Artemis: Toward Accurate Detection of Server-Side Request Forgeries through LLM-Assisted Inter-procedural Path-Sensitive Taint Analysis

YUCHEN JI, ShanghaiTech University, China
TING DAI, IBM Research, USA
ZHICHAO ZHOU, ShanghaiTech University, China
YUTIAN TANG, University of Glasgow, United Kingdom
JINGZHU HE*, ShanghaiTech University, China

Server-side request forgery (SSRF) vulnerabilities are inevitable in PHP web applications. Existing static tools in detecting vulnerabilities in PHP web applications neither contain SSRF-related features to enhance detection accuracy nor consider PHP's dynamic type features. In this paper, we present Artemis, a static taint analysis tool for detecting SSRF vulnerabilities in PHP web applications. First, Artemis extracts both PHP built-in and third-party functions as candidate source and sink functions. Second, Artemis constructs both explicit and implicit call graphs to infer functions' relationships. Third, Artemis performs taint analysis based on a set of rules that prevent over-tainting and pauses when SSRF exploitation is impossible. Fourth, Artemis analyzes the compatibility of path conditions to prune false positives. We have implemented a prototype of Artemis and evaluated it on 250 PHP web applications. Artemis reports 207 true vulnerable paths (106 true SSRFs) with 15 false positives. Of the 106 detected SSRFs, 35 are newly found and reported to developers, with 24 confirmed and assigned CVE IDs.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Security and privacy** → **Web application security**.

Additional Key Words and Phrases: PHP, server-side request forgery, taint analysis

ACM Reference Format:
Yuchen Ji, Ting Dai, Zhichao Zhou, Yutian Tang, and Jingzhu He. 2025. Artemis: Toward Accurate Detection of Server-Side Request Forgeries through LLM-Assisted Inter-procedural Path-Sensitive Taint Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 128 (April 2025), 29 pages. https://doi.org/10.1145/3720488

1 Introduction

PHP is currently the dominant programming language for building web applications [19]. PHP web applications allow developers to use server-side requests to interact with third-party applications [66, 69]. Attackers often manipulate user input to send forged server-side requests, leading to the exploitation of server-side request forgery (SSRF) vulnerabilities. Attackers pretend like the server sends the request, bypassing access controls such as firewalls that prevent direct access to specific URLs [11]. Exploitation of SSRF vulnerabilities can lead to severe consequences for applications,

*Jingzhu He is the corresponding author.

Authors' Contact Information: Yuchen Ji, ShanghaiTech University, Shanghai, China, jiych2022@shanghaitech.edu.cn; Ting Dai, IBM Research, Yorktown Height, USA, ting.dai@ibm.com; Zhichao Zhou, ShanghaiTech University, Shanghai, China, zhouzhch@shanghaitech.edu.cn; Yutian Tang, University of Glasgow, Glasgow, United Kingdom, yutian.tang@glasgow.ac.uk; Jingzhu He, ShanghaiTech University, Shanghai, China, hejzh1@shanghaitech.edu.cn.

ACM 2475-1421/2025/4-ART128
https://doi.org/10.1145/3720488

including denial of service (DoS), leakage of sensitive data, and remote code execution [69]. In 2019, the exploitation of an SSRF vulnerability in Capital One's service resulted in the leakage of credit card information for over 100 million consumers [1]. Existing research has explored the detection and prevention of various vulnerabilities in web applications, including cross-site scripting (XSS) [34, 49, 55, 58, 77, 81], SQL injection [44, 50], DoS [61, 76], cross-site request forgery (CSRF) [53, 54, 70, 79], prototype injection [51, 59, 60, 62], business logic flow vulnerabilities [36–38, 42, 45, 52, 56, 74], and recurring vulnerabilities [72, 73]. However, limited research focuses on SSRF detection, even though reported SSRF cases are rapidly increasing and the impacts are severe, as illustrated in Figure 1. Since 2021, SSRFs have been ranked as the top 10 vulnerabilities by OWASP, based on the occurrence, impacts, incident rates, and number of CVEs [24].

Static taint analysis tools [7, 27, 33, 39, 64] are widely used by developers to detect vulnerabilities in PHP web applications. The key idea is to track the flow of sources containing untrusted user input and check whether the tainted data reaches specified sink functions that send server-side requests. However, using existing static taint analysis tools to detect SSRFs in PHP results in high false positives and false negatives due to four key challenges.

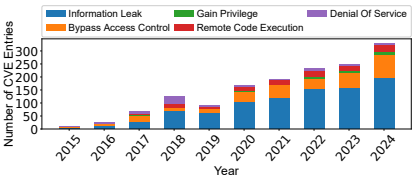


Fig. 1. The statistics of SSRFs in recent 10 years with impacts.

First, existing tools only consider PHP's built-in sources and sinks. In our preliminary study, we find that 61% (153 in 250) applications use third-party APIs to handle user input or send server-side requests. Existing tools produce false negatives without considering third-party sources and sinks. Manually extracting these sources and sinks is time-consuming and does not scale.

Second, while the implicit call graph construction has been researched recently in statically typed languages [71], existing PHP static analysis tools often fail to resolve implicit call targets in their call graphs. For example, magic methods [14], automatically called for undefined object methods, and dynamic constructs like variable classes and methods [30], are ignored. Failing to include implicit method calls when constructing call graphs leads to the misdetection of SSRFs. For example, Rips [39] overlooks method calls because Rips generates call graphs by matching function signatures, disregarding object-oriented methods. PHPJoern [33] constructs call graphs by identifying unique method names. When a method name is not unique, i.e., several classes have methods with the same name, PHPJoern disregards the call. TChecker [64], Phan [7] and Psalm [27] all generate call graphs using type inference and account for object-oriented methods. However, if the variable type cannot be statically inferred due to features such as reflection, any method call on that variable is ignored. Additionally, magic methods are ignored.

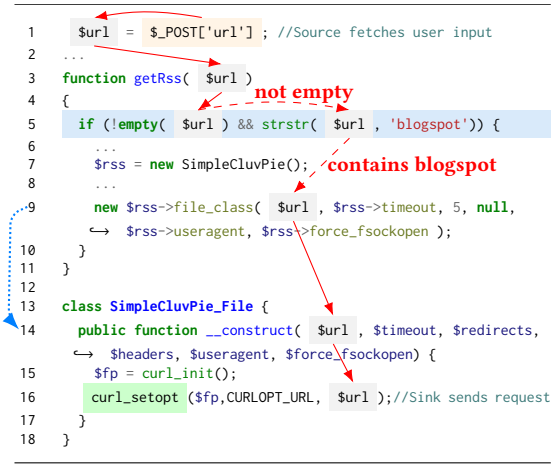
Third, existing tools contain both over-tainting and under-tainting rules. Over-tainting rules overlook data flow paths and string sanitizations, which may result in non-vulnerable code being flagged as tainted. Under-tainting rules exclude certain data structures, code blocks, and indirect paths, which can lead to missed vulnerabilities. For example, Rips recklessly marks the return value of a function as tainted if any argument is tainted, regardless of whether the argument actually affects the return value through data flows. TChecker treats functions that are not connected in the call graph as dead code and omits their analysis. However, bypassed functions can be invoked by reflection in third-party libraries and may contain vulnerabilities. Phan and Psalm taint a string if any of its components are tainted, even if the string is not a URL.

Fourth, existing tools do not consider path conditions of SSRF exploitation, leading to false positives. Specifically, two types of path conditions prevent attacker-controlled URLs from reaching the sink functions. First, the sinks may be on an impossible branch. Second, path conditions may

forbid attacker-controlled URLs. Although several existing tools, such as RIPS, TCHECKER, and PHAN, attempt to model specific sanitization rules for sanitizer functions, these tools do not model path conditions effectively because they solely rely on name matching to recognize sanitizers.

1.1 A Motivating Example

We present a previously unknown SSRF vulnerability in Figure 2 to illustrate how it is exploited and how to detect it using static taint analysis. The application takes a user input at line #1. The input URL reaches the request-sending function on line #16 if it is non-empty and contains the string `blogspot`. An attacker could exploit the vulnerability by crafting a URL like `http://blogspot.evil.com`, where `evil.com` is a domain controlled by the attacker and resolves to `127.0.0.1`. Thus, this URL bypasses the string check and causes the server to send requests to internal addresses. Exploitation of this SSRF causes sensitive data leakage.




```
1   $url = $_POST['url'] ; //Source fetches user input
2   ...
3   function getRss( $url )
4   {
5     if (!empty( $url ) && strstr( $url , 'blogspot')) {
6       ...
7       $rss = new SimpleCluvPie();
8       ...
9       new $rss->file_class( $url , $rss->timeout, 5, null,
      ↪  $rss->useragent, $rss->force_fsockopen );
10    }
11  }
12
13  class SimpleCluvPie_File {
14    public function __construct( $url , $timeout, $redirects,
      ↪  $headers, $useragent, $force_fsockopen) {
15      $fp = curl_init();
16      curl_setopt ($fp,CURLOPT_URL, $url );//Sink sends request
17    }
18  }
```

Fig. 2. A new SSRF in CommentLuv v3.0.4 whose taint propagation path from source to sink contains path conditions and implicit call flows. ⇢ represents implicit call flows. → represents data flows. ⇢ represents (controlled) data flows with path conditions.

To detect this SSRF vulnerability using taint analysis, we taint the built-in source `$_POST` that takes a user input at line #1 and the built-in sink `curl_setopt` that sends the server-side request at line #16. We aim to find whether there exists a feasible path from the tainted source to sink, like the path indicated by the red solid lines in Figure 2.

