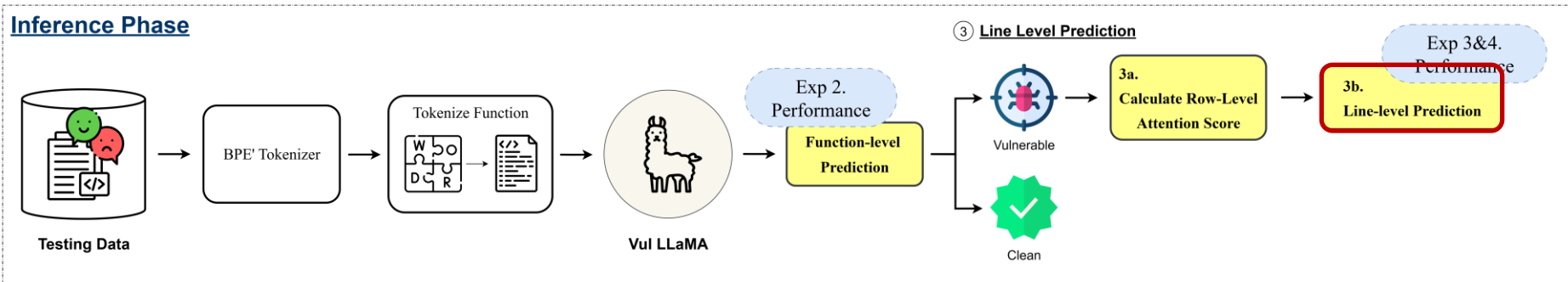
加總第 l 行當中所有 token i 對 token j 的注意力權重

$$S_l = \sum_{i,j \in L_l} A_{ij}$$

227	RefPtr<Uint8Array> dstPixels = copySkImageData(input, info);	0.8
228	if (!dstPixels)	0.3
229	return nullptr;	0.3
230	return newSkImageFromRaster(0.5
231	info, std::move(dstPixels),	0.6
232	static_cast<size_t>(input->width()) * info.bytesPerPixel());	1
233	}	0

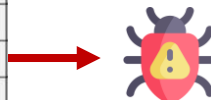
模型預測階段

3b. 漏洞行定位



輸出所有行 l 的 S_l ，得分高者即預測為漏洞行

227	RefPtr<UInt8Array> dstPixels = copySkImageData(input, info);	0.8
228	if (!dstPixels)	0.3
229	return nullptr;	0.3
230	return newSkImageFromRaster(0.5
231	info, std::move(dstPixels),	0.6
232	static_cast<size_t>(input->width()) * info.bytesPerPixel());	1
233	}	0



