Friends or Foes? Combining Static Analysis Tools and LLMs for Vulnerability Detection

Rafael Ramires*, Sarmad Bashir†, Abbas Khan†, Mehrdad Saadatmand†, Ibéria Medeiros*
*LASIGE, DI, Faculdade de Ciências, Universidade de Lisboa, Portugal
†RISE Research Institutes of Sweden, Västerås, Sweden
rframires@fc.ul.pt, sarmad.bashir@ri.se, abbas.khan@ri.se, mehrdad.saadatmand@ri.se, ivmedeiros@fc.ul.pt

***Abstract*—Industrial products and devices (e.g., vehicles, drones) are broadly accessed and managed using web applications. These applications have been a major concern for enterprises, as they are the preferred targets of attackers due to persistent vulnerabilities in their code. To address vulnerabilities, Static analysis tools (SASTs) have been widely used for detection, alongside the growing trend of employing prompt-engineered large language models (LLMs). Although they have proven useful for detection, both techniques tend to generate false positives (FPs), thereby unnecessarily increasing manual effort in the search for non-existent vulnerabilities; moreover, SASTs tend to miss vulnerabilities. In contrast, fine-tuned LLMs have proven effective at reasoning and classification tasks, but often require expensive training with balanced corpora. In this paper, we study SASTs, both types of LLMs, and their combination to improve overall vulnerability detection in web applications. We tested two modern SAST tools and two LLM models, across seven datasets for SQL injection (SQLi) vulnerability detection. Our findings reveal that combining the results of multiple solutions can improve vulnerability detection. The best combination integrates both LLMs and a SAST, where i) the fine-tuned LLM, together with the SAST, reduces FPs, mainly produced by the prompt-engineering LLM, and ii) both LLMs overcome SAST's limitation of missing vulnerabilities. On average, the F1-Score increases by 17-60% when SASTS and LLMs are combined. In particular, it can improve from 6% (with a standalone solution) to ≈100% when LLMs are combined with SASTs.**

***Index Terms*—Vulnerability detection, Static analysis, Fine-tuning and Prompt-engineering LLMs, Software security**

I. Introduction

Our societies are centered around services and products supported by web applications across diverse industries (e.g., transportation, banking) and business criticality. These applications enable the monitoring and control of primary daily-life infrastructures (e.g., power and water plants) and, conversely, the management of technological infrastructures that support and deliver services and products to end users (e.g., cloud services, hospitals). Despite their widespread use across contexts, one common feature is their connection to databases, which enables the storage and manipulation of data whose privacy should not be compromised. However, they have been at the centre of enterprises' concern, as they are the most preferred attackers' target due to the widespread use and persistence of vulnerabilities in their code.

SQL injection (SQLi) remains one of the most prevalent and severe vulnerabilities of web applications [1]. Recently, they were discovered in systems supporting companies in their voice communications [2], data management and processing [3], and even in their defence against web application attacks [4]. Such attacks allowed attackers to bypass authentication and gain access to sensitive data and companies' systems. Despite efforts to improve their security, particularly by constructing SQL queries using prepared statements and placeholders to mitigate SQLi, these measures have not been sufficient, as the number of reported vulnerabilities continues to rise [5]. Notably, SQLi vulnerabilities increased by 135% from 2021 to 2022, 20%–25% each year from 2022 to 2024, and 32% from 2024 to 2025.

Open- and closed-source static analysis tools (SASTs) have been used to discover vulnerabilities [6]–[9]. These tools can provide information about vulnerabilities in the code, but they carry a tendency to produce false positives (FPs) [6] that unnecessarily increase manual effort in code reviews, and often miss vulnerabilities (i.e., false negatives (FNs)).

Recently, prompt-engineering large language models (LLMs) – *peLLMs* – have been applied for vulnerability detection [10]–[15]. The results, while promising, have also exacerbated the FP problem due to LLMs' tendency to hallucinate. Fine-tuning LLMs – *ftLLMs* – have also proven effective at reasoning and classification tasks, but often require expensive training with balanced corpora. This involves retraining parts of the model on a specialised dataset to tune it to a specific domain (e.g., requirement engineering for railway [16]) and so adapt its responses accordingly. Unlike *peLLMs*, *ftLLMs* have not often been used for vulnerability detection [17]; they have still shown better performance than *peLLMs* [18], [19].

Therefore, SASTs and LLMs can be considered reliable options for vulnerability detection, but neither stands out over the other, and neither demonstrates complete effectiveness in this field. For SASTs, various studies have proposed benchmarks and comparisons of them [20]–[22], and have combined the results of SASTs to improve detection and reduce FPs [23]–[25], the latter of which have proven effective in detection but not in reducing FPs. For LLMs, the proposed works primarily rely on text-based input prompts that may include zero to a few shots. To the best of our knowledge, neither the combination of the two LLM types nor the combination of the results of SASTs and LLMs into a single meta-output to improve overall vulnerability detection has been explored yet.

In this paper, we argue that combining SASTs and LLMs

can be beneficial, as the number of real vulnerabilities detected can increase, though this may come at the cost of more FPs. Furthermore, *ftLLMs* can reduce FP rates, as they are trained for the task. In this way, we study the potential of combining different output solutions to obtain a single meta-output that improves vulnerability findings. We hypothesise that: $H1$: The number of vulnerabilities detected increases as the number of combined solutions increases; $H2$: The number of FPs increases as the number of combined solutions increases; $H3$: The strongest combinations consistently include both LLMs and at least one SAST across different datasets and vulnerability expressiveness; $H4$: *ftLLMs* can excel *peLLMs* in Precision and Recall, as they are specifically trained for the vulnerability detection task.

In our study, we tested two SASTs and two LLMs, one open-source and one commercial of each solution type. For LLMs, we used one *ftLLM* and one *peLLM* with a role-based prompt strategy. The study was conducted using a methodology that follows a procedure to create seven curated datasets of SQLi vulnerabilities in PHP code (the most widely used server-side language for web application development [26]), a process to obtain results for the solutions' combinations, and a set of distinct metrics to rank the combinations for each dataset. The datasets include varying levels of SQL injection expressiveness (e.g., the way SQL queries are constructed and the presence of sensitive sinks) and complexity to evaluate the detection capabilities of the tools and models. The results showed that combining both LLMs with a SAST tool can improve the detection of SQLi vulnerabilities. The *ftLLM*, together with SAST, can reduce the many FPs, mainly caused by the *peLLM*; in contrast, both LLMs mitigate SAST's FNs. In general, F1-Score improves from 6% (with a standalone solution) to ≈100% when LLMs are used in combination with SASTs; overall, it increases by 17-60%. Our findings support all hypotheses, except $H2$ (something desired).

The contributions of this paper are: (1) A methodology for combining and ranking the results of SASTs and LLM models, summarising them in a single result. (2) A study and an experimental evaluation of the approach using two SASTs and two LLMs (open-source and commercial) over seven datasets of SQLi vulnerabilities.

II. Methodology

In this section, we present our general approach for evaluating the performance of the combination of SASTs and LLMs in SQLi vulnerability detection. To achieve this, we need a *Global Dataset* from which we can derive subsets in order to evaluate the strengths of the solutions and, based on their results, compare and rank their combinations. The dataset must comprise vulnerable and not-vulnerable SQLi instances, i.e., Positive (P) and Negative (N) instances, respectively. To evaluate the effectiveness of combinations, first, a set of metrics derived from P and N must be defined, as well as a method for combining the results. Subsequently, metrics are computed for each combination, enabling ranking. Figure 1 gives an overview of the methodology, which is explained

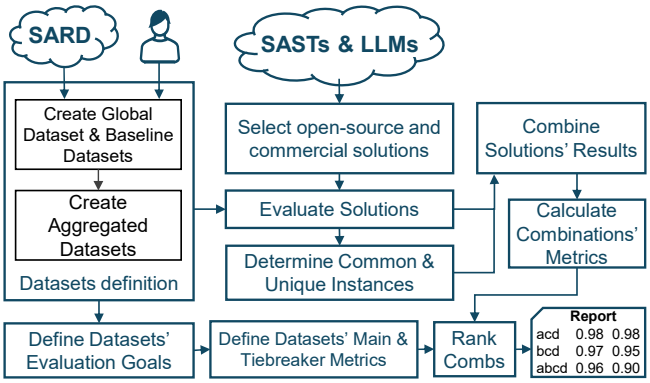

Fig. 1. Methodology overview to improve vulnerability detection.

in the following sections. However, before introducing our methodology, we provide a brief overview of SQLi vulnerabilities, as these are the focus of our study and evaluation.