However, existing taint analysis tools fail to detect this vulnerability due to two reasons. First, the implicit call flow between line #9 and line #14 is obscured by the dynamically assigned class name `$rss->file_class`, making it difficult for static analysis tools to build a complete call graph. As a result, the taint flow terminates at line #9. Second, existing tools fail to consider the path conditions within an if branch at line #5. The path condition performs a string check on the `$url`. Only when `$url` contains the sub-string `blogspot`, there exists a feasible path from line #5 to line #9, making the sink function reachable.

ARTEMIS overcomes the above challenges to detect this SSRF. First, by examining all `__construct` methods within the application (keyword `new` represents a call to the constructor method), ARTEMIS resolves the implicit call from line #9 to line #14, identifying the correct class (`SimpleCluvPie_File`) based on the constructor's argument count because the constructor on line #14 is the only constructor that accepts six parameters. Second, ARTEMIS extracts the path condition and analyzes the string check imposed by `strstr`. ARTEMIS confirms that attacker-controlled URLs containing `blogspot` reach the sink function, making exploitation feasible.

1.2 Contributions

In this work, we present ARTEMIS[1], a holistic taint analysis tool to detect SSRFs in PHP web applications. ARTEMIS consists of four integrated modules: 1) *source and sink identification*; 2)

[1]Artemis is the Greek goddess of the hunt who defeated Apate, the goddess of forgery.

statically inferred call graph construction; 3) *rule-based taint analysis*; and 4) *false positive pruning*. First, Artemis extracts both PHP built-in and third-party source and sink functions. For functions in third party libraries, Artemis extracts all the functions' signatures with PHPDoc comments [17]. Artemis then leverages a large language model (LLM) to further determine the candidate sources and sinks. Second, Artemis constructs the call graphs considering PHP's dynamically typed features. Specifically, Artemis builds the implicit caller-callee relationships statically when the callee is implemented using magic methods or the callee has a variable class or method name. Third, Artemis performs taint analysis to report candidate paths from sources to sinks based on augmented propagation rules to prevent over-tainting and under-tainting. Artemis also constructs implicit data flows to improve detection coverage. Moreover, Artemis applies unique safety string assurance rules to eliminate the tainted paths along which SSRF exploitation is impossible. Lastly, Artemis prunes false positives by conducting path condition analysis to eliminate infeasible paths with incompatible conditions. Specifically, Artemis identifies always unsatisfied conditions and URL rejection conditions from the extracted path conditions. During result validation, we use a multi-turn LLM conversation to create concrete SSRF exploits for Artemis's reported vulnerable paths. Automatically generating exploits with LLM significantly reduces the manual effort required to confirm taint analysis results. Artemis makes the following contributions:

- **Third-party sources and sinks extraction.** We leverage an LLM-assisted method to augment the set of SSRF sources and sinks by incorporating third-party library APIs besides PHP built-in functions. We show that 50% SSRF vulnerabilities detected by Artemis are introduced by third-party libraries.
- **Implicit call graph construction schemes.** We observe that 13% call targets are implicit including magic methods, variable class names of callees, and variable method names of callees in widely used PHP applications. We develop a set of implicit call graph construction schemes through slight over-approximation. The results show that Artemis detects 17 additional vulnerabilities without introducing any new false positives when incorporating implicit call graph construction.
- **Augmented taint analysis rules.** We augment taint propagation rules to prevent over-tainting and under-tainting. We incorporate PHP array semantics and develop array-specific taint rules. We develop PHP-specific implicit data flow reconstruction rules. Moreover, we analyze the strings to eliminate the tainted paths along which SSRF exploitation is impossible. The results show that the augmented taint rules increase the detection coverage by 4.5% and reduce the false positives by 90.2% for reported paths.
- **SSRF-specific false positive pruning.** We develop simple but effective SSRF-specific schemes to eliminate infeasible paths with incompatible constraints purely statically. Specifically, we extract path conditions of tainted paths and prune paths with always unsatisfied and URL rejection conditions to reduce false positives. Our false positive pruning schemes reduce the number of reported false positive paths from 35 to 15.

We have implemented a prototype of Artemis and evaluated it on 250 PHP web applications. Our results show that Artemis reports 207 true vulnerable paths (106 true SSRFs) and 15 false positives, significantly outperforming existing tools. Among the 106 detected SSRFs, 35 are new, and 24 of them have been confirmed by developers with assigned CVE IDs.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the design details of the Artemis system. Section 4 presents the experimental evaluation. Finally, the paper concludes in Section 5. Omitted details of design and evaluation can be found in the full version of this paper [48].

2 Related Work

Taint-based Vulnerability Detection. Existing work has designed generic vulnerability detection tools based on taint analysis. Rips [39] performed taint analysis and built call graphs by matching function signatures and specified over 900 propagation rules for PHP built-in functions. However, it ignored object-oriented methods when building call graphs. PHPJoern [33] improved call graph construction by analyzing object-oriented methods. However, when functions in different classes had identical names, PHPJoern could not distinguish them and omitted these functions. TChecker [64] is built on PHPJoern by enhancing call graph construction for object-oriented methods through type inference. TChecker also added propagation rules for class field variables during taint analysis. However, if a variable's type cannot be statically inferred, TChecker ignores its method calls. WAP [65] utilized supervised machine learning methods to prune false positive patterns on the vulnerable paths identified by taint analysis, which requires manual effort to collect vulnerable code samples for training. Compared to prior work, Artemis recognizes third-party sources and sinks, constructs implicit call graphs, reduces both over-tainting and under-tainting during taint analysis, and prunes false positives caused by path conditions. As a result, Artemis detects more SSRFs with significantly fewer false positives and does not rely on manual efforts such as collecting vulnerable code samples. Existing work has also introduced taint-based detection tools for specific types of vulnerability. Splendor [75] analyzed database operations in propagation paths using a heuristic token-matching strategy to detect second-order XSS vulnerabilities in PHP applications. Torpedo [67] investigated whether retrieved string values from the database could lead to DoS vulnerabilities using string analysis. In contrast to these tools, Artemis focuses on SSRF.

Fuzzing-based Vulnerability Detection. Existing work has also explored vulnerability detection approaches based on fuzzing. For example, Ufuzzer [47] targeted file-upload vulnerabilities by fuzzing inputs to reach sink functions along the propagation paths identified by symbolic execution tools. fuse [57] crafted mutation strategies focused on modifying standard upload requests to detect file-upload vulnerabilities in applications built in various languages. Navex [32] performed taint analysis to identify vulnerable paths and then fuzzed input to detect vulnerabilities. witcher [80] increased code coverage during fuzzing by modifying user inputs to generate SQL special characters and shell commands, aiming to detect SQL injections and code injections that could lead to remote code execution in applications built with various languages. Atropos [46] used a feedback-driven approach by modifying the PHP interpreter to generate logs that guide fuzzing mutations to detect potential vulnerabilities such as XSS and object injection in PHP web applications. Pellegrino et al. [69] examined the security implications of server-side requests (SSR) and developed a black-box fuzzing tool named Guenther to detect SSR misuses, including SSRF. However, Guenther requires manual input of the URL and parameters for fuzzing, which involves significant manual efforts to crawl and analyze. The recently introduced SSRFuzz [82] detects SSRF in PHP applications by combining dynamic taint analysis with black-box fuzzing. The authors first examined every function in the PHP manual to identify all potential sinks. Then, SSRFuzz crawled the application, dynamically tracking taints and logging HTTP requests and parameters when tainted input reached sinks. Finally, SSRFuzz applied black-box fuzzing with SSRF-specific mutations and monitoring rules to identify vulnerabilities. Compared to SSRFuzz, Artemis identifies both PHP built-in and third-party functions as sources and sinks using LLM and performs static taint analysis. The exploit is then constructed automatically using LLM from the results of static taint analysis, without the need for crawling and fuzzing.

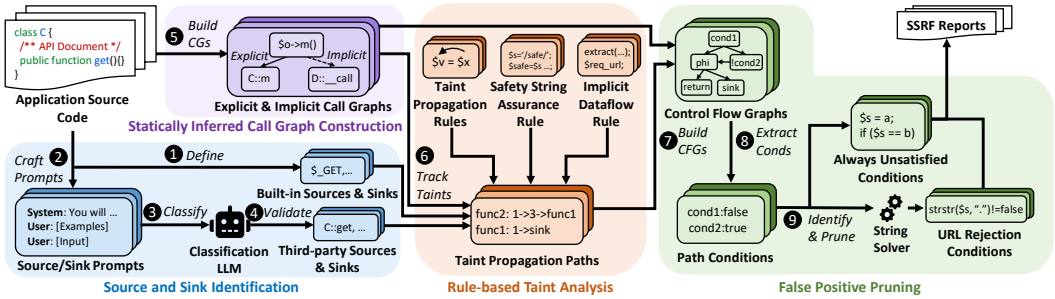


Fig. 3. The overall architecture of Artemis.

3 System Design

In this section, we present the design details of Artemis. Figure 3 shows the overall architecture of Artemis. First, Artemis takes the source code of the application as inputs and extracts both PHP built-in and third-party functions as candidate sources and sinks. Second, Artemis constructs both explicit and implicit call graphs statically from the source code. Third, Artemis performs a taint analysis based on constructed call graphs, augmented propagation rules, implicit data flow rules, and safety string assurance rules to locate tainted paths from candidate sources to sinks. Lastly, Artemis prunes false positives that are caused by path insensitivity.

3.1 Source and Sink Identification

Source and sink functions are crucial for SSRF detection, where source functions take the user inputs and sink functions send server-side requests. Artemis identifies them from both PHP standard libraries and third-party libraries.