04

Experiment

4.1 實驗設置

4.2 實驗設計

4.3 實驗一

4.4 實驗二

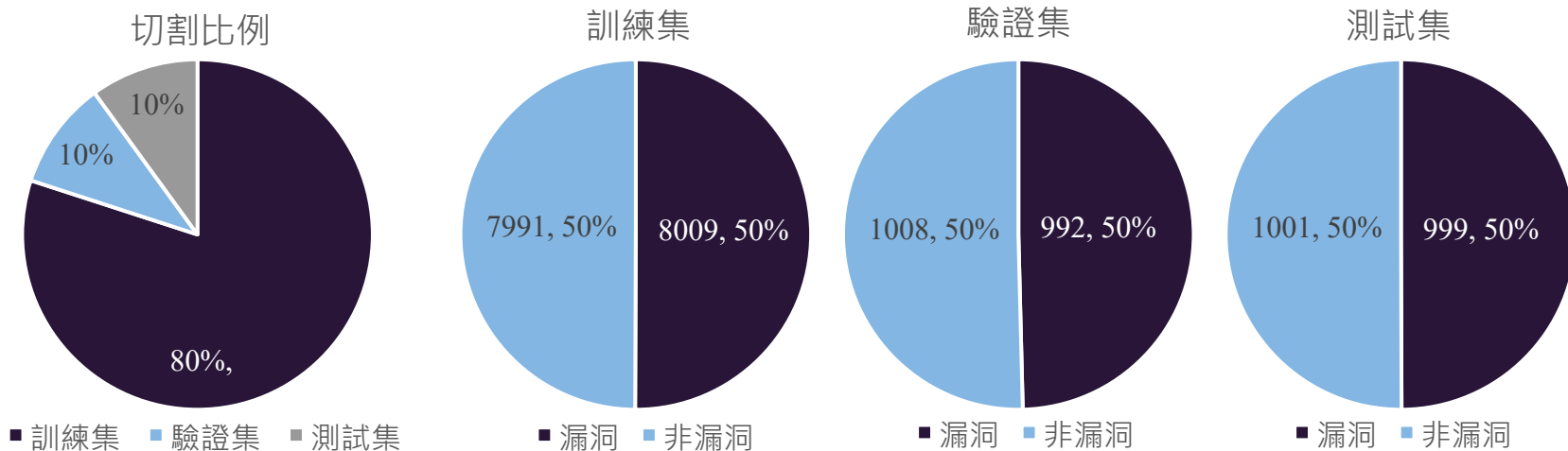
4.5 實驗三

4.6 實驗四

實驗設置

Dataset

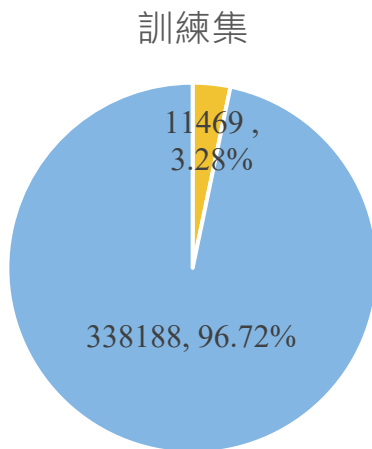
- 資料來源：AndroZoo
- 前處理：使用 MobSF 標記漏洞函式與漏洞行