A. SQL Injection Vulnerabilities

An SQLi occurs when an attacker injects SQL commands and/or meta-characters into the application's unsanitised entry points (e.g., `$_POST` in PHP), such as user forms. These injections, when inserted into an SQL query, can alter its structure so that it is interpreted as a different query, thus allowing the attacker, for instance, to bypass authentication and access to sensitive data from the database. Listing 1 shows an example of a vulnerable PHP code to SQLi. This code verifies whether a user is in the database after they attempt to log in with their credentials, which are requested via two unsanitised entry points (lines 1 and 2). An attacker, for instance, can inject the malicious string `' OR 1=1; --` on the `username` field, which will then alter the final query to `SELECT * FROM users WHERE user='' OR 1=1`. The query is then sent by the *sensitive sink* `mysqli_query` to the database for execution, which enables the extraction of all users' information. In this context, a sensitive sink is a function that, when executed with abnormal data (like injected SQL code), can retrieve data unduly from the database.

```
1 $u = $_GET['username'];
2 $p = $_GET['password'];
3 $qry = "SELECT * FROM users WHERE user='$u' AND pass='$p'";
4 $r = mysqli_query($conn, $qry);
```

Listing 1. Example of PHP code vulnerable to SQLi.

According to Halfond et al. [27], different types of SQLi have distinct goals and characteristics. However, they occur essentially because of unsanitised input fields, which lead languages to adopt mechanisms to prevent them. PHP was endowed with input sanitisation functions (e.g., `mysqli_real_escape_string`) and prepared statements (e.g., `mysqli_prepare`) to mitigate the lack of validation on entry points. Nonetheless, because developers are the ones who apply these mechanisms, they are prone to human error. For this reason, there has also been an effort to develop solutions to detect or prevent this kind of vulnerability [6]–[15], [28]–[30], which, as we previously mentioned, are not

sufficiently effective, and, therefore, our focus on exploring their combination.

B. Datasets Characterisation and Composition

We curated a *Global Dataset* comprising 5,185 instances of PHP code: 5,158 from the NIST Software Assurance Reference Dataset (SARD) [31] and 27 handmade edge cases purposely designed to test specific conditions. Next, we present the characterisation of these instances and how we derived other datasets from them.

1) NIST SARD Dataset: From SARD, we gathered 5,158 PHP test cases, including vulnerable cases and their different patched versions, from various test suites created in different time periods. Specifically, we collected 39 tests from the "PHP Vulnerability Test Suite 2015", 1,429 instances from the "PHP Vulnerability Test Suite 2018", and 3,690 from the "PHP test suite - data flow 1.0.0" from 2022. This dataset contains several SQLi sensitive sinks, as well as forms of providing inputs (i.e., entry points) and sanitising and validating inputs. To make a fair comparison among the SAST and LLM solutions, their combinations, and a more realistic evaluation, we divided the dataset into three subsets based on the following criteria: timeline creation, sensitive sinks, and entry points. The following sections detail each subset and the rationale for including it in the evaluation.

a) Subset 2015 (S-2015): This dataset comprises 39 cases: 13 vulnerable (positive, P) and 26 not-vulnerable (negative, N). All cases involve the `mysql_query` sensitive sink. In addition, specific `$_GET` and `$_POST` entry points are used, and various forms of sanitisation (e.g., `mysql_real_escape_string`, `htmlentities`, `htmlspecialchars`) and validation (e.g., `intval`, `white_lists`) are presented in their instances. *S-2015* is 66% balanced, i.e., 66% of their instances (26 out of 39) are equally distributed by vulnerable and not-vulnerable.

Since it is the oldest dataset and its instances are simple (i.e., few lines of code and without conditional instructions), it is expected that all solutions correctly classify/identify almost all instances, with a slight acceptable deviation because the dataset is not fully balanced. Additionally, some unusual SQLi sanitisation functions are used, such as `htmlentities` (a specific XSS vulnerability sanitisation function) and `sprintf` to build SQL statements. However, we intend to verify how solutions behave in these cases.

b) Subset 2018 (S-2018): This subset contains 1,429 instances, split into 716 positive and 713 negative instances, and is therefore balanced. Input values are provided from 10 different sources, namely PHP super global arrays (e.g., `$_POST`, `$_GET`) and external/system execution files (e.g., `exec`, `system` functions), with 60% of the cases coming from the former source. Almost half of the dataset contains the sink `mysql_query`, where the other half contains the `mysqli_query`. Sanitisation/validation is present over 13 distinct functions, including those identified in *S-2015*, plus `mysqli_real_escape_string`, `addslashes`, filter functions, and data type casting; however, standard SQLi sanitisation functions predominate (489 out of 1,429).

We intend to observe how SASTs perform across the diversity of input and sanitisation/validation forms. We recall that SASTs detect what they learn from their vulnerability knowledge database, which comprises entry points, sanitisation/validation functions, and sensitive sinks associated with the vulnerability classes. We also aim to verify how LLMs handle these forms and assess their overall performance.

c) Subset 2022 (S-2022): *S-2022* contains 3,690 PHP programs, split in a highly imbalanced way: 101 vulnerable and 3,589 non-vulnerable. Its instances are divided equally by 5 different sensitive sinks (`mysqli_prepare`, `mysqli_real_query`, `mysqli_multi_query`, `mysqli::real_query`, `mysqli::multi_query`), thus including 738 instances for each sink. Instances are also equally split into 6 types of entry points – `$_GET`, `$_POST`, `$_REQUEST`, `$_COOKIE`, `apache_request_headers`, `getallheaders` –, with 615 instances for each type. Hence, despite being highly imbalanced across instance categories, its composition is harmonious with respect to entry points and sinks. In terms of sanitisation and validation functions, in addition to those identified in previous subsets, *S-2022* includes others and more recent forms, namely, related to cryptography (e.g., `crypt`, `hash`) and encoding (e.g. `base64`), as well as the standard ones associated with the aforementioned sinks. Lastly, 1,080 instances contain conditional instructions (e.g., if-statements).

With *S-2022*, we aim to verify not only how solutions perform across such a variety of features but also how LLMs behave with imbalanced datasets.

2) Handmade Edge Cases Dataset (HEC): Decision statements and loops allow developers to control program flow and can pose a challenge for SAST tools in vulnerability detection. We generated a handcrafted test suite of 27 cases to verify the solution's ability to handle programs with these features, comprising 18 positive and 9 negative cases. This suite not only tests if-statements but also covers loop constructs such as for, while, and do-while. In this set, we incorporated 7 tests featuring different coding styles from Medeiros et al. [32], as well as 3 classic FNs and FPs. *HEC* contains the `$_GET` and `$_POST` entry points, the `mysqli_query` sensitive sink, and is partially balanced in 66% of its size.

```
1 $u = $_POST['user'];
2 if (isset($u)){
3     $u = mysqli_real_escape_string($u);
4     $q = "SELECT * from users WHERE user='" . $u . "'";
5     $r = mysqli_query($con, $q);
6 }
7
8 $q = "SELECT user from users WHERE user='" . $u . "'";
9 $r = mysqli_query($con, $q);
```

Listing 2. Classic False Negative case

Listings 2 and 3 present classic settings of an FN and an FP, respectively, with which SASTs usually fail. In the FN example, the input is sanitised on line 3, making line 9 non-vulnerable on SASTs. In the FP example, SASTs typically fail

to recognise the `$u == "M"` condition in line 2 as a valid sanitisation method. Finally, Listing 4 presents an FP where the SASTs' sanitisation check fails to consider `substr($u, 0, 4)` in line 2 as proper sanitisation.

```
1 $u = $_POST['user'];
2 if (isset($u) and $u=="M"){
3     $q = "SELECT * from users WHERE user='" . $u . "'";
4     $r = mysqli_query($con, $q);
5 }
```