Built-in Sources and Sinks. We consider 5 built-in superglobals [18] as sources, including `$_GET`, `$_POST`, `$_REQUEST`, `$_COOKIE`, and `$_SERVER`, because they commonly retrieve user inputs from incoming requests in PHP language. We follow the practice of previous efforts [82] to extract built-in SSRF sinks that can handle both network-based URLs (e.g., `http://` and `ftp://`) and file-accessing URLs (e.g., `file://` and `phar://`) to send server-side requests. By combining results from previous efforts and existing tools [6, 27, 82], 86 built-in sinks are collected. The 86 built-in sinks include 11 network request sending functions such as `curl_init()` from cURL and `fsockopen()` from the `socket` library and 75 remote file accessing functions such as `file_get_contents()`.

Third-party Sources and Sinks. Third-party library functions are commonly used as sources and sinks in modern PHP applications. These libraries simplify development by encapsulating built-in sources with additional layers of input encoding, validation, and sanitization [5, 8]. Similarly, they wrap built-in sinks with features like argument validation, response parsing, and error handling mechanisms such as timeouts and retries [3, 4]. We perform an offline analysis to identify source and sink functions from third-party libraries. This allows us to cache the results for quick retrieval during analysis, eliminating the need for repeated reanalysis.

We extract candidate source and sink functions from third-party libraries, with a focus on public functions with PHPDoc comments [17], as functions lacking PHPDocs are less likely intended for external use. Next, we prune those source candidates with return[2] types of `void`, `int`, `float`, or `bool`, as these types of user input cannot effectively manipulate URLs for sending server-side requests to user-specified but restricted destinations.

[2]The return types can be extracted from the `@return` tag in functions' corresponding PHPDocs.

Furthermore, we refine the source and sink candidates by utilizing few-shot learning with GPT-4o [68]. We develop a custom prompt template [3] that includes function names and PHPDocs, along with one positive and one negative example. We apply the template to each source and sink candidate with its corresponding function name and PHPDoc and send the prompt to GPT-4o to classify each candidate function as a source, sink, or neither.

To ensure accuracy, we manually verify the third-party source and sinks generated by GPT-4o and ensure that no false positives were made at this stage. GPT-4o reports 48 sources and 42 sinks. After manual verification, we confirm 42 sources and 40 sinks as our third-party source/sink set.

3.2 Statically Inferred Call Graph Construction

To statically construct call graphs in dynamically typed PHP applications, Artemis identifies both explicit call targets with literal string class and method names, as well as implicit call targets involving magic methods, variable class names, and variable method names. We design call target connection strategies to cover all types of PHP method invocations while incorporating acceptable estimations to address the challenges of statically inferring dynamic type features. In designing our strategies, we conduct a preliminary statistical analysis of 442,129 method call sites from 55 PHP applications with reported SSRF CVEs. We apply a slight overestimation in challenging cases with lower frequencies to ensure that these strategies do not generate substantial false positives but enhance the efficiency of our analysis. In Section 4.2.1, we demonstrate that ignoring implicit call targets, despite their infrequent appearance, results in non-negligible false negatives.

Method Invocations. PHP methods can be invoked in seven forms: ❶ `C::m`, ❷ `$c::$m`, ❸ `$v->m`, ❹ `$v->$m`, ❺ **`new`** `C()`, ❻ **`new`** `$c()`, and ❼ `call_user_func` and `call_user_func_array`. The first two forms apply to static methods, the third and fourth apply to instance methods, the fifth and sixth apply to constructor calls, which are equivalent to `$v->__construct()`, and the seventh is equivalent to one of the first four. `C` denotes a literal class name in static method calls. `$c` denotes a variable class name in static method calls. `$v` denotes the receiving variable in an instance method call. The class name of `$c` or `$v` is literal `O`, denoted by `$c:O` or `$v:O` when it can be inferred through type inference [6, 64]. If type inference cannot infer any concrete types, the class name is considered variable, represented by `$v:any`. `m` denotes a literal method name. `$m` denotes a variable method name.

Explicit Call Target Connection. Artemis identifies explicit call targets by matching their signatures in both PHP built-in and target application code bases. For function calls, Artemis considers their signatures as their literal names with the parameter count. For method calls, Artemis considers their signatures as their class name and method literal name with parameter count and types. For a static method call in the form of `C::m(arg)`, its signature is `C::m(1)`, where 1 is the parameter count. For a method call in the form of `$o->m(a1,a2)` where `$o = new O`, its signature is `O::m(2)`. PHP does not support method overloading, meaning that it is not possible to define two methods of the same name in the same class. As a result, explicit method calls can be resolved uniquely in one class. When `$o` is inferred to have multiple types due to dynamic typing [29] or reflection [28], each candidate class of `$o` is checked to determine explicit call targets. To address method inheritance [23] from a parent class to a subclass, Artemis conducts additional matching if the current method's signature does not match anything. It iteratively replaces the class name in the signature with its parent class name until a match is found or reaches the root class without identifying a valid method. The formal rules to connect explicit calls are denoted as:

$$\texttt{\$v->m(k)}, \texttt{\$v:O}, \exists \texttt{C::m(k)}, \texttt{C} \supseteq \texttt{O} \implies \texttt{C::m}$$

$$\texttt{O::m(k)}, \exists \texttt{C::m(k)}, \texttt{C} \supseteq \texttt{O} \implies \texttt{C::m}$$

Implicit Call Target Connection. When a method call target's method name and class name

[3]The template can be found in the full version of this paper [48].

```
1  ClientWrapper:: get ($_GET['uri']);
2
3  class ClientWrapper {
4   /** @var Client */
5   private static $client;
6
7   public static function __callStatic ( $m , $o){
8    $url = $o[0];
9    self::$client::{ $m }($url, []);
10  }
11 }
12
13 class Client {
14  public static function  get ($url, $args){}
15  ...
16  public static function request($m, $uri, $ops) {}
17 }
```

(a) CVE-2023-40969 with magic method call

```
1  class AJXP_Controller {
2    public static function find($action, $httpVars,
     ↪ $fileVars) {
3    $callback = getCallback($actionName);
4    $plugin = $callback->getPlugin();
5    $methodName = $callback->get("methodName");
6    return $plugin -> $methodName ($action, $httpVars,
     ↪ $fileVars);
7   }
8  }
9
10 class HttpDownloader extends AJXP_Plugin {
11  public function switchAction($action, $httpVars,
    ↪ $fileVars) {
12   ...
13  }
14 }
```

(b) CVE-2019-15033 with variable class name and variable method name.

Fig. 4. Examples of implicit call targets. →, ⇢, and ⇢ represent the explicit/implicit data flows and implicit call flows.

are known (e.g., `O::m(2)`) but its definition in the corresponding class hierarchy is missing, Artemis considers it as a magic call and searches for magic methods, including `__call()` and `__callStatic()`, within the inheritance chain. The formal rules to connect magic calls are denoted as:

$$\texttt{\$v->m(k)}, \texttt{\$v:O}, \nexists \texttt{C::m(k)}, \exists \texttt{C::\_\_call}, \texttt{C} \supseteq \texttt{O} \implies \texttt{C::\_\_call}$$

$$\texttt{O::m(k)}, \nexists \texttt{C::m(k)}, \exists \texttt{C::\_\_callStatic}, \texttt{C} \supseteq \texttt{O} \implies \texttt{C::\_\_callStatic}$$

The definition of a magic method in a class can be treated as the definition of any undefined method in the same class with a minor twist. Specifically, a magic method takes two parameters—the first is a string representing the invoked method name, and the second is an array containing the actual arguments passed to the invoked method. For example, as shown in Figure 4a, the signature of call at line #1 is `ClientWrapper::get(1)`, which does not match existing method signatures in `ClientWrapper`. However, a magic method `__callStatic()` exists, therefore Artemis builds an implicit call flow from line #1 to line #7.

When a call target has a literal method name but a variable class name, (e.g., `$v::m(k)`) Artemis first searches through all built-in and application classes to find the class methods whose signatures match the literal method name. If no match is found, but magic methods exist in classes, Artemis considers the magic methods (i.e., `__call()` or `__callStatic()`) in this class as candidate call sites, regardless of their parameters. The formal rules to connect callees with literal method names but variable class names are denoted as:

$$\texttt{\$v->m(k)}, \texttt{\$v:any}, \exists \texttt{C::m(k)}, \texttt{D::m(k)}, \ldots \implies \texttt{C::m}, \texttt{D::m}, \ldots$$

$$\texttt{\$c::m(k)}, \texttt{\$c:any}, \exists \texttt{C::m(k)}, \texttt{D::m(k)}, \ldots \implies \texttt{C::m}, \texttt{D::m}, \ldots$$

In the motivating example in Figure 2, the implicit call target is a class constructor (`__construct()`) call at line #10 that has a variable class name `$rss->file_class`. By searching all classes in the CommentLuv application, Artemis identifies that the constructor function of the `SimpleCluvPie_File` class is the only constructor method with the same number of parameters as the call site, thereby revealing the implicit call flow.