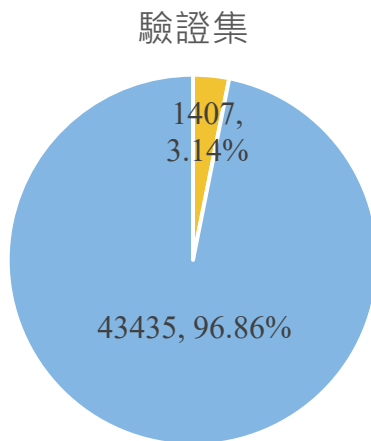
實驗設置

Dataset

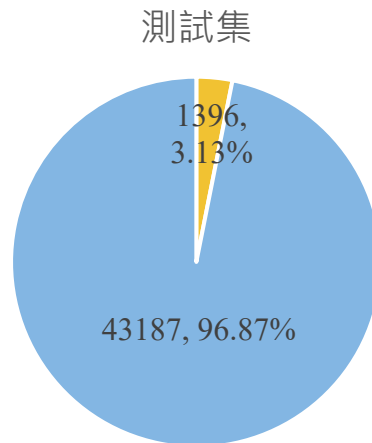
- 漏洞行佔比統計



■ 漏洞行 ■ 非漏洞行



■ 漏洞行 ■ 非漏洞行



■ 漏洞行 ■ 非漏洞行

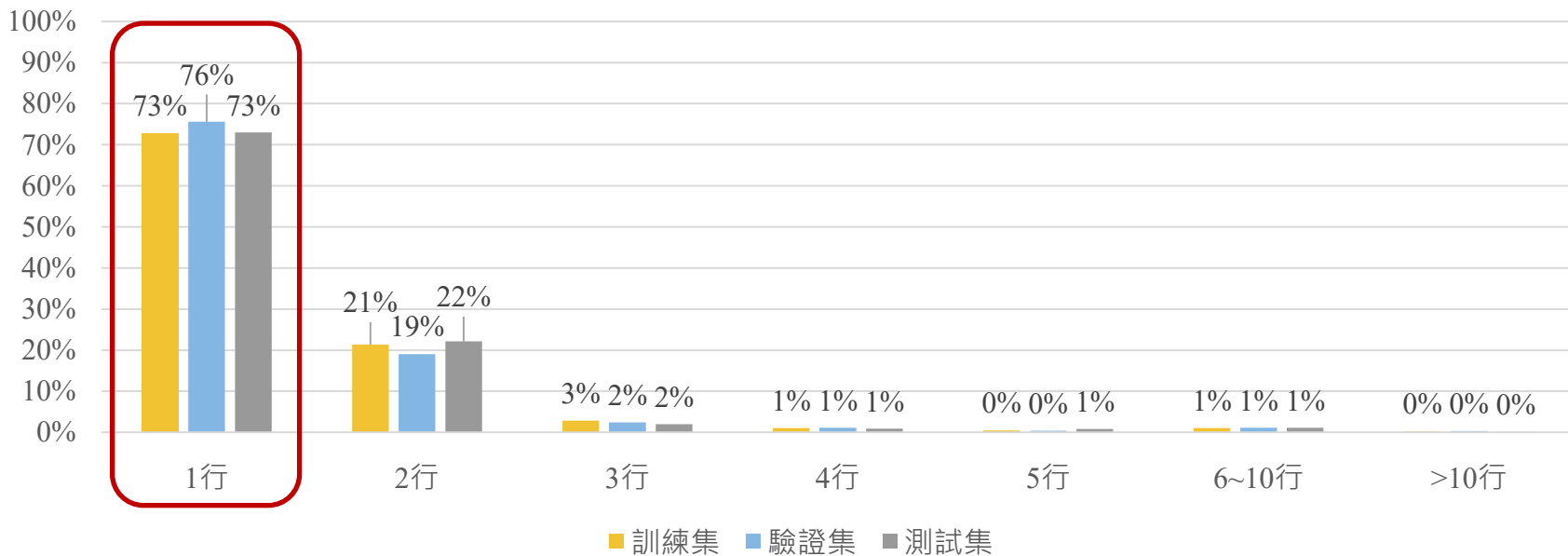
實驗設置

Dataset

➤ 大多數函式僅含
1 行漏洞行

• 漏洞行數統計

子集中各漏洞函式所含漏洞行數統計



實驗設置

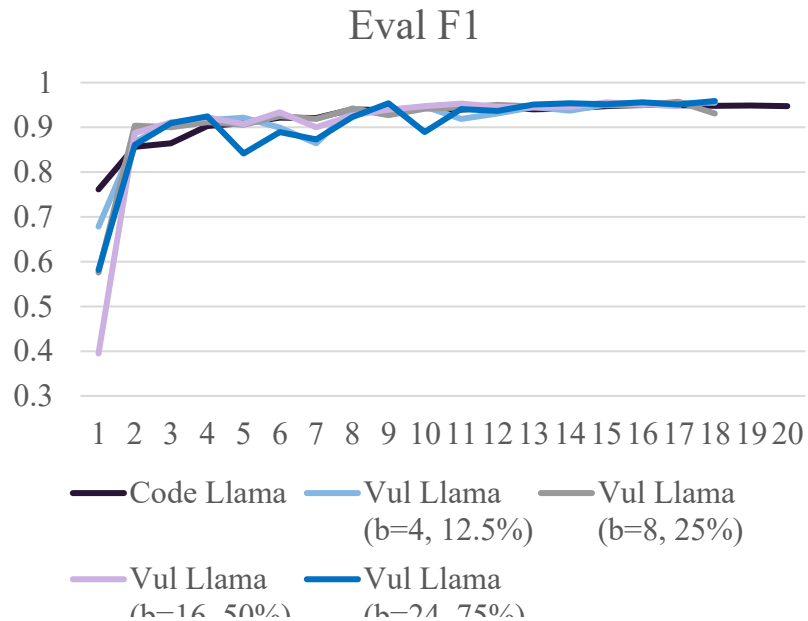
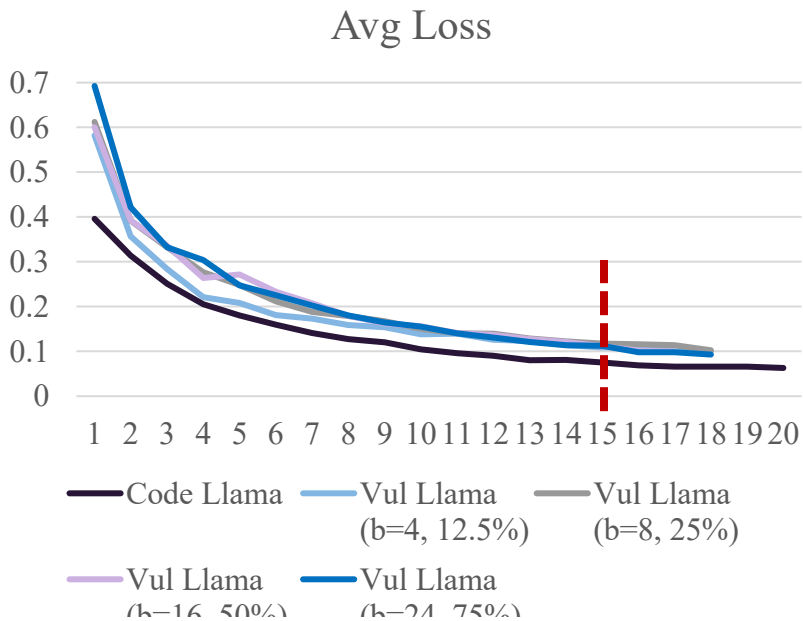
Hardware