Listing 3. Classic False Positive case with a if-statement

```
1 $u = $_POST['user'];
2 $u1 = substr($u, 0, 4);
3 $q = "SELECT * from users WHERE user='" . $u1 . "'";
4 $r = mysqli_query($con, $q);
```

Listing 4. Classic False Positive case with a substring function

3) Aggregated Datasets: From the previous four datasets, which we call *Baseline Datasets*, we defined three others, each aggregating instances according to a specific criterion.

a) Aggregated Subset 1 (AS-1): The use of `apache_request_headers` and `getallheaders` entry points is uncommon in PHP web application development. As *S-2022* contains 1,230 instances that use such functions, and we consider that they can introduce bias into the solutions' results, we aggregated them in the *AS-1* dataset. *AS-1* comprises 615 instances of each entry point function, and 246 instances of each sensitive sink, considering the five types of sinks existing in *S-2022*. Also, 360 out of the 1,230 instances include if-statements. Its instances are split into 30 vulnerable and 1,200 non-vulnerable, meaning that *AS-1* is highly imbalanced (as *S-2022*), with only 5% of its instances divided equally by both classes.

b) Aggregated Subset 2 (AS-2): This subset aggregates the remaining cases of *S-2022*, i.e., those with entry point of `$_GET`, `$_POST`, `$_COOKIE`, and `$_REQUEST`, and the cases of *S-2015*, since its instances also include these types of entry point functions. In total, the *AS-2* subset includes 2,499 cases: 84 vulnerable and 2,415 non-vulnerable. It also includes 492 cases for each of the five sinks of *S-2022* and 39 cases for the `mysql_query`. 720 cases have conditional instructions.

c) Aggregated Subset 3 (AS-3): Lastly, we constructed *AS-3* as a set of instances from the four baseline datasets, aiming to combine all characteristics of these datasets into a single, balanced subset. It contains 1,418 test cases (709 P and 709 N cases), namely all instances of *S-2015* and *HEC*, 202 instances of *S-2022*, comprising all 101 P cases and 101 N cases involving all features of *S-2022*, and 1,150 instances of *S-2018*, equally distributed by positive and negative cases and also involving all *S-2018*'s features.

To summarise the datasets described above, Table I presents their composition and characterisation. In total, we have seven datasets (four baseline and three aggregated). *S-2018* and *AS-3* are balanced, *S-2015* and *HEC* are partially balanced (in 66%), and *S-2022*, *AS-1* and *AS-2* are balanced in 5% – 7%.

C. Metrics

To evaluate the performance of the solutions and their combination, we use the standard metrics derived from the confusion matrix: Precision, Recall, and F1-Score. Additionally, we use two other metrics – Informedness and Markedness [23]. Furthermore, we use Accuracy to measure the solutions' performance on balanced datasets. Higher values of these metrics are preferable. Higher Recall and Precision indicate low rates of FPs and FNs. Given TP and TN, the well-classified vulnerable and non-vulnerable instances, and FP and FN, the misclassified as vulnerable and non-vulnerable instances, the above metrics are defined in Table II.

Since datasets differ in their characteristics and some are imbalanced, it is important to define a main metric that more effectively evaluates and ranks solution combinations, and a tiebreaker metric to break ties. Table III defines these metrics. We consider Recall, the main metric for *S-2015* and *S-2018*, as these datasets are old and (partially) balanced, and we expect that any solution can correctly detect a large number of TPs and TNs. For *AS-3*, we also adopt Recall because it is balanced. In contrast, because *AS-1*, *AS-2* and *S-2022* are

TABLE I
DATASETS COMPOSITION AND CHARACTERISATION

		Baseline datasets				Aggregated datasets		
	Feature	**S-2015**	**S-2018**	**S-2022**	**HEC**	**AS-1**	**AS-2**	**AS-3**
General	Total	39	1,429	3,690	27	1,230	2,499	1,418
	Vulnerable (P)	13	716	101	18	30	84	709
	Not-Vulnerable (N)	26	713	3,589	9	1,200	2,415	709
	Balanced dataset	66%	100%	5%	66%	5%	7%	100%
	Conditional instructions (ifs and loops)			1,080	27	360	720	87
Entry Points	`$_GET`	11	551	615	16		626	613
	`$_POST`	28	209	615	11		643	220
	`$_COOKIE`			615			615	40
	`$_SESSION`		90					25
	`$_REQUEST`			615				31
	`apache_request_headers`			615		615		30
	`getallheaders`			615		615		30
	backticks, `exec`, `shell_exec`		309					180
	`system`, `proc_open`, `popen`, `fopen`		270					249
Sen. Sinks	`mysql_query`	39	714				39	614
	`mysqli_query`		715		27			602
	`mysqli_prepare`			738		246	492	42
	`mysqli_real_query`			738		246	492	10
	`mysqli_multi_query`			738		246	492	44
	`mysqli::real_query`			738		246	492	41
	`mysqli::multi_query`			738		246	492	45
Sanitisation Functions	`mysql_real_escape_string`	3	189				3	192
	`mysqli_real_escape_string`		300		7			167
	`addslashes`		1					1
	`sprintf`		2	47			47	2
	`htmlentities`	2	2				2	4
	`htmlspecialchars`	3	3				3	6
	Cast data type		45					45
	Filter function		146					146
	`intval`	4	7	300		60	244	16
	`floatval`	8	10	300		60	248	22
	`ctype*`, `double*`			480		240	240	13
	`preg_replace`	2	2	2			4	4
	`sub_str`, `str*`			480	1	240	240	17
	white_lists	4	6	300	1	120	184	11
	`crc32`, `crypt`, `hash*`			840		240	600	35
	`base64`, `bin2hex`, `encode`, `gz*`			840		240	600	28

TABLE II
METRICS DERIVED FROM THE CONFUSION MATRIX

Metric	**Definition and Formula**
Accuracy *(Acc)*	Measures how correctly the tool classifies instances as vulnerable and not-vulnerable over the total number of instances. $Acc = (TP + TN)/(P + N)$
Recall *(Rec)*	Measures the proportion of actual vulnerabilities that are correctly identified by the tool. $Rec = TP/(TP + FN)$
Precision *(Pr)*	Measures the proportion of correctly identified vulnerabilities (TP) out of all instances flagged as vulnerabilities by the solution. $Pr = TP/(TP + FP)$
F1-Score *(F1)*	Measures the harmonic mean of Precision (Pr) and Recall (Rec), providing a single metric that balances the trade-off between these two. A higher F1 Score is preferred, especially when the number of detected vulnerabilities is imbalanced. $F1 = 2 * (Pr * Rec)/(Pr + Rec)$
Informedness *(Inf)*	Measures how consistently a solution predicts a TP and a TN. Knowing the number of P (P = TP + FN) and N (N = FP + TN) instances, i.e., the number of vulnerable and not-vulnerable instances of the dataset, the metric increases with every TP in the proportion 1/P, and decreases with every FP in the proportion 1/N. Since the prevalence of P instances is less than the prevalence of N instances (e.g., in datasets AS-1, AS-2), the metric prioritises the solutions reporting more vulnerabilities and, at the same time, not too many FPs. $Inf = Rec + InverseRec - 1$.
Markedness *(Mark)*	Measures how consistently a solution has the outcome as a marker. Considers only the number of TPs and TNs reported. The metric sums the proportions of the P and the N instances that are correctly identified as such. $Mark = Pr + InversePr - 1$. For the Precision (1st part of the formula), a TP increases the metric while an FP decreases the metric in the same proportion. For the Inverse Precision (2nd part of formula), a FP also decreases the metric but at the inverse proportion of the N instances reported (i.e., low value). Therefore, the metric severely penalises solutions reporting FPs.

TABLE III
MAIN AND TIEBREAKER METRICS FOR EACH DATASET.

