STaint: Detecting Second-Order Vulnerabilities in PHP Applications with LLM-Assisted Bi-Directional Static Taint Analysis

Yuchen Ji
ShanghaiTech University
Shanghai, China
jiych2022@shanghaitech.edu.cn

Hongchen Cao
ShanghaiTech University
Shanghai, China
caohch2023@shanghaitech.edu.cn

Jingzhu He*
ShanghaiTech University
Shanghai, China
hejzh1@shanghaitech.edu.cn

***Abstract*—Second-order vulnerabilities, such as second-order Cross-Site Scripting (XSS) and Server-Side Request Forgery (SSRF), occur when user-controlled inputs are stored in databases and later retrieved in different execution contexts, complicating static detection. Existing static analysis approaches struggle primarily with two challenges. First, they struggle in accurately identifying database-accessing functions defined by third-party libraries or custom data access layers, often leading to missed taint propagation paths. Second, they may fail to contextually model database operations when queries are dynamically constructed and depend on runtime parameters. To address these limitations, we propose STAINT, a novel bi-directional static analysis method that integrates taint analysis with large language models (LLMs). Using semantic reasoning, STAINT accurately identifies and models custom database reads and writes, effectively reconstructing comprehensive taint data flows in the database. Preliminary evaluations on ten real-world PHP applications show that STAINT successfully detects 56 second-order vulnerability paths, including 7 previously unknown cases, outperforming existing techniques.**

***Index Terms*—Second-order vulnerabilities, Taint analysis, PHP**

I. INTRODUCTION

PHP remains one of the most widely used server-side languages for web applications, such as content management systems (CMS) and e-commerce platforms [1]. These applications frequently accept user input for use in security-critical operations. When untrusted data from an external source propagates through an application and is used in a sensitive operation without proper sanitization, taint-style vulnerabilities occur [2]. Common examples include SQL injection, Cross-Site Scripting (XSS), and Server-Side Request Forgery (SSRF). The consequences of taint-style vulnerabilities are severe, ranging from sensitive data leakage and session hijacking to complete server compromise. For example, a 2019 SSRF vulnerability in a Capital One service led to the leakage of data from over 100 million user accounts, resulting in an $80 million regulatory penalty [3].

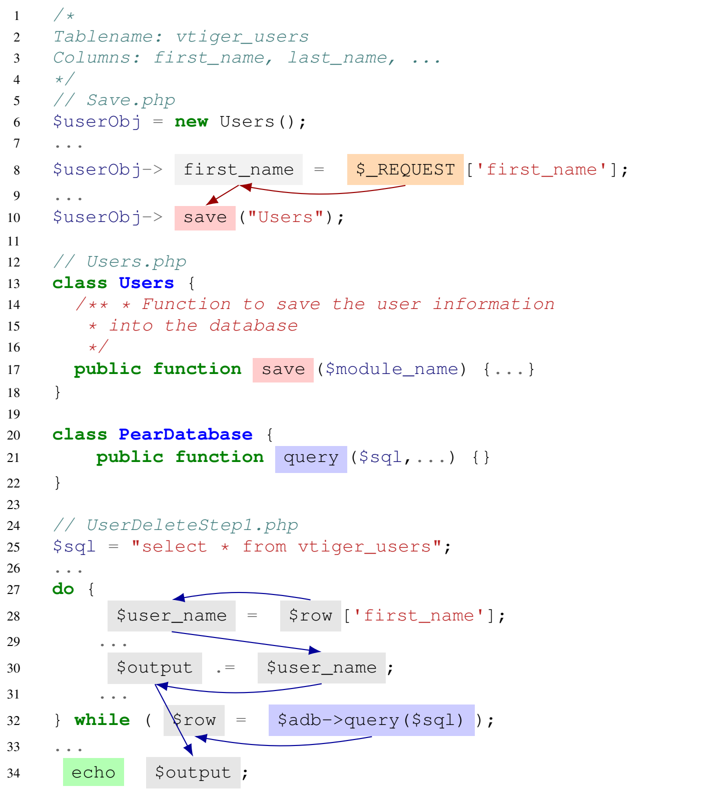
Taint-style vulnerabilities can be classified by their execution path [4]. Vulnerabilities triggered immediately within a single request are known as first-order and have been extensively studied [5]–[12]. In contrast, second-order vulnerabilities occur when a malicious payload is first stored in a database, and later retrieved and executed in a different context [4]. These vulnerabilities are explored less often in static analysis due to the significant challenges involved in tracking data across separate program executions. For instance, a second-order XSS attack might involve injecting a payload into a user profile, which is stored in the database and later rendered on an administrator's dashboard, executing the malicious script in a privileged context.