本研究租借國網中心晶創25 (Nano 5) 新一代加速器叢集式超級電腦進行實驗

硬體名稱	規格
CPU	Intel Xeon(R) Platinum 8480+(8480CL)
GPU	NVIDIA H100
RAM	120M

實驗設置

Epoch time



實驗設計

1

透過調整雙向自注意力層數，增強 Code LLaMA 的上下文理解能力

2

評估 Vul LLaMA 的函式級別漏洞檢測能力

3

評估 Vul LLaMA 的行級別漏洞定位能力

4

測試 Vul LLaMA 在行級別漏洞定位任務的成本效益

DC

根據行級別輸出結果觀察本研究與 LineVul 的行級別漏洞判斷差異

實驗一

挑選最佳雙向自注意力層數 (N_b)

測試 $N_b = 12.5\%, 25\%, 50\%, 75\%$ 的影響，選擇最佳 N_b 值



Confusion Matrix

- $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$
- $Precision = \frac{TP}{TP+FP}$
- $Recall = \frac{TP}{TP+FN}$
- $F1\ Score = \frac{2 \times Precision \times Recall}{Precision+Recall}$

Predict
Class

Actual Class

	Actual Class	
	Positive	Negative
Positive	TP	FP
Negative	FN	TN

實驗一

挑選最佳雙向自注意力層數 (N_b)

雙向自注意力層數 (N_b)	佔比(%)	函式級別			
		Accuracy	Precision	Recall	F1
0	0	0.5145	0.5074	0.9550	0.6627
4	12.5	0.9600	0.9527	0.9680	0.9603
8	25	0.9475	0.9589	0.9349	0.9468
16	50	0.9555	0.9496	0.9620	0.9557
24	75	0.9515	0.9457	0.9580	0.9518

➤ 加入雙向注意力可大幅提升原始模型表現，且在 $N_b = 4$ 時表現最佳，故後續實驗皆將採用 $N_b = 4$ 作為 Vul LLaMA 模型設定

實驗二

Vul LLaMA 函式級別漏洞檢測能力

比較 Code LLaMA、LineVul、Vul LLaMA 三個模型的函式級別漏洞檢測能力

模型	Accuracy	Precision	Recall	F1
Code LLaMA	0.5145	0.5074	0.9550	0.6627
LineVul	0.975	0.9647	0.986	0.9752
Vul LLaMA	0.9600	0.9527	0.9680	0.9603

➤ Vul LLaMA 加入 4 層雙向自注意力的表現，與使用全雙向自注意力架構的 LineVul 差距約 1%

實驗三

Vul LLaMA 行級別漏洞定位能力

比較 Code LLaMA、LineVul、Vul LLaMA 三個模型的行級別漏洞定位能力



Top-k Accuracy

在排名前 k 高的行當中，命中漏洞的比例



Initial False Alarm (↓)