When a call target has a literal class name but a variable method name (e.g., `C::$m(k)`), Artemis searches all the methods in the corresponding class and classes in the class hierarchy. All methods that have the same number of parameters as the call site (e.g., `C:f(k)`, `B::g(k)` where B is parent class of C) are considered as candidate call targets. Moreover, all magic methods in the target classes are

considered candidate call sites without parameter checking. The formal rules to connect callees with literal class names but variable method names are denoted as:

$$\texttt{\$v->\$m(k)}, \texttt{\$v:O}, \exists \texttt{C::m(k)}, \texttt{C::n(k)}, ..., \texttt{C} \supseteq \texttt{O} \implies \texttt{C::m}, \texttt{C::n}, ...$$

$$\texttt{O::\$m(k)}, \exists \texttt{C::m(k)}, \texttt{C::n(k)}, ..., \texttt{C} \supseteq \texttt{O} \implies \texttt{C::m}, \texttt{C::n}, ...$$

For example, in Figure 4a, the call target at line #14 has a literal class name `Client` and a variable method name `$method`. ARTEMIS can not only identify the call site as `Client::get` from the aforementioned magic call flow construction but also search for all methods in the `Client` class and pinpoint `get` as the call site, as it has the same number of arguments as the call target at line #16. In our experiments, 132 (0.03%) of the call sites are classified under this category. Thus, even though our method overestimates the *parameter count*, it does not result in a notable increase in false positives in detection while enhancing coverage.

When a call target has a variable class name and a variable method name, (e.g., `$c::$m`), ARTEMIS searches all methods, including magic call methods, in all built-in and application classes to find those with the same *number of parameters* and *compatible types* as the call target, with magic methods not requiring parameter checks. Types of formal parameters t_f and actual parameters t_a match if a type in t_a can be cast to a type in t_f. If a type cannot be inferred, the type `mixed` is used, which is compatible with all types. The formal rules to connect callees with variable class names and variable method names are denoted as:

$$\texttt{\$v->\$m(a1:}t_{a_1}\texttt{,...an:}t_{a_n}\texttt{)}, \texttt{\$v:any}, \exists \texttt{C::m(A1:}t_{f_1}\texttt{,...,An::}t_{f_n}\texttt{)}, t_{a_1} \subseteq t_{f_1}, \ldots, t_{a_n} \subseteq t_{f_n} \implies \texttt{C::m}$$

$$\texttt{\$c::\$m(a1:}t_{a_1}\texttt{,...an:}t_{a_n}\texttt{)}, \texttt{\$c:any}, \exists \texttt{C::m(A1:}t_{f_1}\texttt{,...,An::}t_{f_n}\texttt{)}, t_{a_1} \subseteq t_{f_1}, \ldots, t_{a_n} \subseteq t_{f_n} \implies \texttt{C::m}$$

For instance, in the example shown in Figure 4b, the call at line #7 involves both an unknown class name and an unknown method name. Since the method at line #12 takes three parameters, and the parameter types match[4], we consider it a match and treat it as a potential call target. In our experiments, 88 call sites (0.02%) fall into this category. Therefore, our conservative call resolution, which over-approximates potential call targets by matching parameter count and type, not only ensures completeness but also keeps false positives at a manageable level.

Variadic methods [31], which accept varying numbers of arguments, are not included to simplify our design. In our experiment, each application averages only two such methods, and manual validation showed that they are not used in SSRFs.

3.3 Rule-Based Taint Analysis

ARTEMIS performs rule-based taint analysis to track the propagation of tainted data within the constructed call graphs. This analysis traces how attacker-controlled input moves through various call sites and control flows until it reaches sink functions, leading the server-side requests to unintended destinations. We design a set of taint propagation rules to handle generic operations in PHP syntax. To ensure high detection accuracy, our design focuses on mitigating the issues of over-tainting and under-tainting prevalent in existing tools. Furthermore, we design taint clearance rules to terminate the tracking of tainted data when it is no longer relevant or has been neutralized.

Language Abstraction. We perform an abstraction[5] of the PHP language syntax geared toward taint analysis [6], specifically investigating the propagation of tainted sources to sink operations, which may lead to requests being directed to unintended destinations via various control and data flows. In this abstraction, a PHP program is conceptualized as a series of statements in the static single assignment (SSA) forms illustrated in Figure 5.

Variable States. For a variable with scalar types, such as `string`, ARTEMIS considers such variable v as tainted, i.e., $v : \tau$, when v directly or indirectly derives its value from a tainted

[4]The types are inferred by type inference where `$action` is a string while `$httpVars` and `$httpVars` are arrays.
[5]The full language abstraction description can be found in the full version of this paper [48].
[6]The syntax abstraction along with the taint propagation is performed on PHP v7.4.

source, inheriting the tainted status. Otherwise, the variable v is safe, i.e., $v : \mu$, because it is not influenced by any tainted source in any way. For complex variable types, such as `array`, Artemis considers an array variable a as tainted, if and only if all elements inside of a are tainted, denoted by $a : \tau \equiv \forall v \in a, v : \tau$. The variable a is safe if and only if all the elements inside a are safe, denoted by $a : \mu \equiv \forall v \in a, v : \mu$. Otherwise, a is partially tainted, i.e., $a : \widetilde{\tau}$.

To represent taint states, Artemis uses a dedicated data structure T that comprises a triplet of components: $\texttt{self}, \texttt{arr}_s, \texttt{arr}_r$. `self` represents the tainted or safe state of the variable itself. $\texttt{arr}_s$ and $\texttt{arr}_r$ are exclusively applicable for array variables. $\texttt{arr}_s$ tracks the states of the array elements with statically known keys. $\texttt{arr}_r$ tracks the states of the array elements whose keys can only be inferred during runtime execution.

To prevent over-tainting, we use two separate taint structures for each variable. T_f is for local file URLs and T_r is for request URLs. Unless stated otherwise, the rules apply to both taint structures.

$v := v_1$	/*variable assignment*/
$v := \ominus v_1$	/*unary op assignment*/
$v := v_1 \oplus v_2$	/*binary op assignment*/
$v := (T) v_1$	/*type cast*/
$a := [(k_1 \Rightarrow)^* v_1, \cdots]$	/*array initialization*/
$a[k^*] := v_1$	/*array element assignment*/
$v := a[k]$	/*assign. from array element*/
$\texttt{foreach}(a \texttt{ as } (k \Rightarrow)^* v)\{s\}$	/*foreach loop */
$v := \phi(v_t, v_f)$	/*branch value merging*/
$\texttt{call}\,(m, v_a \rightarrow v_f)$	/*v$_a$/v$_f$ actual/formal arg.*/
$v := \texttt{return}\,(m, v_r)$	/*method m returns var. v_r*/
$v := \texttt{yield}\,(m, v_r)$	/*m returns v_r in a generator*/
$v := \texttt{new}\ T\,(v_a \rightarrow v_f)$	/*v$_a$/v$_f$ constructor call*/

Fig. 5. Abstracted PHP language statement syntax in SSA form. * represents an optional element.

Taint Propagation: Generalized Rules. When a statement is evaluated, the states of the corresponding variables are updated. Table 1 (rules ❶-❼) shows all the generalized taint propagation rules in our language abstraction used by Artemis. Among all rules, those for variable assignment, type casting, and unary operations are consistent with those used in state-of-the-art approaches [6, 27, 33, 39, 64] and require no modification.

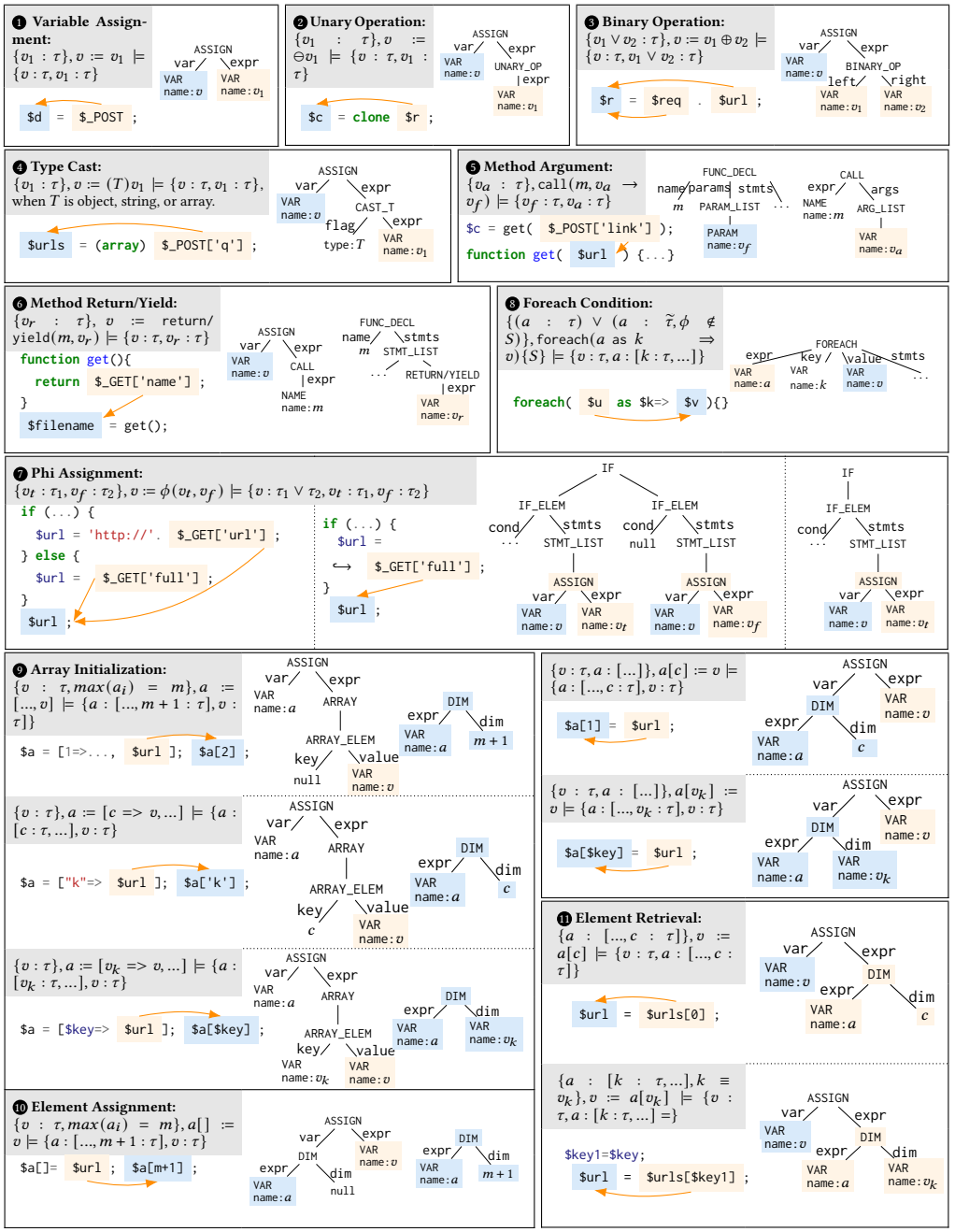
Upon encountering branching, as seen in `if-else` structures, Artemis employs the phi assignment rule to merge the states originating from different branches. For composite conditions, e.g., `if(cond1 && cond2)` and `switch-case` blocks, Artemis flattens them into a series of nested conditional statements with atomic conditions, e.g., `if(cond1){if(cond2)}`.