Dataset	Main Metric	Tiebreaker Metric	Dataset	Main Metric	Tiebreaker Metric
S-2015	Recall	F1-Score	AS-1	Informedness	Precision
S-2018	Recall	F1-Score	AS-2	Informedness	Precision
S-2022	Informedness	Precision	AS-3	Recall	F1-Score
HEC	F1-Score	Markedness			

highly imbalanced, we select Informedness to main metric, as we aim to prioritise solutions that correctly detect TPs. Lastly, for *HEC*, we use F1-Score as the main metric to evaluate the effectiveness of solutions in processing edge cases.

D. SASTs and LLMs Selection

We consider that for a correct evaluation of solutions and their combination, both open- and closed-source (i.e., commercial) solutions should be included. Additionally, based on the literature [8], [10]–[12], [14]–[16], we selected the most reputable and best-performing SASTs and LLMs rather than evaluating a range of solutions of these types. Thus, we selected two SAST tools and two LLM models, one open-source and one commercial, for each category. For LLMs, we also considered one fine-tuning (*ftLLM*) and one prompt engineering (*peLLM*). Next, we briefly present each solution.

1) KAVe: KAVe [33] is an open-source SAST tool that targets SQLi and XSS vulnerabilities in PHP web applications through a knowledge-based multi-agent system operating over a Multi-Layer Knowledge Graph (MLKG). The MLKG fuses complementary program representations (Function Call Graph, Control Flow Graph, Dependence Variable Graph, and Program Dependence Graph), and enriches them with security properties such as entry points, sanitisation and validation functions, and sensitive sinks. A set of cooperating agents then performs static taint analysis on this graph, using pruning to discard code regions that are provably irrelevant for vulnerability propagation and focusing traversal on security-relevant paths. This design allows KAVe to capture both intra- and inter-procedural flows while mitigating FPs. Ramires et al. [8] demonstrated that the tool outperformed several representative SASTs in the literature. Therefore, KAVe serves as a strong, research-grade open-source SAST baseline in this work.

2) SonarQube: SonarQube [34] (SonarQ for short) is a widely adopted commercial SAST tool and code-quality platform that analyses source code to detect bugs, code smells, and security issues across multiple languages, including PHP. For security, it relies on large rule sets and taint-analysis to flag untrusted inputs that reach sensitive operations, covering common web vulnerabilities, such as SQLi. In this work, SonarQ is used as a representative commercial-grade SAST baseline for SQLi vulnerability detection in PHP code.

3) Zero-shot GPT-4o LLM with role-based prompt engineering: GPT-4o [35] is a state-of-the-art commercial LLM with strong multilingual code understanding and generation capabilities. In our study, GPT-4o is used in a strict zero-shot configuration, i.e., without any task-specific fine-tuning; instead, its behaviour is steered through role-based prompts that explicitly instruct the model to act as a security analyst specialised in SQLi vulnerability detection for PHP. The prompt (see Listing 5) assigns the model a generic security analyst role and asks it to inspect each PHP test case and decide whether it is vulnerable to SQLi. This minimal role-based setup reflects a realistic usage mode in which practitioners rely on proprietary LLMs with very lightweight prompt engineering, and thus serves as a high-end commercial baseline for comparison with the fine-tuned LLM and the SAST tools.

```
Prompt:
You are a Security Analyst specialised in identifying SQL
injection (SQLi) vulnerabilities. Your task is to analyse
PHP code for SQLi vulnerabilities.
- Check if user input is sanitised or escaped before being
  used in SQL queries.
- Respond ONLY with 'Safe' if there are no vulnerabilities
  or 'Unsafe' if there are any vulnerabilities.

PHP code to analyse:
< code of a test case >
```

Listing 5. Role-based prompt for GPT-4o

4) Fine-tuning Qwen2.5-Coder-1.5B with LoRA: LLM models are typically pre-trained on large-scale corpora using self-supervised objectives, enabling them to acquire broad syntactic and semantic knowledge across programming languages. While such pre-training of LLMs has shown strong general representations, performance on a wide range of domain-specific tasks requires *fine-tuning* the model. However, *fine-tuning* all parameters of the modern LLMs is often impractical due to their large scale, resulting in computational and memory overhead. To address this limitation, we adopt a Parameter-Efficient Fine-Tuning (PEFT) [36] approach, in which the pre-trained model weights are frozen, and only a small set of additional parameters is optimised. In particular, we employ Low-Rank Adaptation (LoRA) [37], which introduces trainable low-rank matrices into selected linear layers of the transformer architecture, enabling efficient learning of task-specific updates while keeping the original model parameters fixed. In this work, we opted *Qwen2.5-Coder-1.5B* [38], an open-source *ftLLM* that has proven effective in processing programming languages compared to other competitive models like GPT-4o [18], [19]. To adapt the model to the downstream task of SQLi detection, the base model is loaded in 4-bit precision using QLoRA-style quantisation, substantially reducing memory usage while preserving model capacity. The model is then augmented with LoRA modules inserted into both the attention projection layers and the feed-forward projections, allowing effective task-specific contextual learning while training only a small fraction of parameters compared to full fine-tuning. For training and testing, we use 3-fold stratified cross-validation: we split the dataset into 3 folds, train on 2 folds, and test on the remaining fold. This process is performed three times with varying folds for training and testing for better cross-validation. The final metrics are the average across this three-fold validation.

E. Combination of Solutions' Results

Next, we explain our procedure for combining and evaluating the results of the solutions to determine which is best for each dataset, thereby yielding a single meta-solution.

1) *Calculate the confusion matrices for each solution:* using each dataset, execute the four solutions and calculate their confusion matrices (i.e., TP, TN, FP, FN). Determine the common outputs regarding the elements of confusion matrices to obtain an upset-style data visualisation of their intersections, and the solutions' unique outputs.
2) *Combine the solutions:* For each dataset and combination of solutions, merge the results obtained in 1).
3) *Compose the combined confusion matrices:* with the values of 1) and 2), create the corresponding confusion matrices of the combined solutions, for each dataset.
4) *Calculate the metrics and rank:* using the confusion matrices of 3), determine the metrics and rank the combined solutions, for each dataset.

III. Results and Discussion

To assess the hypotheses $H1$ to $H4$, we evaluate the solution combinations and determine their rationale for improving SQLi vulnerability detection. To do so, the following sections present, respectively, the evaluation of the standalone solutions (Section II-D) when processing the datasets of Table I, the solution combinations based on i) intersection and uniqueness solution results and ii) datasets' code features and balancing, and the testing of $H1$ to $H4$ based on the obtained results.

A. Comparing the Standalone Solution Results

We run the SAST and LLM solutions on 6 datasets: the 3 Baseline datasets (*S-2015*, *S-2018*, *HEC*) and the 3 Aggregated datasets (*AS-1*, *AS-2*, *AS-3*). We opted not to run them on *S-2022*, since the results for this dataset can be inferred from the results of *AS-1* and *AS-2*. Table IV presents the confusion matrices and solutions' metrics across the datasets. For a better understanding of the results, the *General* characteristics of Table I were included in Table IV.