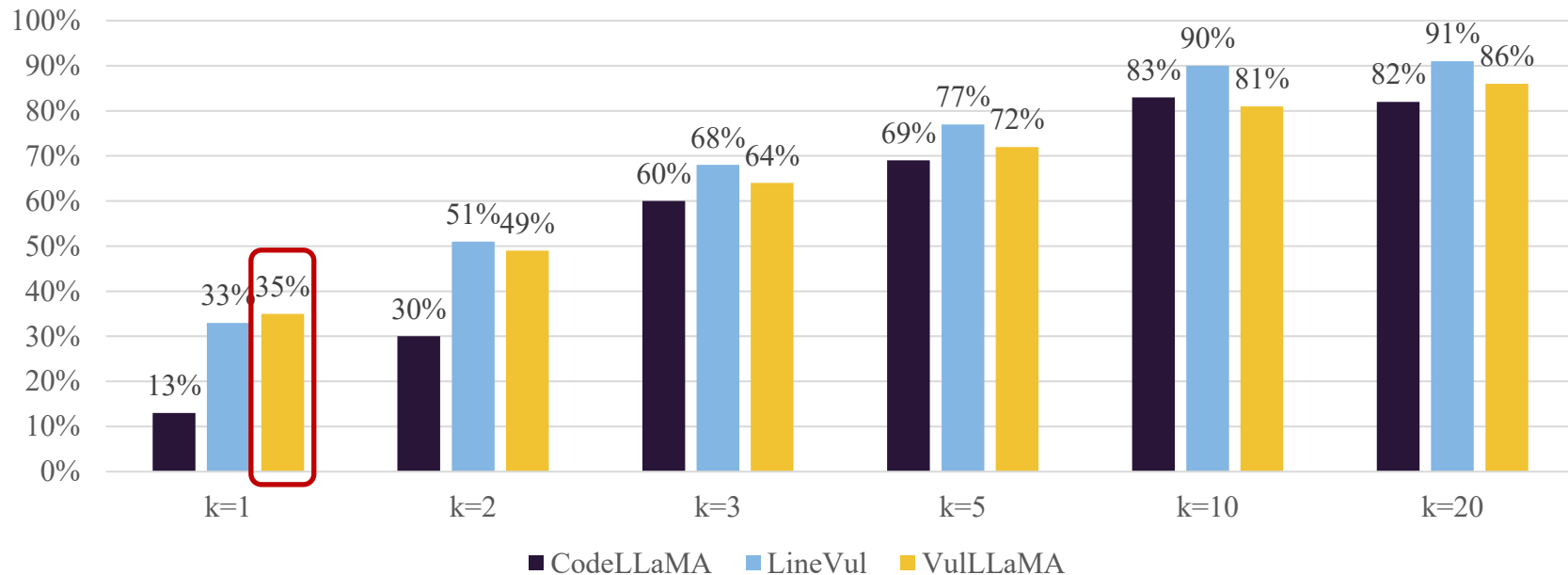
第一次命中真正漏洞行前的平均誤報次數

實驗三

Vul LLaMA 行級別漏洞定位能力

➤ Top1 Accuracy 中
Vul LLaMA 表現最優異

Top k Accuracy



實驗三

Vul LLaMA 行級別漏洞定位能力

模型	IFA (↓)
Code LLaMa	3.29
LineVul	1.86
Vul LLaMA	2.67

- Vul LLaMA 的 IFA 明顯優於原始 Code LLaMA，誤報減少了約 20%
- 儘管 Vul LLaMA 略遜於 LineVul 的 1.86，但這仍是一個不需全雙向架構即可達成的進步

實驗四

M_{VL} 效益成本

比較 M_{VL} 、 M_{CL} 、 M_{LV} 於行級別漏洞定位的成本效益



Effort@20%Recall (↓)

measures the effort (in LOC)
needed to locate 20% of actual
vulnerable lines.

「找到 20% 的漏洞，需要花多少
effort (掃多少行)？」



Recall@1%LOC

measures the proportion of
actual vulnerable lines found
within top 1% LOC of test set.

「在前 1% 的程式碼行當中，能找到
多少比例的真實漏洞？」

實驗四

範例

行數	程式碼	分數	Ground Truth
1	<code>int uid = getUserId();</code>	0.48	✗
2	<code>string name = inputName();</code>	0.55	✗
3	<code>string path = "/data/user/";</code>	0.70	✗
4	<code>path += name;</code>	0.80	✗
5	<code>File file = new File(path);</code>	0.91	✓
6	<code>if (file.exists()) {</code>	0.60	✗
7	<code> log(path);</code>	0.40	✗
8	<code> writeLog(name);</code>	0.50	✓
9	<code> file.delete();</code>	0.87	✗
10	<code>}</code>	0.83	✓

Effort@20%Recall

Step1. 根據模型分數進行排名

排名	行數	分數	Ground Truth
1	5	0.91	✓
2	9	0.87	✗
3	10	0.83	✓
4	4	0.80	✗
5	3	0.70	✗
6	6	0.60	✗
7	2	0.55	✗
8	8	0.50	✓
9	1	0.48	✗
10	7	0.40	✗

Step2. 計算 Effort@20%Recall

- Ground Truth 分析
 - 總漏洞數 = 3
 - $3 \times 20\% = 0.67 \rightarrow 1$
- 計算 Effort
 - 總行數 = 10
 - $1/10 = 0.1 = 10\%$

Recall@1%LOC

Step1. 根據模型分數進行排名

排名	行數	分數	Ground Truth
1	5	0.91	✓
2	9	0.87	✗
3	10	0.83	✓
4	4	0.80	✗
5	3	0.70	✗
6	6	0.60	✗
7	2	0.55	✗
8	8	0.50	✓
9	1	0.48	✗
10	7	0.40	✗

Step2. 計算 Recall@1%LOC

- 決定1%的程式碼行數
 - 此 function 總行數=10
 - $10 \times 1\% = 0.1 \rightarrow 1$
- 1%LOC中檢查到的漏洞數
 - 抓到1個漏洞
 - 總漏洞數=3
 - $1/3 = 0.33 = 33\%$