During method calls, states transition from actual arguments to formal arguments, navigate through the statements within the method body, and eventually extend outward from the method call to the returned or yielded variable at the caller's site. In the case of API calls, such as `$u = trim($_GET['q']`, where the implementation (e.g., `trim()`) is inaccessible to Artemis, the propagation flow simplifies—Artemis copies and merges the states from actual arguments, e.g., `$_GET['q']`, to the receiving variable, e.g., `$u`.

Taint Propagation: Array-Specific Rules. We design propagation rules (❽-⓫ in Table 1) for array-related operations specifically because prior approaches either tend to over-taint a whole array with one tainted element or overlook taint propagation on the arrays in the `foreach` loop.

During array initialization, Artemis stores the state of all array elements in either $\texttt{arr}_s$ or $\texttt{arr}_r$ depending on the corresponding keys. When the key has a statically known value c, the state of the element is stored in $\texttt{arr}_s$ under the key c. When the value of the key can only be inferred at runtime, the element's state is stored in $\texttt{arr}_r$ under a symbolic key denoted by v_k. In cases where no explicit key is specified, we interpret the key as an integer that increments by one from the largest previously used numeric key. If all previous keys are statically known, i.e., $\texttt{arr}_r$ is empty, a concrete integer k_c is computed based on existing numeric keys in $\texttt{arr}_s$, denoted by $k_c = 1 + \max_{k \in \texttt{arr}_s} k$. The state of the corresponding element is stored in $\texttt{arr}_s$ under k_c. Otherwise, a new symbolic value

Table 1. Generalized and array-specific taint propagation rules in both SSA and AST forms for different PHP syntax with source code examples. The tainted state is propagated from the *src* variable(s) to the *tgt* variable(s).



```
1  $config = array(...
2    'updateUrl'  => 'http://...',
3    );
4  ...
5  $config[ 'download' ] =  $_GET['download'] ;
6  ...
7  file_get_contents($config['updateUrl'], ...);
```

(a) Array with both tainted and safe elements.

```
1  $urls  =  $_POST [...];
2  ...
3  foreach ( $urls as $imgUrl ) {
4    ...
5    readfile( $imgUrl , ...);
6  }
```

(b) CVE-2022-40357 with tainted `foreach` loop.

Fig. 6. Examples of array-specific taint propagation rules. → represents the taint data flow from `source` to `target`.

$\max(a_i) + 1$ is created as the key and the state of the corresponding element is stored in $\texttt{arr}_r$. This aligns with the definition of arrays in PHP. In all cases, `self` is maintained to track whether all array elements have the same tainted state.

During array element assignment, Artemis propagates the state from the assigning variable to the corresponding array element. If the key is a statically known value c, the state of the element is stored in $\texttt{arr}_s$ under c. If the value of the key can only be inferred at runtime, a symbolic key v_k using the name of the key variable is used to store the state of the corresponding element in $\texttt{arr}_r$. In cases where no explicit key is specified and the exact number of elements in the array is challenging to determine due to loops, a symbolic key $\max(a)$ is employed to store the state of the corresponding element in $\texttt{arr}_r$. `self` is also updated to track whether all array elements have the same tainted state. This design is based on the observation that when this syntax is used, developers care less about the exact value of the key, rather, they just want to append an element to the array. Additionally, when used inside loops, all elements have the same taint. Therefore, this approximation models the propagation of this syntax well.

During array element retrieval (with a given key k), Artemis propagates the state from the corresponding array element to the receiving variable. Artemis first checks the array's `self` field. If `self` is either τ or μ, indicating that all elements share the same state, the receiving variable adopts this state. Otherwise, this array is partially tainted, we need to check the $\texttt{arr}_s$ and $\texttt{arr}_r$ fields. If k is statically known, we search for k in $\texttt{arr}_s$, and the state of the receiving variable is updated to match the state of the element retrieved from $\texttt{arr}_s$. If k can only be inferred at runtime, we search in $\texttt{arr}_r$ for an exact match with k and all of k's aliases, where an exact match means k has the same variable name and type with the array key. The state is copied to the receiving variable only when a match is found. For example, in Figure 6a, at line #1-3, the `$config` array is initialized with the key `updateUrl` and a constant string. Therefore, the value is untainted, i.e, $\texttt{self} = \mu, \texttt{arr}_s = [\texttt{updateUrl} = \mu]$. At line #5, the array element with key `download` is assigned a tainted value, leading to `$config` being partially tainted, i.e, $\texttt{self} = \widetilde{\tau}, \texttt{arr}_s = [\texttt{updateUrl} = \mu, \texttt{download} = \tau]$. At line #7, since `self` is $\widetilde{\tau}$ and `updateUrl` exists in $\texttt{arr}_s$, the result taint is correctly retrieved as μ, i.e., not tainted. Existing tools [39] mark the whole `$config` array as tainted after line #5, therefore, the access to the element at line #7 is also marked as tainted, leading to a false positive.

Upon encountering `foreach` loops, when all elements in the iterated array a are tainted, i.e., $a : \tau$, or when the array is partially tainted (i.e., $a : \widetilde{\tau}$) but there are no branching paths in the loop body, the created value variable will be considered tainted. Although this is a conservative approximation, it does not lead to false negatives in practice because in cases where a vulnerability path involves a `foreach` loop, the array being iterated is typically derived from a source variable or initialized within a loop, resulting in all elements having the same taint. For example, in Figure 6b, at line #1, `$urls` is retrieved from `$_POST`, which is a built-in source, i.e., $\texttt{self}_{\texttt{POST}} = \tau$. Therefore, `$urls` is

considered tainted, i.e., $\texttt{self}_{\texttt{urls}} = \tau$. Inside the `foreach` loop at line #3, `$imgUrl` is created by `$urls`, thus it is also tainted since $\texttt{self}_{\texttt{urls}} = \tau$. We model the `foreach` loop to avoid under-tainting, while existing tools [33, 64] fail to maintain taint propagation for variables created inside `foreach`, resulting in false negatives.

Taint Continuity: Implicit Dataflow Reconstruction Rule. In addition to applying the aforementioned taint propagation rules, we must also trace taint propagation in cases where data flows are implicitly present due to the dynamic features of PHP. We manually go through the PHP documentation and discover that the built-in `extract` [12] function creates variables implicitly by extracting them from an array. Specifically, a variable is created implicitly for every key/value pair in the first array parameter. The created variable names are derived from the array keys with a prefix specified in the third parameter. When an `extract` function is encountered, both the array and the prefix are recorded. If a variable is subsequently accessed without preceding data flow and the prefix matches, a synthetic assignment from the array is generated to explicitly establish the data flow. For example, between two consecutive statements `extract($_GET,...,'req');` and **`return`** `$req_url;`, Artemis interprets an intermediate hidden statement `$req_url=$_GET['url'];` to uncover the implicit data flow generated by the `extract` function.

Taint Clearance: Safety String Assurance Rule. To exploit an SSRF vulnerability, the tainted string must either be an arbitrary file URL fully controlled by an attacker or a request-sending URL where the attacker has full control over the IP address to which the request is sent. However, when tainted strings are concatenated with other strings to form a new string, the resulting string does not necessarily become a file URL or request-sending URL. Simply tainting the result if any of its components are tainted leads to over-tainting. Artemis applies safety string assurance rules to prevent such over-tainting cases where SSRF exploitation is no longer possible. In Section 4.2.2, we demonstrate that these safety string assurance rules are crucial in reducing over-tainting and false positives.