TABLE IV
Confusion matrices and metrics of solutions over datasets

		Baseline dataset				Aggregated dataset		
General		**S-2015**	**S-2018**	**S-2022**	**HEC**	**AS-1**	**AS-2**	**AS-3**
Total		39	1,429	3,690	27	1,230	2,499	1,418
Vulnerable (P)		13	716	101	18	30	84	709
Not-Vulnerable (N)		26	713	3,589	9	1,200	2,415	709
Balanced dataset		66%	100%	5%	66%	5%	7%	100%
Conf. Matrix		**S-2015**	**S-2018**	**S-2022**	**HEC**	**AS-1**	**AS-2**	**AS-3**
KAVe	TP	10	168	44	16	0	54	184
	TN	21	652	2,229	7	1,200	1,050	620
	FP	5	61	1,360	2	0	1,365	89
	FN	3	548	57	2	30	30	525
SonarQ	TP	5	93	35	18	30	10	151
	TN	23	667	3,444	6	1,170	2,297	655
	FP	3	46	145	3	30	118	54
	FN	8	623	66	0	0	74	558
GPT-4o	TP	13	715	73	18	30	56	680
	TN	12	121	1,474	0	466	1,020	165
	FP	14	592	2,115	9	734	1,395	544
	FN	0	1	28	0	0	28	29
Qwen	TP	0	696	0	11	0	0	637
	TN	26	703	3,589	8	1,200	2,415	655
	FP	0	10	0	1	0	0	54
	FN	13	20	101	7	30	84	72
Metric		**S-2015**	**S-2018**	**S-2022**	**HEC**	**AS-1**	**AS-2**	**AS-3**
KAVe	Acc	.79	.57	.62	.85	.98	.44	.57
	Rec	.77	.23	.44	.89	0	.64	.26
	Pr	.67	.73	.03	.89	0	.04	.67
	F1	.71	.36	.06	.89	0	.07	.37
	Inf	.58	.15	.06	.67	0	.08	.13
	Mark	.54	.28	.01	.67	0	.01	.22
SonarQ	Acc	.72	.53	.94	.89	.98	.92	.57
	Rec	.38	.13	.35	1	1	.12	.21
	Pr	.63	.67	.19	.88	.50	.08	.74
	F1	.48	.22	.25	**.93**	.67	.09	.33
	Inf	.27	.07	.31	.50	.98	.07	.14
	Mark	.37	.19	.18	.88	.50	.05	.28
GPT-4o	Acc	.64	.59	.42	.67	.40	.43	.60
	Rec	1	1	.72	1	1	.67	.96
	Pr	.48	.55	.03	.67	.04	.04	.56
	F1	.65	.71	.06	.80	.08	.07	.70
	Inf	.46	.17	.13	0	.39	.09	.19
	Mark	.48	.54	.01	0	.04	.01	.41
Qwen	Acc	.67	.98	.97	.70	.98	.97	.91
	Rec	0	.97	0	.61	0	0	.90
	Pr	0	.99	0	.92	0	0	.92
	F1	0	.98	0	.73	0	0	.91
	Inf	0	.96	0	.50	0	0	.82
	Mark	0	.96	0	.45	0	0	.82

1) SASTs Analysis: KAVe performed better than SonarQ when processing the oldest datasets – *S-2015* and *S-2018* –, achieving a <Recall, F1-Score> of <0.77, 0.71> and <0.23, 0.36>, respectively, against <0.38, 0.48> and <0.13, 0.22>. KAVe performs approximately twice as well as SonarQ. The FPs and FNs of both tools were associated with instances involving the `sprintf` function to construct SQL queries using placeholders as formatted strings, and the XSS sanitisation functions `htmlentities` and `htmlspecialchars`. In *S-2018*, both tools reported a very high number of FNs (548 and 623 out of the 713 N cases, respectively, for KAVe and SonarQ), largely due to the use of non-standard entry-point forms (e.g., backticks). We recall that if SAST tools do not have existing entry points in their vulnerability knowledge databases for the code under analysis, they will miss vulnerabilities that rely on them. This is also particularly evident when KAVe processes *AS-1*, in which all instances have `apache_request_headers` and `getallheaders` as entry points, leading the tool to fail to detect all 30 vulnerability cases. For the same reason, it reported the 1,200 TN cases as such. In contrast, SonarQ correctly detected those vulnerable cases and almost all non-vulnerable cases, thus widely outperforming KAVe. In the *AS-2* dataset, KAVe detected 64% of the SQLi-vulnerable cases, while SonarQ detected only 12%. However, unlike KAVe, this tool correctly flagged 95% of the dataset's negative cases. KAVe detected around half of the negative cases, indicating a high rate of FPs, and so a very low Precision (0.04). This poor performance can be attributed to KAVe's inability to recognise validation and sanitisation forms in *AS-2*, such as encryption and encoding, which leads the tool to flag these cases as vulnerable. Nevertheless, SonarQ's 0.08 precision was not higher than KAVe's. This is due to its lack of detection of vulnerable cases.

Regarding the *HEC* dataset, SonarQ performed better than KAVe with the handmade, tailored edge cases. Both tools obtained similar Precision (0.89), but SonarQ achieved perfect Recall (1 or 100%), while KAVe only reached 0.89. KAVe had 2 FNs (one listed in Listing 2) and 2 FPs, both related to if-statements (one listed in Listing 3). SonarQ did not have FNs, but had 3 FPs, including those presented in Listings 3 and 4.

Lastly, both tools reached a moderate Accuracy of 0.57 with the balanced *AS-3* dataset. KAVe had slightly better Recall and F1-Score, whereas SonarQ outperformed it in Precision.

2) LLMs Analysis: The *peLLM*, i.e., role-based GPT-4o (or simply GPT-4o), showed a similar pattern across all 7 datasets: it achieved a total Recall (except in a few cases), indicating that it detected (almost) all SQLi cases. However, for *AS-2*, GPT-4o achieved a Recall of 0.67 due to the presence of the `sprintf` function in cases where it fails detection. For the same reason, GPT-4o achieved a Recall 0.72 for *S-2022*. On the other hand, the Precision did not exceed 0.04 in some datasets and ranged from 0.48 to 0.67 in others, indicating that GPT-4o tends to generate a considerable number of FPs. The discrepancy between these values and the distance between these metrics stems from LLM hallucinations, a natural characteristic and limitation of general-purpose *peLLM*. For the vulnerability detection task, this means that GPT-4o hits every instance containing a sensitive sink as vulnerable. Thus, it flags all vulnerable instances as such, but at the same time, it will also output non-vulnerable cases as vulnerable, generating several FPs. Due to this discrepancy, GPT-4o achieved a moderate F1-Score ($>= 0.65$) on some datasets, giving a false impression that the model is precise, i.e., that it produces few FPs. Furthermore, this false impression can be confirmed by Accuracy, which was $\sim$0.60 on balanced datasets (*S-2018* and *AS-3*), indicating that the model failed on TN instances.

Qwen *ftLLM* (Qwen for short) performed poorly on imbalanced datasets, yielding the worst metrics. In contrast, on balanced datasets (*S-2018* and *AS-3*), Qwen achieved the best performance. Note that *ftLLM*' performance is as good as the training data. Hence, in imbalanced training datasets where one class predominates, the model tends to classify examples as that class. *S-2022*, *AS-1*, and *AS-2* are highly imbalanced, in which non-vulnerable instances predominate; therefore, Qwen classifies all test cases as non-vulnerable, not allowing it to correctly identify SQLi vulnerabilities. For *S-2015*, this scenario is also observed. Although it is balanced at 66%, the majority of instances are non-vulnerable, so Qwen classifies instances as such. *HEC* is also balanced at 66%, but the opposite is observed, although not all cases were classified as vulnerable, justified by the small difference in the number of instances between the two classes. Even so, Qwen outperformed GPT-4o. The single FP it had is that of Listing 4, but it had more FNs, including that of Listing 2.

3) Global Analysis: After the previous detailed analysis, no solution outperforms the others in all datasets. Comparing SAST tools, they performed similarly, with KAVe achieving slightly better results on some datasets (*S-2015*, *S-2018*, *AS-2*, *AS-3*), while SonarQ slightly performed better on others (*S-2022*, *HEC*, *AS-1*). When comparing them with GPT-4o, GPT-4o performs better in *S-2018* and *AS-3*, but in terms of Accuracy and Recall. Qwen clearly performed the best on balanced datasets (*S-2018* and *AS-3*).

These observations lead us to conduct a thorough analysis of TPs and TNs, focusing on the intersections of the solutions' results across datasets. Table V shows the findings in the form of UpSet-style data visualisation, where the rows with a single solution (a, b, c or d) indicate that it was the unique solution that flagged that number of cases, whereas the other rows present the intersections between solutions. For example, for TPs of *AS-2*, we observe that KAVe (a) was the unique tool that detected 11 cases, GPT-4o (c) was the unique solution that detected 13 other cases, 35 instances were only detected by both solutions (ac), and no instances were detected by all solutions ($abcd$); however, $abcd$ failed the detection of the same 15 cases (FNs). These results, at first glance, suggest that combining the solutions may enhance vulnerability detection, prompting us to conduct the analysis in the next section.