實驗四

M_{VL} 效益成本

比較 M_{VL} 、 M_{CL} 、 M_{LV} 於行級別漏洞定位的成本效益

模型	Effort@20%Recall (↓)	Recall@1%LOC
Code LLaMA	7.21%	1.86%
LineVul	1.65%	14.1%
Vul LLaMA	2.98%	8.02%

討論

根據行級別輸出結果觀察本研究與 LineVul 的行級別漏洞判斷差異

1. 因 tokenizer 導致輸入階段就丟失漏洞行的比例

LineVul	Vul LLaMA
3.51%	0%

➤ Vul LLaMA 可以確保關鍵漏洞行不被遺漏

討論

根據行級別輸出結果觀察本研究與 LineVul 的行級別漏洞判斷差異

2. 在兩模型都命中的漏洞函式中，指出漏洞行的比例

LineVul	Vul LLaMA	同時
15.91%	27.73%	56.36%

➤ 在已命中漏洞行時，Vul LLaMA 更能快速指出真正的漏洞行

05

Conclusion

5.1 總結

5.2 結論

總結

1. 本研究提出結合雙向自注意力的 Vul LLaMA 架構，在函式與行級別漏洞檢測任務中均顯著提升了 Code LLaMA 的表現
2. 雖然 Vul LLaMA 整體表現略低於 SoTA 模型 — LineVul，但在 Top-1 Accuracy、完整性（0% 漏行率）等指標上仍具優勢。此外，Vul LLaMA 在模型皆命中的漏洞樣本中有 27.73% 的行排序優於 LineVul，顯示模型能更有效地將真正的漏洞行排序在前，協助縮短人工檢查與回應時間。

結論

- 綜合分析結果，Vul LLaMA 採用的「少量雙向注意力層」設計，兼具效能與實用性，在保有原架構推論特性的同時，也顯著強化了定位精準度與資料完整性，提供未來 LLM-based 漏洞檢測模型一項具參考價值的設計方向。

06

Future Work

未來研究

本研究觀察到同類型漏洞的程式碼，常出現具語意關聯的關鍵子詞。未來可進一步蒐集「漏洞指標子詞」，並於計算行級別分數時設計加權機制，提升模型對關鍵語意片段的辨識能力，達到更精確的行級別漏洞定位。

CWE-926 (Improper Enforcement of WebView Security)

```
webView.getSettings().setJavaScriptEnabled(true);
```

```
webView.addJavascriptInterface(...);
```

```
webView.loadUrl("http://...");
```

感謝聆聽

敬請指導

Appendix

補充 - 未經訓練的 BPE vs 經過訓練的 BPE'

```
private void checkPermission() {  
    if (ContextCompat.checkSelfPermission(this,  
        Manifest.permission.READ_CONTACTS) !=  
        PackageManager.PERMISSION_GRANTED) {  
        // ask again  
    }  
},
```

Android 系統檢查是否擁有該權限的 API

	checkPermission		PackageManager		PERMISSION_GRANTED						
BPE	_check	Permission	_Package	Manager	PER	MI	SSION	_	GR	ANT	ED
BPE'	_checkPermission		_PackageManager		PERMISSION		_		GRANTED		

補充 - 未經訓練的 BPE vs 經過訓練的 BPE'

↪ 用以取得裝置唯一識別碼 (IMEI) 的 Android API

```
String deviceId = telephonyManager.getDeviceId();
```

```
SharedPreferences.Editor editor = prefs.edit();  
editor.putString("key", "value").apply();
```

↪ 將 key-value 對存入偏好設定中的 API

	getDeviceId			Prefs		putString	
BPE	get	Device	Id	_pre	fs	put	String
BPE'	getDeviceId			_prefs		putString	