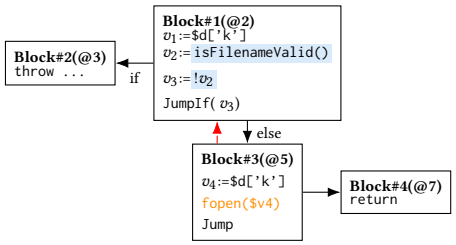
We take string concatenation in the form of `$v=$v1.$v2` without loss of generality to illustrate our rules. Other string manipulations, such as interpolation and formatting, can be converted to equivalent string concatenation forms when handling safety strings. For example, string interpolation such as `$v="http://`**`$domain`**`"` is equivalent to `$v="http://".$domain`. After the concatenation step, for T_f (taint for file URLs), we consider a variable v to be untainted, represented as $T_{f_v} = (\mu, [], [])$, if $v1$ and $v2$ are not both tainted. This is because if any part of the string is not attacker-controlled, the concatenated file path cannot be arbitrary. For T_r (taint for request-sending URLs), the taint status of $v1$ is first checked. If $v1$ is tainted as part of a request URL (i.e., $T_{f_{v1}}$ is set), then v is also considered tainted. If $v1$ is not tainted, its value is evaluated. If $v1$ is a constant or literal string, we compare the value of $v1$ against valid URL schemes and also use a URL parser to parse $v1$. If $v1$ represents a valid scheme (e.g., `http` or `tcp`) or appending $v2$ could form a new host (e.g., `$v = "http://a" . $v2` and `$v2=".evil.com"`), the taint depends only on $v2$, because now $v2$ determines the IP that the request is sent to. Otherwise, T_r remains untainted. This approach ensures that the tainted input affects the domain segment of the URL, controlling the IP address to which the request is sent as defined by the standard [35]. In other cases, the result follows conventional binary operation rules, where the result is tainted if any part of the string is tainted.

3.4 False Positive Pruning

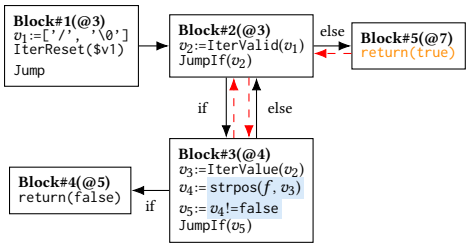
The rule-based taint analysis described in Section 3.3 is path insensitive, which can lead to false positives by reporting infeasible SSRF propagation paths. This occurs when the code conditions always cannot be satisfied or reject URLs. For instance, in Figure 7, we identify the tainted path from the source `downloadFile` at line #8 to the sink `fopen` at line #5. However, there is a conditional check at line #2 along the path. When the attacker manipulates `$d['filename']` to be a URL, the

```php
1  private function downloadFile($d) {
2    if (!$this->isFilenameValid($d['filename']))
3      throw ...;
4    else {
5      fopen($d['filename'], ...); //sink
6    }
7  }
8  $this->downloadFile($_REQUEST);
```

```php
1  public function isFilenameValid($f) {
2    // ...
3    foreach (['/', '\0'] as $char)
4      if (strpos($f, $char) !== false)
5        return false;
6    // ...
7    return true; //Expected return stmt
8  }
```



(a) CFG of `downloadFile`



(b) CFG of `isFilenameValid`

Fig. 7. An example of false positive caused by path condition, with the source code block and the corresponding CFG in SSA form. - -> represents the (backward) control dependency edges. ⟶ represents the block linkage in a CFG.

user-defined function `isFilenameValid` always returns `false` because the URL contains a forward slash (/). It means that SSRF exploitation is impossible along the path, which should be flagged as a false positive. We refer to the conditional checks on the taint propagation paths as path conditions.

To prune infeasible propagation paths, we perform a lightweight static analysis with a focus on SSRF-specific string conditions. We start by extracting a consolidated list of conditions for each tainted path. Next, we prune paths that contain conditions that are always unsatisfied or reject URLs before they reach the sensitive sink. By modeling and checking only conditions that are relevant in SSRF, we prune path-related false positives without expensive symbolic execution. Section 4.2.2 shows that analyzing path conditions is important in reducing false positives.

Path Condition Extraction. Given a taint propagation path from source to sink (Section 3.3), Artemis identifies all function invocations along the path, extracts path conditions from each function's control flow graph (CFG), and merges these conditions into a consolidated path condition list. For each function in a taint path, Artemis identifies the sink block containing the sink function invocation (leaf node in the call graph) or the end block containing a return statement (intermediate node in the call graph). Starting from the identified sink or end block, we conduct a backward dependency analysis by iteratively traversing preceding blocks until we reach either the entry block of the CFG or a previously visited block, which indicates the presence of a loop. Artemis extracts condition labels (i.e., true/false) derived from dependence edges. Artemis connects the traversal flow using the constructed call graphs (Section 3.2) whenever it encounters function calls. It merges the extracted conditions from each function in the call graph by applying the corresponding logical operators (AND/OR) according to the control flow dependencies.

We use the example in Figure 7 to illustrate how Artemis extracts interprocedural path conditions within a tainted path. Given a taint propagation path where the tainted source `$_REQUEST` is passed as an argument to the `downloadFile()` function at line #8 and subsequently flows to the sink function `fopen()` at line #5, Artemis identifies Block #3 within the `downloadFile()` function as the tainted block where the sink function is invoked. Starting from Block #3, Artemis tracks back to Block #1 with an edge labeled as `false`, indicating that: 1) a path condition is defined in Block #1, 2) branching occurs at Block #1, and 3) the path from Block #1 to Block #3 is taken when the condition is false. Artemis

Table 2. String checking functions and core translation rules. s represents the string from user input, c and p represents constants used in condition.

Function	Type	Translation
in_array(s,a)	Allowlist	/
array_key_exists(s,a)	Allowlist	/
strpos(s,c)	URL-blocking	(str.indexof s c 0)
stripos(s,c)	URL-blocking	(str.indexof (str.to_lower s) (str.to_lower c) 0)
strstr(s,c)	URL-blocking	(str.contains s c)
stristr(s,c)	URL-blocking	(str.contains (str.to_lower s) (str.to_lower c))
preg_match(p,s)	URL-blocking	(str.in_re s (re.from_ecma2020 p))
preg_match_all(p,s)	URL-blocking	(str.in_re s (re.from_ecma2020 p))

then retrieves the source code line number (i.e., 2) from Block #1 to extract the condition's source code as `!$this->isFilenameValid($d['filename'])`. Artemis considers the path label to normalize the path condition from Block #1 to Block #3 as `isFilenameValid()` returning true. Artemis then locates the definition of `isFilenameValid()` function from the constructed call graphs and identifies Block #5 as the target block, which is the only block that returns true. Starting from Block #5, Artemis tracks back to Block #2 with an edge labeled as `false`. Artemis then retrieves the source code line number (i.e., 3) from Block #2 to extract the condition's source code as `foreach (['/', '\0'] as $char)`, indicating that: 1) the path from Block #2 to Block #5 marks the end of a loop, and 2) any condition within the loop that would cause early termination must be negated; otherwise, the path from Block #2 to Block #5 would be infeasible. From Block #2, Artemis backtracks every path until it reaches Block #2 again and identifies that branching and condition definition occur at Block #3. Artemis then retrieves the source code line number (i.e., 4) from Block #3 to extract the condition's source code as `strpos($f, $char) !== false`. With negation, Artemis normalizes the path condition from Block #2 to Block #5 in `isFilenameValid()` function as `strpos($f, $char) == false`. It is also the path condition from the source `$_REQUEST` to the sink `fopen()` in the `downloadFile()` function.

Always Unsatisfied Conditions. Artemis prunes unreachable paths after identifying that their conditions are always unsatisfied, including 1) parameter and argument mismatch; and 2) conflicting logic. When a condition depends on a function parameter but the passed argument at the call site does not match, it results in an unsatisfiable condition. We denote the condition on formal argument involving constants as $C(v_f, S_1)$ and the called method is m with constant parameter S_2, then if:

$$C(v_f, S_1) \wedge \text{call}(m, S_2 \rightarrow v_f) \equiv \bot$$

, the path is pruned. For example, if a function checks whether a parameter equals a specific value (e.g., `if($param === "A")`), but the passed argument is `"B"`, the condition is always false. Conflicting logic arises when contradictory conditions exist, such as in the cases of `if ($x == 10)` and `if ($x != 10)`, which are mutually exclusive. Formally, for conditions $C_1(v)$ and $C_2(v)$ that checks variable v with statically known conditions, if:

$$C_1(v) \wedge C_2(v) \equiv \bot$$

then the path is pruned.

URL Rejection Conditions. Artemis prunes secure paths where attacker-controlled URLs cannot reach the sink due to path conditions that perform string checks. Specifically, two types of path conditions need to be identified: 1) allowlist checks, where tainted user input is restricted to a predefined list of values; and 2) string checks that block URLs from passing because URL-required characters are disallowed, such as / or .. To accurately identify the two types of conditions, first,

Artemis identifies potential conditions by locating related functions shown in Table 2. Allowlist checks are recognized directly by their function names while URL-blocking string checks are modeled based on their logic. Conditions involving 1) external variables (e.g., class properties or globals) or 2) non-constant variables not directly derived from user input (e.g., from function calls) are excluded. Then, each string check is translated into a string constraint, with the core rules for this process outlined in Table 2. Lastly, Artemis uses the OSTRICH SMT solver [41] to verify whether valid URLs in the form of `protocol://domain.tld` can satisfy the constraints. For example, in Figure 7, the condition on line #4 in `isFilenameValid` uses `strpos`, making it a candidate for URL rejection. The conditions are modeled as string constraints `(assert (= (str.indexof w char 0) (- 1)))` where `char` can be `/` and `\0`. Constraint for valid URL `w` is modeled by regular expression. Then, we use the SMT solver to determine that the union of the constraints is unsatisfiable for a valid URL. Therefore, we mark the path as a false positive.