TABLE V
COMMON TP, TN, FP AND FN BETWEEN SOLUTIONS, AND UNIQUE TP AND TN BY A SINGLE SOLUTION.

True Positives							
Solutions	**S-2015**	**S-2018**	**S-2022**	**HEC**	**AS-1**	**AS-2**	**AS-3**
a	0	0	11	0	0	11	5
b	0	0	1	0	0	1	0
c	3	0	10	0	0	13	9
d	0	1	0	0	0	0	7
ab	0	0	1	0	0	1	0
ac	5	12	30	0	0	35	33
ad	0	0	0	0	0	0	6
bc	0	0	31	2	30	1	2
bd	0	0	0	0	0	0	1
cd	0	520	0	0	0	0	439
abc	5	8	2	5	0	7	14
abd	0	0	0	0	0	0	1
acd	0	90	0	0	0	0	50
bcd	0	27	0	0	0	0	59
abcd	0	58	0	11	0	0	75
FNs abcd	0	0	15	0	0	15	9
True Negatives							
Solutions	**S-2015**	**S-2018**	**S-2022**	**HEC**	**AS-1**	**AS-2**	**AS-3**
a	0	0	0	1	0	0	2
b	0	0	0	0	0	0	5
c	0	0	0	0	0	0	1
d	0	19	44	2	0	44	18
ab	0	5	0	6	0	0	12
ac	0	0	0	0	0	0	0
ad	3	22	43	0	15	31	25
bc	0	0	0	0	0	0	12
bd	5	30	808	0	0	813	37
cd	0	0	25	0	0	25	0
abc	0	0	0	0	0	0	14
abd	6	511	1,220	6	719	507	437
acd	0	0	33	0	15	18	0
bcd	0	7	483	0	0	483	8
abcd	12	114	897	0	415	494	130
FPs abcd	0	5	0	0	0	0	8

a: KAVe, b: SonarQ, c: GPT-4o, d: Qwen

B. Comparing the Combination of Solution Results

We assessed all combinations of the 4 solutions to determine which performs best on each dataset. To do so, we created the confusion matrix for the combinations based on the results of Section III-A, and calculated the metrics. Afterwards, we ranked the combinations for each dataset, using the main metrics of Table II. As the main metrics are Recall and Informedness, both of which rely on TPs and FNs, we ranked the results according to the following criterion: maximum TPs, minimum FNs, and maximum TNs. Tables VI and VII present the results, respectively, to the Baseline datasets and the Aggregated datasets. The bottom part of Table VII also shows the overall results for the *Global Dataset* (dataset that comprises all Baseline datasets).

Analysing the Top-4 combinations of the Baseline datasets (Table VI), it is evident that, datasets partially or fully bal-

TABLE VI
RANKED SOLUTIONS' RESULTS COMBINATION FOR BASELINE DATASETS.

Comb.	TP	TN	FP	FN	Acc	Rec	Pr	F1	Inf	Mark
S-2015										
cd	13	26	0	0	1	1	1	1	1	1
abc	13	26	0	0	1	1	1	1	1	1
acd	13	26	0	0	1	1	1	1	1	1
bcd	13	26	0	0	1	1	1	1	1	1
abcd	13	26	0	0	1	1	1	1	1	1
bc	13	23	3	0	.92	1	.81	.90	.88	.81
ac	13	21	5	0	.87	1	.72	.84	.81	.72
ab	10	26	0	3	.92	.77	1	.87	.77	.90
ad	10	26	0	3	.92	.77	1	.87	.77	.90
abd	10	26	0	3	.92	.77	1	.87	.77	.90
bd	5	26	0	8	.79	.38	1	.56	.38	.76
S-2018										
ad	716	708	5	0	1	1	.99	1	.99	.99
abd	716	708	5	0	1	1	.99	1	.99	.99
acd	716	708	5	0	1	1	.99	1	.99	.99
bcd	716	708	5	0	1	1	.99	1	.99	.99
abcd	716	708	5	0	1	1	.99	1	.99	.99
cd	716	703	10	0	.99	1	.99	.99	.99	.99
abc	715	689	24	1	.98	1	.97	.98	.96	.97
bc	715	667	46	1	.97	1	.94	.97	.93	.94
ac	715	659	54	1	.96	1	.93	.96	.92	.93
bd	704	708	5	12	.99	.98	.99	.99	.98	.98
ab	195	689	24	521	.62	.27	.89	.42	.24	.46
S-2022										
abc	86	3,589	0	15	1	.85	1	.92	.85	1
abcd	86	3,589	0	15	1	.85	1	.92	.85	1
acd	85	3,589	0	16	1	.84	1	.91	.84	1
ac	85	2,773	816	16	.77	.84	.09	.17	.61	.09
abd	76	3,589	0	25	.99	.75	1	.86	.75	.99
ab	76	3,556	33	25	.98	.75	.70	.72	.74	.69
bcd	75	3,589	0	26	.99	.74	1	.85	.74	.99
bc	75	3,538	51	26	.98	.74	.60	.66	.73	.59
cd	73	3,589	0	28	.99	.72	1	.84	.72	.99
ad	44	3,589	0	57	.98	.44	1	.61	.44	.98
bd	35	3,589	0	66	.98	.35	1	.51	.35	.98
HEC										
acd	18	9	0	0	1	1	1	1	1	1
cd	18	8	1	0	.96	1	.95	.97	.89	.95
ac	18	7	2	0	.93	1	.90	.95	.78	.90
ad	16	9	0	2	.93	.89	1	.94	.89	.82
bd	16	8	1	2	.89	.89	.94	.91	.78	.74
bcd	16	8	1	2	.89	.89	.94	.91	.78	.74
bc	16	6	3	2	.81	.89	.84	.86	.56	.59
abd	16	3	6	2	.70	.89	.73	.80	.22	.33
abcd	16	3	6	2	.70	.89	.73	.80	.22	.33
ab	16	1	8	2	.63	.89	.67	.76	0	0
abc	16	1	8	2	.63	.89	.67	.76	0	0

a: KAVe, b: SonarQ, c: GPT-4o, d: Qwen

TABLE VII
RANKED SOLUTIONS' RESULTS COMBINATION FOR AGGREGATED DATASETS AND GLOBAL DATASET.

Comb.	TP	TN	FP	FN	Acc	Rec	Pr	F1	Inf	Mark
AS-1										
ab	30	1,200	0	0	1	1	1	1	1	1
ac	30	1,200	0	0	1	1	1	1	1	1
bc	30	1,200	0	0	1	1	1	1	1	1
bd	30	1,200	0	0	1	1	1	1	1	1
cd	30	1,200	0	0	1	1	1	1	1	1
abc	30	1,200	0	0	1	1	1	1	1	1
abd	30	1,200	0	0	1	1	1	1	1	1
acd	30	1,200	0	0	1	1	1	1	1	1
bcd	30	1,200	0	0	1	1	1	1	1	1
abcd	30	1,200	0	0	1	1	1	1	1	1
ad	0	1,200	0	30	.98	0	0	0	0	0
AS-2										
abcd	69	2,415	0	15	.99	.82	1	.90	.82	.99
abc	69	2,371	44	15	.98	.82	.61	.70	.80	.60
acd	68	2,415	0	16	.99	.81	1	.89	.81	.99
ac	68	1,558	857	16	.65	.81	.07	.13	.45	.06
bcd	58	2,415	0	26	.99	.69	1	.82	.69	.99
bc	58	2,340	75	26	.96	.69	.44	.53	.66	.43
cd	56	2,415	0	28	.99	.67	1	.80	.67	.99
abd	56	2,415	0	28	.99	.67	1	.80	.67	.99
ab	56	2,346	69	28	.96	.67	.45	.54	.64	.44
ad	54	2,415	0	30	.99	.64	1	.78	.64	.99
bd	10	2,415	0	74	.97	.12	1	.21	.12	.97
AS-3										
acd	699	696	13	10	.98	.99	.98	.98	.97	.97
abcd	698	701	8	11	.99	.98	.99	.99	.97	.97
cd	694	682	27	15	.97	.98	.96	.97	.94	.94
bcd	693	699	10	16	.98	.98	.99	.98	.96	.96
abc	692	683	26	17	.97	.98	.96	.97	.94	.94
ac	692	641	68	17	.94	.98	.91	.94	.88	.88
abd	690	700	9	19	.98	.97	.99	.98	.96	.96
ad	689	683	26	20	.97	.97	.96	.97	.94	.94
bc	681	656	53	28	.94	.96	.93	.94	.89	.89
bd	652	698	11	57	.95	.92	.98	.95	.90	.91
ab	245	682	27	464	.65	.35	.90	.50	.31	.50
GLOBAL DATASET										
acd	832	4,332	5	16	1	.98	.99	.99	.98	.99
abcd	831	4,326	11	17	.99	.98	.99	.98	.98	.98
ac	831	3,460	877	17	.83	.98	.49	.65	.78	.48
abc	830	4,305	32	18	.99	.98	.96	.97	.97	.96
bcd	820	4,331	6	28	.99	.97	.99	.98	.97	.99
cd	820	4,326	11	28	.99	.97	.99	.98	.96	.98
bc	819	4,234	103	29	.97	.97	.89	.93	.94	.88
abd	818	4,326	11	30	.99	.96	.99	.98	.96	.98
ad	786	4,332	5	62	.99	.93	.99	.96	.93	.98
bd	760	4,331	6	88	.98	.90	.99	.94	.89	.97
ab	297	4,272	65	551	.88	.35	.82	.49	.34	.71