To identify second-order vulnerabilities using static taint analysis, accurate modeling of data flows through databases is required. Existing work such as RIPS [13], PHPJOERN [14] and TCHECKER [2] have established comprehensive taint propagation rules for PHP but focused primarily on first-order vulnerabilities. However, extending their analysis to second-order vulnerabilities is non-trivial.

Previous work has attempted to bridge this gap. Johannes et al. [15] first integrated SQL parsing with taint analysis, and TORPEDO [16] later combined string analysis with SQL parsing. SPLENDOR [4] introduced heuristic token matching to identify database operations without the need for explicit SQL strings. Beyond PHP, DBridge [17] developed a pointer analysis for Java database-backed applications, but relied on manually specified database access APIs for frameworks such as JDBC. Despite these advances, two major challenges persist for static taint analysis.

The first challenge is **recognizing custom database functions**. Although PHP provides built-in database methods such as `PDO::query`, real-world applications heavily rely on custom Data Access Layers (DALs) or third-party libraries to abstract database interactions [4]. Tools such as the updated RIPS [15] and TORPEDO [16] only recognize built-in functions, leading to false negatives when custom wrappers are used. SPLENDOR [4] partially addresses this by heuristically identifying the most frequently called functions as database function candidates, but this approach remains unreliable for diverse, project-specific abstractions. For example, in Figure 1, the `Users::save` method on line 10 is a custom DAL function that existing tools would fail to identify as a database

* Jingzhu He is the corresponding author.



```
1   /*
2   Tablename: vtiger_users
3   Columns: first_name, last_name, ...
4   */
5   // Save.php
6   $userObj = new Users();
7   ...
8   $userObj-> first_name  =  $_REQUEST ['first_name'];
9   ...
10  $userObj-> save ("Users");
11
12  // Users.php
13  class Users {
14    /** * Function to save the user information
15     * into the database
16     */
17    public function  save ($module_name) {...}
18  }
19
20  class PearDatabase {
21      public function  query ($sql,...) {}
22  }
23
24  // UserDeleteStep1.php
25  $sql = "select * from vtiger_users";
26  ...
27  do {
28      $user_name  =  $row ['first_name'];
29      ...
30      $output  .=  $user_name ;
31      ...
32  } while ( $row  =  $adb->query($sql) );
33  ...
34   echo   $output ;
```

Fig. 1. A second-order vulnerability in Corebos. → represents taint flows from source to database write function, while → represents taint flows from database read function to sink.

write operation, thus breaking the taint flow analysis.

The second challenge is **accurately modeling the contextual behavior of database operations**. Static analysis needs to accurately determine the specific database tables and columns affected by each operation to correctly trace second-order vulnerabilities [4]. Tools like RIPS and TORPEDO primarily rely on static SQL string extraction and parsing and thus struggle to model dynamic or abstracted queries constructed at runtime. SPLENDOR's fuzzy token-matching approach partially alleviates this by identifying likely SQL tokens in source code tokens. However, the technique in SPLENDOR remains ineffective when code tokens do not directly reflect common SQL tokens. For example, in Figure 1, user input taints the `first_name` attribute of `$userObj` on line 8. Accurate analysis must recognize that invoking `Users::save("Users")` (line 10) results in a database write to the `first_name` column of the `vtiger_users` table. RIPS and TORPEDO fail here because there are no static SQL statements. SPLENDOR fails here because recognizable SQL operation tokens such as `SELECT` or `UPDATE` are absent.