比較 - 使用不同方式計算注意力分數

Sum

$$S_l = \sum_{i,j \in L_l} A_{ij}$$

Mean

$$S_l = \frac{1}{|L_l|} \sum_{i,j \in L_l} A_{ij}$$

Square Mean

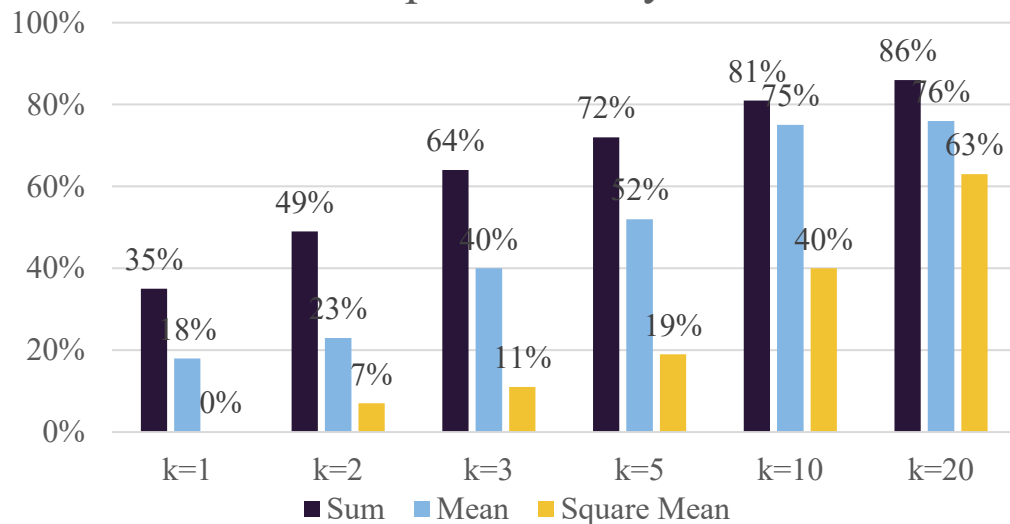
$$S_l = \frac{1}{|L_l|^2} \sum_{i,j \in L_l} A_{ij}$$

比較 - 使用不同方式計算注意力分數

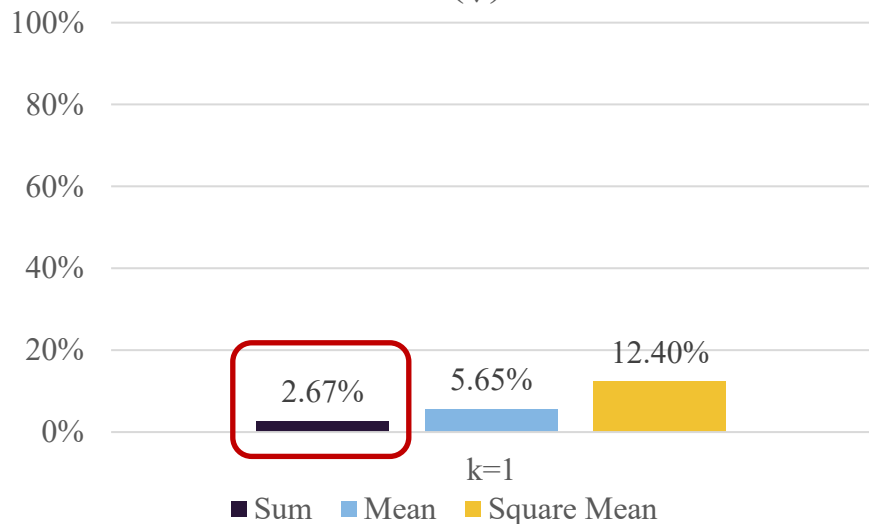
- 行級別定位精準度

➤ 直接加總表現最佳

Top k Accuracy



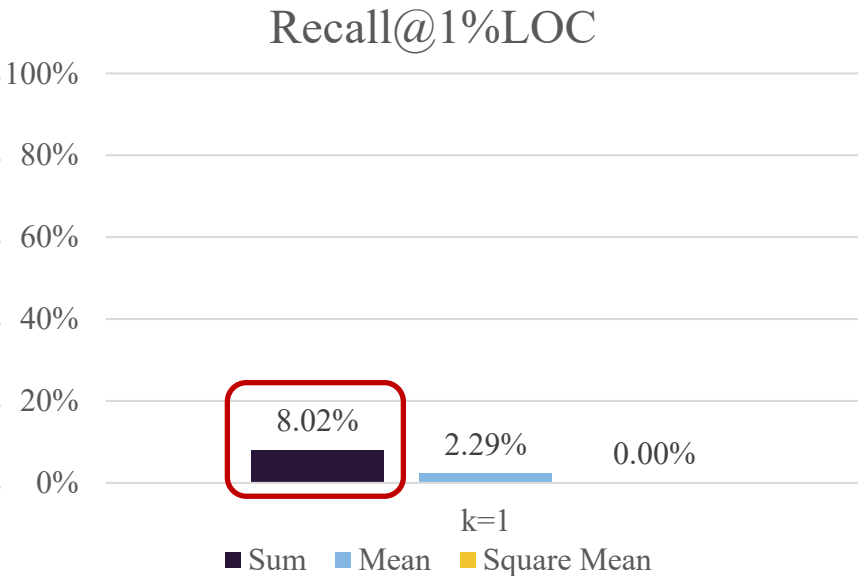
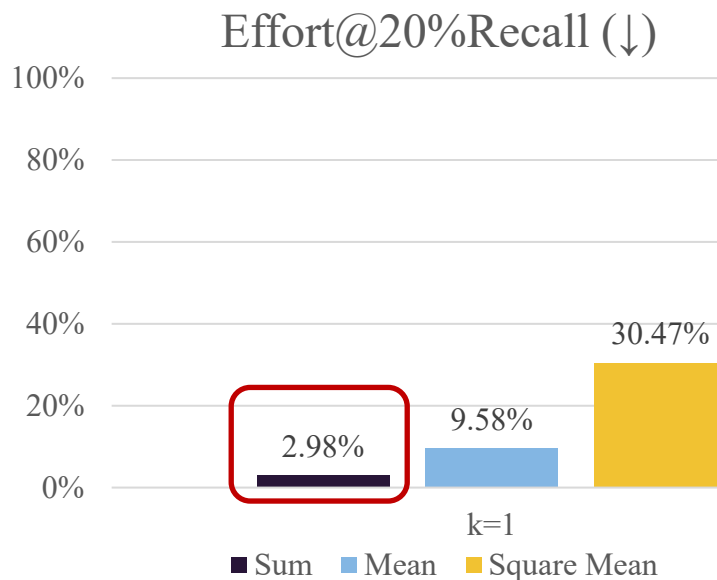
IFA(↓)