a: KAVe, b: SonarQ, c: GPT-4o, d: Qwen

anced, such as *S-2015* and *S-2018*, where Recall is essential, there is no need to combine all solutions ($abcd$) to reach 1 (i.e., 100%) on Recall. *LLM models (c and d) contribute positively when combined with a SAST tool (a or b).* Moreover, it is visible that their combination is beneficial: while c (GPT-4o) can detect a significant percentage of TPs, but with the downside of producing several FPs, d (Qwen) has the capacity of improving TP detection and correctly detecting TNs, thus reducing the FPs produced by c. This observation is evident in *S-2018*, and, therefore, it is of utmost interest to use d with balanced datasets, or at least 66% balanced. In *HEC*, acd was the best combination with an Informedness of 100%. Once again, we observe both LLMs together with a SAST. In *S-2022*, the highly imbalanced dataset, both SASTs together with c appear as the best combination. a (KAVe) is the SAST tool that often appeared in Top-3 or Top-4. The ab (KAVe with SonarQ) SAST combination appears only in the lower half of the ranking, indicating that although its combined results improve detection, it is widely outperformed when a single SAST is combined with LLMs (c and d).

For the aggregated datasets, although the obtained metrics for *AS-1* are close to 100%, this dataset indicates only that security analysts should know the SAST tools they use, ensuring that their vulnerability knowledge databases are consistent with the types of vulnerabilities and the features they intend to search for. For *AS-2* and *AS-3*, the same observation from the Baseline datasets holds: *LLM results improve detection when combined with at least one SAST tool.*

Lastly, for the Global dataset, acd yields the best results, consistent with our previous observations.

C. Testing the Hypotheses

Based on our findings, for our hypothesis, we can state: $H1$ is true because the combination of various solutions increases the number of vulnerabilities found; In contrast, $H2$ is false because we observed that the number of FPs does not always increase with the number of solutions in a combination (e.g., *S-2015*: cd, abc; *S-2018*: $abcd$); Across the diverse datasets and regardless of vulnerability expressiveness, the strongest combinations consistently include both LLMs and at least one SAST, with acd achieving the best overall results; therefore, we accept $H3$; We also observed that the *ftLLM* trained on

balanced datasets excels the *peLLM* in Accuracy, Precision, and F1-Score (e.g., in *S-2018* and *AS-3* of Table IV). When combined with the *peLLM*, it cancels out the FPs produced by this, indicating that it outperforms the other within the combination. Hence, we also consider $H4$ true.

In summary, all hypotheses are true, except $H2$. Therefore, it is beneficial to combine SAST tools and LLMs, specifically *peLLM* and *ftLLM* models trained on balanced datasets. This combination of LLMs harmonises user-prompt interaction, for guiding *peLLMs* at the cost of producing excessive FPs, with a *ftLLM* trained explicitly for the vulnerability-detection task, which enables high detection while reducing the undesired FPs. SAST tools within the combined solution help increase TPs, as they can detect vulnerabilities that LLMs miss (e.g., *S-2022* in Table V, reflected then in TPs of Table VI).

IV. Related Work

A. Vulnerability Detection using Static Analysis

Substantial research has investigated static analysis for detecting web application vulnerabilities, especially SQLi in PHP, ranging from pattern-based taint analysis over abstract syntax trees and control-flow graphs to more advanced data-driven approaches [6], [7], [9], [39]. Tools such as WAP combine static analysis with data mining to detect and automatically correct web vulnerabilities [6], and Merlin further extends static detection to multi-language web applications [7]. Building on richer code representations, Recurscan and HiddenCPG use code property graphs and graph-based pattern matching to detect recurring vulnerabilities and vulnerable clones at scale [9], [39], and KAVe extends this direction with a multi-layer knowledge graph and a multi-agent system to improve precision for PHP applications [8]. Recent work has also explored combining various SAST tools to improve vulnerability detection, at the cost of increasing FPs [23], and others integrate multiple SAST tools with machine-learning-based fusion to improve coverage [40]. Finally, NLP-oriented techniques for processing web applications as natural language demonstrate yet another static perspective on vulnerability identification, reinforcing that no single SAST or representation is sufficient on its own [29], [30] and motivating the combined-solution view adopted in this paper.

B. Vulnerability Detection using Large Language Models

LLMs have recently been explored for software security tasks such as vulnerability detection and repair. Several studies show that they, when prompted appropriately, can identify and even propose fixes for security issues, but they also report unstable precision, hallucinated findings, and limited control- and data-flow awareness, especially in security-critical settings [10], [11], [13]. To better understand this space, recent surveys analyse the capabilities and limitations of LLMs in this field, highlighting open challenges in making prompt-based approaches robust enough for practical vulnerability assessment pipelines [14], [15], [41], [42]. In parallel, emerging work proposes hybrid approaches that combine LLMs with static analysis, for example, by using prompts to guide taint analysis or to refine SAST reports, showing that LLMs can complement, but not yet replace, traditional static analysis techniques [12], [13].

Fine-tuning LLMs on domain-specific corpora is a widely adopted approach to enhance their effectiveness on downstream tasks, including vulnerability detection. A recent comprehensive review of LLMs for vulnerability detection and repair tasks by Zhou et al. [17] finds that fine-tuning is the predominant paradigm, with zero-shot and few-shot methods used far less frequently, since these tasks require reasoning about the semantic behaviours of programs and long-range contextual dependencies that may not be fully captured in general-purpose pretraining corpora [43]. The study also shows that adapting models to domain-specific data is essential to align representations with these characteristics, thereby improving vulnerability detection accuracy. Chan et al. [44] performed a comprehensive comparative analysis of vulnerability detection methods, evaluating zero-shot and few-shot prompting against fine-tuned models across multiple datasets. Their results indicate that, under the evaluated settings, fine-tuning consistently achieved a superior precision–recall trade-off compared to prompt-only approaches.

Although diverse approaches to vulnerability detection are found in the literature, none have combined the results of SAST tools and fine-tuned and prompt-strategy LLM models in a single solution, as we propose. Unlike approaches that combined multiple SASTs for detection improvement purposes, but also incrementing FPs, our SASTs-LLMs proposal shows that it improves detection and reduces FPs and FNs.