4 Evaluation

In this section, we present our experimental evaluation. We first evaluate the detection capability of Artemis against five generic static PHP vulnerability detection tools based on taint analysis. Next, we evaluate the generalizability of the source and sink identification module, the performance of the call graph construction module, and finally, the detection speed and scalability of Artemis. We have implemented a prototype of Artemis. The source/sink identification module is developed using the GPT-4o-2024-08-06 model [68] with default configurations, accessed via API. The maximum output token length is set to 1024. The call graph construction and rule-based taint analysis module are built on top of Phan [6, 7], which utilizes the php-ast extension [15] to extract abstract syntax trees (ASTs). The false positive pruning module is built on top of the Joern framework [13]. All experiments are run on a system with an Intel i7-10700 CPU with 8 cores, 32GB RAM, and running 64-bit Ubuntu 22.04 with kernel version 5.15.0. The applications are set up using the PHP version recommended in their respective installation documentation.

4.1 Methodology

4.1.1 Target Applications and Vulnerability Collection. We select target applications based on two criteria. First, we collect vulnerable applications from the Common Vulnerabilities and Exposures (CVE) database [10] to evaluate whether Artemis is able to detect known SSRFs. Using keywords such as "server-side request forgery" and "SSR", we filter applications that meet the following conditions: 1) the application is written in PHP language and open-source, and 2) the CVE reports a true SSRF vulnerability in the application code. From this list, we prioritize frequently occurring applications and download both the vulnerable and latest versions for evaluation. We collect 55 applications with reported SSRFs. Second, we gather popular open-source PHP applications from GitHub (via the Awesome-Selfhosted project [9]) and the WordPress plugin repository [16] to assess Artemis's ability to discover new SSRFs. We sample a total of 195 applications. We use the latest available versions at the time of the experiment. In total, we collected 250 PHP applications using the two criteria. The applications have varying complexity, with line of code (LoC) ranging from 780 to 872506, with an average of 173049.

The collected SSRF CVEs cover a variety of root causes, including 1) missing URL validation, where user-provided URLs are used without validation; 2) missing URL segment validation, where user input modifies parts of the URL's domain without validation; 3) incomplete input validation, where basic checks are bypassed with attacker-controlled domains; and 4) flawed input validation, where allowlists are used but flawed, leading to bypasses. SSRFs usually have severe impacts on running applications. The collected CVEs cause impacts including 1) access control bypass; 2) sensitive data leakage; 3) denial of service (DoS); 4) privilege escalation; and 5) arbitrary remote

Table 3. Feature comparison of Rips, PHPJoern, TChecker, **Psalm**, Phan, and Artemis.

Feature	Rips	PHPJoern	TChecker	Psalm	Phan	Artemis
Sources/Sinks	PHP built-in	PHP built-in	PHP built-in	PHP built-in	PHP built-in	PHP built-in and third-party
Explicit Calls	Unique function name matching	Unique function/method name matching	Type inference	Type inference	Type inference	Type inference
Implicit Calls	✗	✗	✗	✗	✗	Relaxed implicit call graph construction
Taint Propagation	Generalized rules, sanitizer functions	Generalized rules, sanitizer functions	Generalized rules, sanitizer functions	Generalized rules, sanitizer functions	Generalized rules, sanitizer functions	Generalized rules, refined array rules, implicit dataflow rules, and safety string analysis
False Positive Pruning	✗	✗	✗	✗	✗	Path condition analysis

code execution. For example, the SSRF in Figure 6b results in data leakage as the response of the crafted request is returned to the attacker. In contrast, the SSRF in Figure 2 allows attackers to circumvent access controls to communicate with internal network hosts.

4.1.2 Alternative Approaches. We compare Artemis with five generic static PHP vulnerability detection tools based on taint analysis, i.e., **Rips** [39], **TChecker** [64], **PHPJoern** [33], **Psalm** [27] and **Phan** [7]. We summarize their analysis features and compare them with Artemis in Table 3. To tune the five generic tools for SSRF detection, we configure them with the same PHP built-in sources and sinks as Artemis to ensure a fair comparison. For call graph construction, Rips matches only function names, while PHPJoern matches unique function and method names. TChecker, Psalm, and Phan employ type inference for explicit calls, but none of the five tools handle implicit calls. TChecker starts call graph construction and taint analysis from the top-level function of each PHP file, while the other tools start their analysis from each defined function. In taint propagation, the five tools use generalized taint rules (rules ❶-❼ in Table 1) same as Artemis with manually defined sanitizers for non-SSRF vulnerabilities, such as SQL injection and XSS. Rips ignores object-oriented features, therefore if the right-hand side includes objects, the taint is not propagated for rules ❶-❹ in Table 1. None of the five tools prunes false positives with path condition analysis as Artemis does.

To assess the impact of the third-party sources and sinks, we extend all alternative approaches by incorporating both built-in and third-party sources and sinks identified by Artemis, resulting in five modified tools: **Rips***, **TChecker***, **PHPJoern***, **Psalm***, and **Phan***.

To assess the impact of the false positive pruning module in Artemis, we integrate it into TChecker, Psalm and Phan which have type inference support. We refer to these modified versions as **TChecker**†, **Psalm**†, and **Phan**†.

4.1.3 Ablated Versions. To evaluate the contribution of each component in Artemis, we conduct an ablation study. We systematically remove one module from taint analysis at a time to assess the module's impact on detection performance. We compare Artemis against four ablated versions.

Artemis^a. We remove the *third-party source/sink identification* module from Artemis. Specifically, we only use the PHP built-in functions as the source and sink functions.

Artemis^c. We remove the *statically inferred call graph construction* module from Artemis. Specifically, we only use the explicit call graph construction in Phan.

Artemis^t. We remove the *rule-based taint analysis* module from Artemis. Specifically, we use the taint propagation rules in Phan.

Artemis^p. We remove the *false positive pruning* module from Artemis. The results from the *rule-based taint analysis* module are directly reported as candidate SSRFs.

4.1.4 Exploit Generation. Creating exploits manually from taint analysis reports is a time-consuming process requiring domain expertise. We automate the exploit generation process by leveraging LLM with a multi-turn conversation to generate exploits. The conversation prompt template can be found in the full version of this paper [48].

Due to context limitations in LLMs [22], we equip them with tools for dynamic context retrieval like code search, extraction, execution, and database queries [26]. When additional context is required, the LLM generates tool calls that convert to PHP function calls that gather information and return it to the LLM. In our implementation, we choose GPT-4o as the backbone LLM because 1) it has a relatively large context window; and 2) it has the best built-in support for tool calling.

The exploit generation is complex, therefore we break it down into three subtasks:

Payload Generation. The LLM identifies the user input type (e.g., GET/POST data, cookies) and generates the payload needed to trigger SSRF. For example, it may generate a POST request with the key `url` containing `blogspot` in value for our motivating example in Figure 2.

Route Formation. The LLM determines the URL route to reach the vulnerable code by analyzing routing code or framework knowledge. For instance, in Figure 8, the LLM identifies the correct route to the vulnerable function via domain knowledge about the CakePHP [2] framework it uses.

Value Inference. The LLM infers additional required field values, such as valid user IDs, which are necessary for the request to be accepted. For example, in Figure 8, the LLM needs to fetch a valid product associated with a manufacturer for the request to be accepted.

To enhance accuracy, we automate exploit testing on a live server, logging responses and execution traces. If an exploit fails, the feedback is used to refine it iteratively with the LLM. Unresolved cases after three attempts are marked as undetermined and reviewed manually. During the manual review, we revise the payload, route, and fields generated by the LLM. We then test the corrected exploit on a server to confirm its validity as a true or false positive.

4.1.5 Result Validation. If a path is automatically exploited by the LLM, it is marked as a true positive (TP). Otherwise, we manually verify if it is a TP or a false positive (FP) and write exploitation payloads for TPs. For reports from alternative methods, we check for overlap with TP reports from Artemis. Overlapping reports are marked as TPs, while non-overlapping ones undergo manual code review to identify false positives.

Note that multiple true positive paths can share the same patch, so they are reported as a single vulnerability, a common practice in CVE reporting. For example, in CVE-2018-1000138, the function `getFromWeb` is used to fetch web resources with three different sources, creating three true positive paths. However, as the patch is applied in the `getFromWeb` function, the CVE report combines all three paths under a single CVE.

4.2 SSRF Detection Results

4.2.1 True Positives. Table 4 and Table 5 present the detection accuracy results for Artemis, Rips, PHPJoern, TChecker, Psalm, Phan, and ablated versions of Artemis. Overall, Artemis significantly outperforms other static analysis tools, detecting up to **5** times more SSRFs. In total, Artemis identifies **207** true positive paths, corresponding to **106** true positive SSRFs. Among the 106 SSRFs, **35** are newly discovered by Artemis. Artemis generates exploits for 194 of 222 detected paths, covering 99 of 106 SSRFs. Among the 28 remaining paths, 15 are confirmed as false positives

Table 4. The comparision of detected number of TP paths by Artemis, Rips, PHPJoern, TChecker, Psalm, and Phan on 106 SSRF vulnerabilities (71 known and 35 new). ✗ means no TP path is detected for the corresponding vulnerability.