To overcome these limitations, this paper presents STAINT, a novel bidirectional static analysis approach that leverages Large Language Models (LLMs) to detect second-order vulnerabilities in PHP applications. While recent work has successfully used LLMs to infer project-specific taint specifications [18]–[20] or validate bug reports [21], our work applies LLMs to the problem of identifying and modeling the semantics of database interactions at their call sites in PHP code. STAINT first uses bidirectional static analysis to identify taint flow paths both from sources to database writes and from database reads to sensitive sinks. Then, from the taint flow paths and corresponding source code, LLMs are used to identify custom database functions and model their contextual behavior at each call site. Finally, complete taint flow paths involving databases can be connected by pairing the modeled database operations, and LLMs recognize custom sanitizers on taint flow paths to reduce false positives. The preliminary evaluation shows that our approach enables the semantic interpretation of database operations in highly abstracted code, improving the detection rate of second-order vulnerabilities.

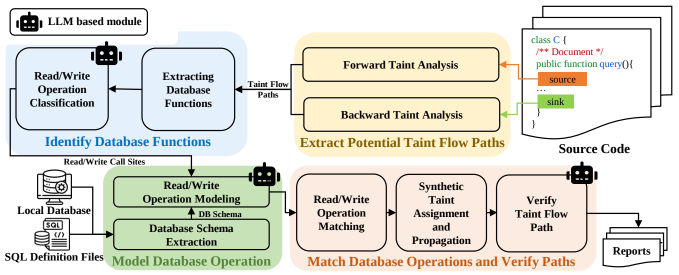

LLM based module
Read/Write Operation Classification
Extracting Database Functions
Taint Flow Paths
Identify Database Functions
Forward Taint Analysis
Backward Taint Analysis
Extract Potential Taint Flow Paths
class C {
/** Document */
public function query(){
source
...
sink
}
}
Source Code
Read/Write Call Sites
Local Database
SQL Definition Files
Read/Write Operation Modeling
DB Schema
Database Schema Extraction
Model Database Operation
Read/Write Operation Matching
Synthetic Taint Assignment and Propagation
Verify Taint Flow Path
Match Database Operations and Verify Paths
Reports

Fig. 2. The overall architecture of STAINT.

II. METHODOLOGY

STAINT is structured into four phases: *Potential Taint Flow Path Extraction*, *Database Function Identification*, *Database Operation Modeling*, and *Database Operation Matching*. Figure 2 illustrates the overall architecture. First, STAINT extracts disconnected taint flow paths using bidirectional taint analysis. Next, STAINT identifies custom database interaction functions. Then, STAINT models the semantics of database operations at call sites using schema information. Finally, STAINT reconstructs second-order flows by matching reads and writes and applies LLM-based verification to eliminate sanitized paths, producing the final detection results.

A. Phase 1: Extract Potential Taint Flow Paths

STAINT first extracts disconnected taint flow paths that may later form complete second-order vulnerabilities. Since such vulnerabilities involve writing tainted data into a database and later reading it, traditional analysis produces two disjoint paths: one from sources to database writes and another from database reads to sinks.

To obtain these paths, we perform bidirectional taint analysis on the application's Code Property Graph (CPG) [22]. Predefined sources (e.g., `$_GET`) and sinks (e.g., `echo`) follow prior work [2], [4], [13], [14], [20]. The output of this phase is a collection of possible second-order vulnerability paths, such as Cross-Site Scripting (XSS), SSRF, and SQL Injection.

B. Phase 2: Identify Database Functions

In second phase, STAINT identifies custom database interaction functions. While built-in PHP functions [23] are predefined and easily classified, project-specific wrappers and third-party functions often evade existing tools. STAINT addresses this gap by analyzing functions present in the extracted taint flow paths with LLMs.

We begin by collecting all functions located on the collected taint flow paths. Built-in functions are filtered out since their classifications are predefined. The remaining functions form a refined candidate set, denoted as API_A.

Each candidate is then classified using an LLM. The input to the LLM consists of the function signature, PHPDoc comments, and the full implementation source code. The model's task is to categorize the function as a *read*, *write*, *both*, or *none*. The output yields two subsets: API_w, containing database write functions, and API_r, containing database read functions.

For instance, in Figure 1, the bidirectional taint analysis captures the functions `save` (line 10) and `query` (line 32). Their respective definitions (lines 17 and 21) are provided to the LLM, which then classifies `save` as a database write and `query` as a read/write function.