V. Conclusions

This paper presented a study on combining results of SAST tools and LLM models, specifically fine-tuning and prompt-engineering models, to improve vulnerability detection of web applications. We proposed a methodology comprising dataset definition, solution selection, dataset metric assignment, solution result combination, and ranking, which we then employed to detect SQLi vulnerabilities. Our findings show that combining at least one SAST with both LLMs improves the F1-score by 17-60% compared to standalone solutions. Furthermore, the fine-tuning LLM, trained on balanced datasets, can reduce the excessive number of FPs generated by the prompt-engineering LLM; at the same time, both LLMs mitigate the limitation of SASTs missing vulnerabilities (i.e., FNs). These findings can help software developers and security analysts adopt a new approach to improve vulnerability detection, without the undesirable time-consuming process of reviewing code for nonexistent vulnerabilities (i.e., FPs). Moreover, the proposed methodology is generic enough, and we therefore envisage its extension to other datasets and solutions, and application to other vulnerability classes and programming languages.

Acknowledgment

This work was partially supported by P2030 through project I2DT (COMPETE2030-FEDER-00389100; ITEA4 ref. 22025), FCT through the LASIGE Research Unit

(UID/00408/2025), KKS through project INDTECH (20200132), and by EC through project HORIZON-CHIPS-JU MATISSE (101140216).

REFERENCES

[1] OWASP Top 10 Team, "OWASP Top Ten 2025," https://owasp.org/Top10/2025/0x00_2025-Introduction/, 2025.

[2] The Hacker News, "Critical mitel flaw lets hackers bypass login, gain full access to mivoice mx-one systems," https://thehackernews.com/2025/07/critical-mitel-flaw-lets-hackers-bypass.html, 2025.

[3] ——, "China-linked hackers exploit sap and sql server flaws in attacks across asia and brazil," https://thehackernews.com/2025/05/china-linked-hackers-exploit-sap-and.html, 2025.

[4] ——, "Fortinet releases patch for critical sql injection flaw in fortiweb," https://thehackernews.com/2025/07/fortinet-releases-patch-for-critical.html, 2025.

[5] CVE, "CVE Details," https://www.cvedetails.com/browse-by-date.php, 2026.

[6] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.

[7] A. Figueiredo, T. Lide, D. Matos, and M. Correia, "Merlin: Multi-language web vulnerability detection," in *Proceedings of the IEEE 19th International Symposium on Network Computing and Applications (NCA)*, 2020, pp. 1–9.

[8] R. Ramires, A. Respicio, and I. Medeiros, "Kave: A knowledge-based multi-agent system for web vulnerability detection," in *2024 IEEE International Conference on Web Services (ICWS)*, 2024, pp. 489–500.

[9] Y. Shi, Y. Zhang, T. Bai, L. Zhang, X. Tan, and M. Yang, "Recurscan: Detecting recurring vulnerabilities in php web applications," in *Proceedings of the ACM Web Conference (WWW'24)*, 2024, p. 1746–1755.

[10] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," in *In Proceedings of IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2339–2356.

[11] D. Noever, "Can large language models find and fix vulnerable software?" 2023. [Online]. Available: https://arxiv.org/abs/2308.10345

[12] M. Keltek, R. Hu, M. F. Sani, and Z. Li, "Lsast: Enhancing cybersecurity through llm-supported static application security testing," in *IFIP International Conference on ICT Systems Security and Privacy Protection*, 2025, pp. 166–179.

[13] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *Proceedings of the 13rd International Conference on Learning Representations*, 2025.

[14] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: literature review and the road ahead," *ACM Transactions on Software Engineering and Methodology*, 2024.

[15] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang, "Llms in software security: A survey of vulnerability detection techniques and insights," *ACM Comput. Surv.*, Sept 2025.

[16] S. Bashir, M. Abbas, M. Saadatmand, E. P. Enoiu, M. Bohlin, and P. Lindberg, "Requirement or not, that is the question: A case from the railway industry," in *Requirements Engineering: Foundation for Software Quality*, 2023, pp. 105–121.

[17] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–31, 2025.

[18] S. Joshi, "Open-source vs. commercial coding assistants: A 2025 comparison of deepseek r1, qwen 2.5 and claude 3.7," *International Journal of Computer Applications Technology and Research*, vol. 14, no. 09, pp. 6–18, 2025.

[19] I. Ahmed, S. Islam, P. P. Datta, I. Kabir, N. U. R. Chowdhury, and A. Haque, "Qwen 2.5: A comprehensive review of the leading resource-efficient llm with potential to surpass all competitors," Feb 2025.

[20] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Transactions on Software Engineering*, vol. 49, no. 12, pp. 5154–5188, 2023.

[21] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, "Automatic static vulnerability detection for machine learning libraries: Are we there yet?" in *Proceedings of the 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023, pp. 795–806.

[22] J. Zhao, K. Zhu, C. Lu, J. Zhao, and Y. Lu, "Benchmarking static analysis for php applications security," *Entropy*, vol. 27, no. 9, 2025.

[23] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios," *Computing*, vol. 101, no. 2, p. 161–185, Feb 2019.

[24] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen, "Comparison and evaluation on static application security testing (sast) tools for java," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, p. 921–933.

[25] G. Bennett, T. Hall, E. Winter, and S. Counsell, "Semgrep*: Improving the limited performance of static application security testing (sast) tools," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24, 2024, p. 614–623.

[26] W3Techs - Web Technology Surveys, "Usage statistics of server-side programming languages for websites," https://w3techs.com/technologies/overview/programming_language, 2025.

[27] W. G. Halfond, J. Viegas, A. Orso *et al.*, "A classification of sql injection attacks and countermeasures." in *Proceedings of the International Symposium on Secure Software Engineering (ISSE)*, 2006.

[28] C. Gould, Z. Su, and P. Devanbu, "Jdbc checker: a static analysis tool for sql/jdbc applications," in *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 697–698.

[29] I. Medeiros, N. Neves, and M. Correia, "Statically detecting vulnerabilities by processing programming languages as natural languages," *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 1033–1056, 2022.

[30] J. Guerreiro and I. Medeiros, "Processing web applications using nlp for vulnerability identification," in *In Proceedings of the 20th European Dependable Computing Conference Companion Proceedings (EDCC-C)*, Apr 2025, pp. 68–71.

[31] National Institute of Standards and Technology (NIST), "NIST Software Assurance Reference Dataset (SARD)," 2025. [Online]. Available: https://samate.nist.gov/SARD

[32] I. Medeiros and N. Neves, "Effect of coding styles in detection of web application vulnerabilities," in *16th European Dependable Computing Conference*, 2020, pp. 111–118.

[33] Rafael Ramires, "KAVe: Knowledge-Based Multi-Agent System Vulnerability Detector," https://github.com/rframires/KAVe.git, 2024.

[34] Sonar, "SonarQube," https://www.sonarsource.com/products/sonarqube/, 2026.

[35] OpenAI, "Gpt-4o," 2025, large language model. [Online]. Available: https://openai.com/product/gpt-4o

[36] N. Ding, Y. Qin, G. Yang, F. Wei, Z. Yang, Y. Su, S. Hu, Y. Chen, C.-M. Chan, W. Chen *et al.*, "Parameter-efficient fine-tuning of large-scale pre-trained language models," *Nature machine intelligence*, vol. 5, no. 3, pp. 220–235, 2023.

[37] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models." *ICLR*, vol. 1, no. 2, p. 3, 2022.

[38] Qwen Team, "Qwen 2.5-Coder," https://github.com/QwenLM/Qwen2.5-Coder, 2024.

[39] S. Wi, S. Woo, J. J. Whang, and S. Son, "HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs," in *Proceedings of the ACM Web Conference (WWW'22)*, 2022, p. 755–766.

[40] X. Li, Y. Li, F. Liu, and F. Zeng, "Research on the integration method of software static testing tools based on machine learning," in *Proceedings of the 24th International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2024, pp. 920–925.

[41] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, vol. 1, no. 2, 2023.

[42] M. Chen, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[43] S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith, "Don't stop pretraining: Adapt language models to domains and tasks," *arXiv preprint arXiv:2004.10964*, 2020.

[44] A. Chan, A. Kharkar, R. Z. Moghaddam, Y. Mohylevskyy, A. Helyar, E. Kamal, M. Elkamhawy, and N. Sundaresan, "Transformer-based vulnerability detection in code at edittime: Zero-shot, few-shot, or fine-tuning?" *arXiv preprint arXiv:2306.01754*, 2023.