比較 - 使用不同方式計算注意力分數


- 行級別漏洞定位成本效益

➤ 直接加總表現最佳



指標 — Top-k Accuracy

- Top-k Accuracy Confusion Matrix

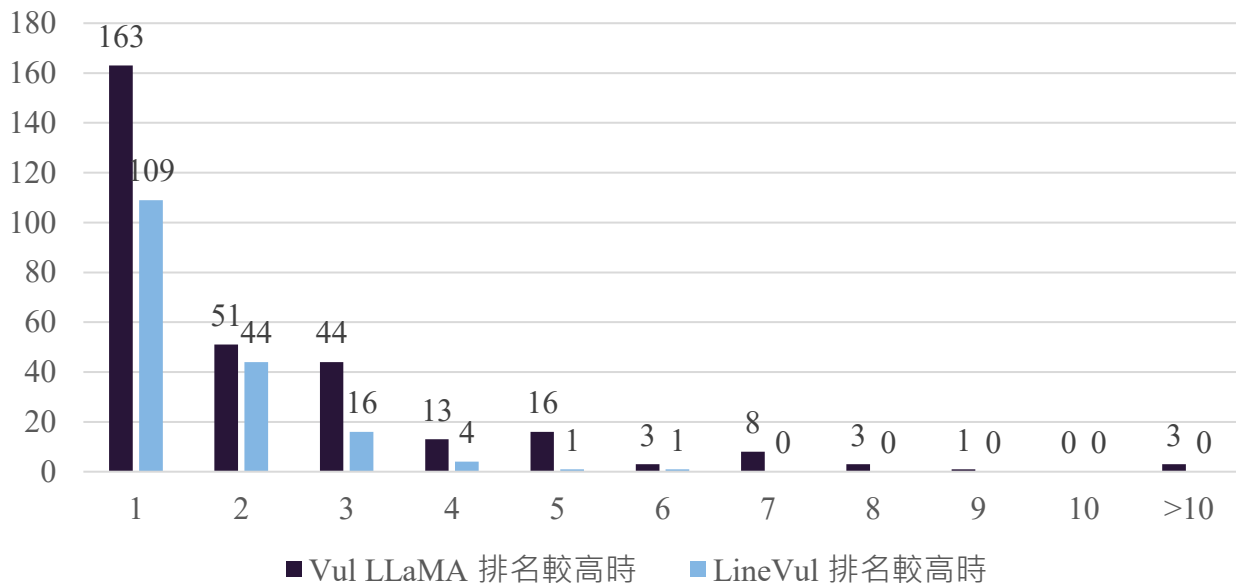
Ground Truth	Prediction	}	歸類
	18.19.20		TP
	18.30.40		
	10.11.12	→	FP

No Buggy line	11	→	FN
---------------	----	---	----

討論

根據行級別輸出結果觀察本研究與 LineVul 的行級別漏洞判斷差異

3. 比對方早指出漏洞行時的排名差異



討論

根據行級別輸出結果觀察本研究與 LineVul 的行級別漏洞判斷差異

4. 兩模型同時定位出漏洞行時的漏洞排名分布