C. Phase 3: Model Database Operations

Identifying a function as a database interaction is insufficient for connecting taint flow paths since we need to model which tables and columns the operation involves in order to pair them.

Before taint analysis, we extract the application's database schema. If schema definition files (`.sql`) are available, they are executed in a local database environment to obtain accurate schema information, including table and column names. If not, we manually run the application once and dump the resulting schema. This schema serves as the ground truth provided to LLMs for inferring the exact database elements referenced in operations.

Then, the LLM is prompted to model the semantics of each database interaction call site within the taint flow paths. For writes in API_w, the input to the LLM includes the function source code, the relevant taint propagation paths reaching the call site, and the schema information. The output specifies the location of the call site, the table name, and the affected columns, stored in the set S_w. Similarly, for reads in API_r, the LLM receives the function code, taint context, and schema information. The output includes the referenced tables, columns, and return type (e.g., associative array or object), which are stored in the set S_r. These return types are crucial for synthesizing taint in subsequent verification analysis.

For example, the call to `save` (line 10 in Figure 1) is modeled as writing to the `first_name` column of the `vtiger_users` table. Conversely, the call to `query` (line 33) is modeled as reading all columns from the same table.

D. Phase 4: Match Database Operations and Verify Paths

The final phase reconstructs complete second-order taint flow paths by matching database writes to subsequent reads and verifying the resulting flows.

TABLE I
Second-order vulnerability detection results. SPLENDOR's results are taken from their paper [4] and are shown for comparison. STAINT was also evaluated on five additional applications not previously tested. Newly detected vulnerabilities are marked with green background.

Application	**STAINT TP**	**STAINT FP**	**Splendor TP**	**Splendor FP**
phpBB v2.0.23	4	2	3	2
PunBB v1.4	5	0	4	0
Catfish v5.4.0	6	0	4	2
osCommerce v2.3.3.4	25	2	7	2
Corebos v5.5	5	2	2	0
Total (Comparison)	**45**	**6**	**20**	**6**
yzmcms v5.3.0	3	1	/	/
Royal Elementor Addons v1.7.1013	3 (new)	1	/	/
WP-DownloadManager v1.68.4	1	0	/	/
MetForm v3.9.7	3 (new)	0	/	/
POST SMTP Mailer v2.8.5	1 (new)	0	/	/
Grand Total (Ours)	**56**	**8**		

The first step involves matching each modeled write operation in S_w with corresponding reads in S_r based on table and column names. Wildcard reads (*) are treated as matching any column in the specified table. A successful match indicates the presence of a potential second-order taint flow through the database.

After a match is identified, it must be verified whether the tainted column is actually used at the read site. Since a read may fetch all columns but only a subset is tainted, synthetic taint assignments are introduced to explicitly track column-level propagation. For example, if a `query` retrieves all columns, a synthetic assignment such as `$row['first_name'] = $_TAINTED;` is inserted to determine if the tainted value flows into a sink. A subsequent forward taint analysis then verifies whether this synthetic taint reaches a sensitive sink, thus confirming a true second-order vulnerability.

To minimize false positives caused by sanitizers, STAINT employs LLM-based sanitizer identification. For each reconstructed taint flow, the LLM is provided with code snippets along the path. Instead of the entire codebase, which would exceed context limits, inspired by prior work [21], [24], we provide two retrieval tools: a text search tool for locating relevant snippets and a function extraction tool for retrieving complete function definitions by line number. With function calling capability, the LLM iteratively queries these tools to gather sufficient context and then determines whether the flow is neutralized by sanitization. The output of this process is the final filtered set of confirmed vulnerabilities.

III. PRELIMINARY EXPERIMENTS

We implemented a prototype of STAINT, integrating Joern [25] for Code Property Graph (CPG) generation, Phan [26]

```
1  function redirect($destination_url, $message) {
2    ...
3    echo $destination_url;
4  }
5
6  function sef_friendly($str) {
7    ...
8    $str = preg_replace(
9      ['/[^a-z0-9\s]/', '/[\s]+/'],
10     ['', '-'],
11     $str);
12   ...
13   return $str;
14 }
15
16 $cur_post['subject'] = $_TAINTED;
17 ...
18 redirect(sef_friendly($cur_post['subject']));
```

Fig. 3. A false positive pruned by STaint.

for static taint analysis, and GPT-4 [27] for semantic reasoning tasks. To evaluate STaint, we conducted experiments on 10 PHP web applications. Our comparison focuses on Splendor, the state-of-the-art approach for second-order PHP vulnerability detection, which has demonstrated superior performance over prior methods such as Rips [15]. Five of the selected applications were previously analyzed by Splendor [4], allowing direct comparison, while the remaining five were newly selected from GitHub and the WordPress Plugin Repository to ensure diversity. To identify new vulnerabilities, we cross-referenced detected vulnerabilities against reports from Splendor, the CVE database, and the application issue trackers. Only vulnerabilities absent from all three sources are considered new and reported to the respective developers.

We measured performance by identifying true positives (TP), i.e., vulnerabilities correctly reported by STaint, and false positives (FP), representing benign paths incorrectly flagged. Table I presents a comparative analysis between STaint and Splendor. Overall, our method detected 56 true second-order vulnerability paths, including 7 previously unknown cases, with 8 false positives, yielding a precision of 87.5%. The 7 unknown cases have been reported to developers and 4 confirmed.

Due to the unavailability of the complete source code of Splendor, direct comparison is limited to five applications analyzed previously. In this subset, STaint identified 45 second-order vulnerability paths, more than doubling the 20 paths reported by Splendor. This substantial improvement demonstrates the effectiveness of our LLM-based approach in addressing the two key challenges of second-order vulnerability detection.

The better detection coverage of STaint is mainly due to its ability to identify and model custom database functions that existing tools lack. For example, in Figure 1, Splendor fails to recognize the database function on line 10 because it is not among the top three most frequently called functions, thus missing this vulnerability. Even if Splendor could recognize `Users::save`, it would fail to model the operation, since the call does not contain tokens contained in traditional SQL statements such as `INSERT` or `UPDATE`. Instead, `save` is used to indicate a database write operation, while object fields such as `first_name` indicate the affected columns. This pattern is common in modern PHP applications, where database operations are abstracted behind semantic method names. On the other hand, STaint's bidirectional taint analysis combined with LLM-based semantic understanding enables it to understand the semantic relationship between function parameters, object properties, and database schema. The database operation is then modeled without the actual SQL statements. As shown in Figure 1, the table name is inferred from the class name `User`, the column name is inferred from the property name `first_name`, while the write operation is inferred from the method name `save`.

STaint's false positive rate depends on the LLM's ability to identify and interpret custom sanitization functions. As shown in Figure 3, the variable `$cur_post['subject']` is tainted and propagates to a sink on line 3 in the function `redirect` on line 18. Although a second-order XSS vulnerability is flagged, a custom sanitizer `sef_friendly` on line 6 removes non-alphanumeric characters and converts spaces to dashes. The LLM correctly identifies the sanitizing logic and determines that an alphanumeric and dash-only payload cannot trigger XSS, thus pruning the false positive. However, false positives still occur, particularly when applications impose length limitations that make exploitation unfeasible. For example, one false positive involved usernames truncated before output using `substr($name, 0, 5)` length limit, which makes XSS exploits unfeasible.

IV. Conclusion and Future Work

This paper presents a novel method for statically detecting second-order vulnerabilities such as second-order XSS and SSRF in PHP web applications. Our approach integrates taint analysis with the semantic reasoning capabilities of large language models (LLMs) to identify and model database operations hidden within user-defined APIs, and finally to reconstruct taint flows that pass through database. Preliminary evaluations show that our method outperforms existing tools like Splendor, identifying 56 second-order vulnerability paths in 10 real-world applications, 7 of which are new.

In future work, we will address the limitations of STaint. First, to reduce false positives, we will use context-sensitive analysis to pinpoint when tainted inputs are constrained (e.g. by limiting length), preventing exploitation. Second, due to the computational demands of large language models on large codebases, we will explore lighter, locally hosted, or fine-tuned models for specific tasks. Third, manual schema extraction limits scalability. Previous research [4] shows that only about 30% of applications have explicit `.sql` files for automatic extraction. To enhance scalability, we plan to integrate LLM-agent-based automated techniques such as ExecutionAgent [28] to set up the project automatically and then dump the schema.

References

[1] W3Techs. (2025) Usage statistics of php for websites. [Online]. Available: https://w3techs.com/technologies/details/pl-php

[2] C. Luo, P. Li, and W. Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2175–2188.

[3] B. Krebs. (2019) What we can learn from the capital one hack. [Online]. Available: https://krebsonsecurity.com/2019/08/what-we-can-learn-from-the-capital-one-hack/

[4] H. Su, F. Li, L. Xu, W. Hu, Y. Sun, Q. Sun, H. Chao, and W. Huo, "Splendor: Static detection of stored xss in modern web applications," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1043–1054.

[5] S. Lekies, B. Stock, and M. Johns, "25 million flows later: large-scale detection of dom-based xss," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1193–1204.

[6] M. Johns, B. Engelmann, and J. Posegga, "Xssds: Server-side detection of cross-site scripting attacks," in *2008 Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 335–344.

[7] S. Bensalim, D. Klein, T. Barber, and M. Johns, "Talking about my generation: Targeted dom-based xss exploit generation using dynamic data flow analysis," in *Proceedings of the 14th European Workshop on Systems Security*, ser. EuroSec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 27–33.

[8] S. Khodayari and G. Pellegrino, "It's (dom) clobbering time: Attack techniques, prevalence, and defenses," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1041–1058.

[9] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-origin state inference (cosi) attacks: Leaking web site states through xs-leaks," February 2020.

[10] T. Van Goethem, I. Sanchez-Rola, and W. Joosen, "Scripted henchmen: Leveraging xs-leaks for cross-site vulnerability detection," in *2023 IEEE Security and Privacy Workshops (SPW)*, 2023, pp. 371–383.

[11] R. Focardi, F. L. Luccio, and M. Squarcina, "Fast sql blind injections in high latency networks," in *2012 IEEE First AESS European Conference on Satellite Telecommunications (ESTEL)*, 2012, pp. 1–6.

[12] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. N. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1059–1076.

[13] J. Dahse and T. Holz, "Simulation of built-in php features for precise static code analysis." in *NDSS*, vol. 14, 2014, pp. 23–26.

[14] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of php application vulnerabilities," in *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 2017, pp. 334–349.

[15] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 989–1003.

[16] O. Olivo, I. Dillig, and C. Lin, "Detecting and exploiting second order denial-of-service vulnerabilities in web applications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 616–628.

[17] Y. Liang, T. Zhang, G. Li, T. Tan, C. Xu, C. Cao, X. Ma, and Y. Li, "Pointer analysis for database-backed applications," *Proc. ACM Program. Lang.*, vol. 9, no. PLDI, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3729307

[18] Z. Li, S. Dutta, and M. Naik, "Iris: Llm-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025.

[19] F. Liu, Y. Zhang, T. Chen, Y. Shi, G. Yang, Z. Lin, M. Yang, J. He, and Q. Li, "Detecting taint-style vulnerabilities in microservice-structured web applications," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2025, pp. 972–990.

[20] Y. Ji, T. Dai, Z. Zhou, Y. Tang, and J. He, "Artemis: Toward accurate detection of server-side request forgeries through llm-assisted inter-procedural path-sensitive taint analysis," *Proceedings of the ACM on Programming Languages*, vol. 9, no. OOPSLA1, pp. 1349–1377, 2025.

[21] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Assisting static analysis with large language models: A chatgpt experiment," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 2107–2111.

[22] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[23] PHP. (2025) Function and method listing. [Online]. Available: https://www.php.net/manual/en/indexes.functions.php

[24] OpenAI, "Function calling," https://platform.openai.com/docs/guides/function-calling, 2025.

[25] joernio. (2025) Joern. [Online]. Available: https://github.com/joernio/joern

[26] phan. (2025) phan. [Online]. Available: https://github.com/phan/phan

[27] OpenAI, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[28] I. Bouzenia and M. Pradel, "You name it, i run it: An llm agent to execute tests of arbitrary projects," *Proc. ACM Softw. Eng.*, vol. 2, no. ISSTA, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3728922