#	CVE/Vuln ID	Artemis	Rips	PHPJoern	TChecker	Psalm	Phan
1	CVE-2015-7816	1	1	✗	✗	1	1
2	CVE-2016-10926	1	1	1	1	1	1
3	CVE-2016-10927	1	1	1	1	1	1
4	CVE-2016-7964	1	✗	✗	✗	1	1
5	CVE-2016-9417	2	✗	✗	✗	2	2
6	CVE-2017-1000419	1	✗	✗	✗	✗	✗
7	CVE-2017-10973	1	1	✗	1	1	1
8	CVE-2017-14323	1	1	1	1	1	1
9	CVE-2017-16870	1	✗	1	✗	1	1
10	CVE-2017-7566	1	✗	✗	✗	✗	✗
11	CVE-2017-9307	3	3	3	3	3	3
12	CVE-2018-1000138	3	3	3	3	3	3
13	CVE-2018-11031	1	✗	✗	✗	✗	✗
14	CVE-2018-14514	6	✗	✗	✗	✗	✗
15	CVE-2018-14728	2	2	2	2	2	2
16	CVE-2018-15495	2	2	2	2	2	2
17	CVE-2018-16444	1	✗	✗	1	1	1
18	CVE-2018-18867	2	2	2	2	2	2
19	CVE-2018-6029	3	✗	✗	✗	✗	✗
20	CVE-2018-9302	4	4	4	4	4	4
21	CVE-2019-11565	3	✗	✗	✗	✗	✗
22	CVE-2019-11574	4	✗	✗	✗	4	4
23	CVE-2019-11767	1	✗	✗	✗	✗	✗
24	CVE-2019-12161	7	✗	✗	✗	✗	✗
25	CVE-2019-15033	1	✗	✗	✗	✗	✗
26	CVE-2019-15494	1	✗	✗	✗	✗	✗
27	CVE-2020-10212	3	3	3	3	3	3
28	CVE-2020-10791	1	✗	✗	✗	✗	✗
29	CVE-2020-14044	1	✗	1	✗	1	1
30	CVE-2020-20341	2	✗	✗	✗	✗	✗
31	CVE-2020-20582	4	✗	✗	✗	4	4
32	CVE-2020-21788	2	✗	✗	✗	✗	✗
33	CVE-2020-23534	1	✗	✗	✗	✗	✗
34	CVE-2020-24063	1	✗	1	1	1	1
35	CVE-2020-25466	2	2	✗	✗	✗	✗
36	CVE-2020-28043	1	✗	✗	✗	✗	✗
37	CVE-2020-28976	1	1	1	1	1	1
38	CVE-2020-28977	1	✗	1	1	1	1
39	CVE-2020-28978	1	✗	1	1	1	1
40	CVE-2020-35313	3	2	2	✗	2	2
41	CVE-2020-35970	1	✗	✗	✗	1	1
42	CVE-2021-24150	2	✗	✗	✗	✗	✗
43	CVE-2021-24371	2	✗	✗	✗	✗	✗
44	CVE-2021-27329	2	✗	✗	✗	✗	✗
45	CVE-2021-28060	1	✗	✗	✗	1	1
46	CVE-2021-4075	3	✗	✗	✗	✗	3
47	CVE-2022-0768	3	✗	✗	✗	✗	✗
48	CVE-2022-1037	1	✗	✗	✗	✗	✗
49	CVE-2022-1191	1	1	1	1	1	1
50	CVE-2022-1213	1	1	1	1	1	1
51	CVE-2022-1239	1	✗	✗	✗	1	1
52	CVE-2022-31386	4	3	4	4	4	4
53	CVE-2022-31830	1	1	1	✗	1	1
54	CVE-2022-38292	1	✗	✗	✗	✗	✗
55	CVE-2022-40357	1	✗	✗	✗	✗	✗
56	CVE-2022-41477	1	1	1	1	1	1
57	CVE-2022-41497	3	2	3	3	3	3
58	CVE-2022-46998	1	✗	✗	✗	1	1
59	CVE-2022-47872	4	✗	✗	✗	4	4
60	CVE-2023-1938	1	✗	✗	✗	✗	✗
61	CVE-2023-1977	1	✗	✗	✗	✗	✗
62	CVE-2023-2927	6	✗	✗	✗	✗	6
63	CVE-2023-34959	1	✗	✗	✗	1	1
64	CVE-2023-3744	1	1	1	1	1	1
65	CVE-2023-39108	2	✗	2	✗	2	2
66	CVE-2023-39109	2	✗	2	✗	2	2
67	CVE-2023-39110	1	✗	1	✗	1	1
68	CVE-2023-40969	1	✗	✗	✗	✗	✗
69	CVE-2023-41054	1	✗	✗	✗	1	1
70	CVE-2023-41055	1	✗	✗	✗	1	1
71	CVE-2023-4651	1	1	✗	✗	1	1
72	BMLT-1	4	✗	✗	✗	✗	✗
73	BN-1	1	✗	✗	✗	✗	✗
74	CHL-1	2	✗	✗	✗	✗	✗
75	COL-1	1	✗	✗	✗	✗	✗
76	CR-1	10	✗	✗	✗	✗	✗
77	CVE-2023-38515	1	1	✗	✗	1	1
78	CVE-2023-46725	1	✗	✗	✗	✗	✗
79	CVE-2023-46730	1	✗	✗	✗	✗	✗
80	CVE-2023-46736	1	✗	✗	✗	✗	✗
81	CVE-2023-48005	2	2	2	2	2	2
82	CVE-2023-48006	1	✗	✗	✗	✗	✗
83	CVE-2023-4878	1	1	1	1	1	1
84	CVE-2023-49159	2	✗	✗	✗	✗	✗
85	CVE-2023-49746	2	✗	✗	✗	✗	✗
86	CVE-2023-50374	1	✗	✗	✗	✗	✗
87	CVE-2023-50621	4	✗	✗	✗	✗	✗
88	CVE-2023-50622	1	✗	✗	✗	1	1
89	CVE-2023-51676	4	✗	✗	✗	✗	✗
90	CVE-2023-52233	1	✗	✗	✗	✗	✗
91	CVE-2023-5798	1	✗	✗	✗	✗	✗
92	CVE-2023-5877	1	1	1	1	1	1
93	CVE-2024-22134	3	✗	✗	✗	✗	✗
94	CVE-2024-32430	1	✗	✗	✗	✗	✗
95	CVE-2024-33629	2	✗	✗	✗	✗	✗
96	CVE-2024-35633	1	✗	✗	✗	✗	✗
97	CVE-2024-35635	1	✗	✗	✗	✗	✗
98	CVE-2024-35637	1	✗	✗	✗	✗	✗
99	CVE-2024-37098	2	✗	✗	✗	✗	✗
100	CVE-2024-38791	1	✗	✗	✗	✗	✗
101	IC-1	4	✗	✗	✗	✗	✗
102	MAC-1	6	✗	✗	✗	✗	✗
103	NONE-1	3	✗	✗	✗	✗	✗
104	FR-1	2	✗	✗	✗	✗	✗
105	PL-1	3	✗	✗	✗	✗	✗
106	SI-1	2	✗	✗	✗	✗	✗
	Total Detected Path#	**207**	**45**	**51**	**43**	**79**	**88**

and 13 as vulnerable. The full version of this paper [48] provides details on the affected applications, including their versions, root causes of the detected vulnerabilities, and their system impacts.

Comparison with Ripses. Rips produces 45 true positive paths (27 SSRFs) with only built-in sources and sinks. When third-party sources and sinks are added, Rips* produces 58 true positive paths (4 more SSRFs). The detection rate is 78.3% and 72.0% lower than Artemis's. The primary factor is that Rips does not support object-oriented features, overlooking vulnerabilities in class

methods. For example, in Figure 8, Rips fails to detect the taint propagation from the method call on line #22 to the method on line #28, resulting in a false negative.

Comparison with PHPJoerns. PHPJoern detects 51 true positive paths (30 SSRFs) with only built-in sources and sinks, which is 75.4% fewer than Artemis. After incorporating third-party sources and sinks, PHPJoern* identifies 80 true positive paths (49 SSRFs), still 61.4% fewer than Artemis. The primary reason is PHPJoern's incomplete call graph construction. PHPJoern struggles with object-oriented programming where methods in different classes may have the same name. For example, in Figure 8, two methods named `getData` exist in different classes, but PHPJoern cannot distinguish them, leading to early termination of taint paths on line #13 and therefore false negatives.

```
1  class Request {
2   public function getData (...) {} //PHPJoern ignores
3  }                                  //calls to getData
4                                     //due to duplicate
5  class PdfWriterService {           //function names in
6   public function getData (...) {} //different classes
7  }
8
9  class ApiController { //TChecker skips updateProducts due
10   public function updateProducts() //to missing call path
11  {       //Unknown type to Phan and Psalm
12   $this-> Product =$this->getTable()->get('Products');
13   $productsData=$this->getRequest()-> getData ('data.data');
14   $products=[]; //foreach is not modeled by TChecker
15   foreach ($productsData as $product){
16    $manufacturerIsOwner=$this->Product->find(...)->count();
17    if (!$manufacturerIsOwner) {
18     throw ...;
19    }
20    $products[]=[$product['image']];
21   }      //Unknown type to Phan and Psalm
22   $this-> Product -> changeImage ($products);
23  }                  //Method call ignored by Rips
24 }
25
26 class ProductsTable {
27  // $products propagates to sink
28  public function changeImage($products) {...}
29 }
```

Fig. 8. A newly found CVE-2023-46725 has been confirmed by the developer. Rips, PHPJoern, TChecker, Psalm, and Phan all fail to detect the vulnerability.

Comparison with TCheckers. TChecker and TChecker† identify 43 true positive paths (25 SSRFs) with only built-in sources and sinks, and TChecker* identifies 52 true positive paths (28 SSRFs) when third-party sources and sinks are included. The detection rate is 79.2% and 74.9% lower than Artemis's. TChecker's design of starting analysis from the top-level function of each PHP file leads to a comparative low detection rate of SSRFs. Specifically, many true SSRF paths do not have an explicit caller-callee relationship with the top-level functions, causing TChecker's misdetection. Furthermore, TChecker's propagation rules are incomplete. TChecker does not model tainted arrays iterated through `foreach` loops. For example, in Figure 8, method `updateProducts` is skipped from the analysis of Tchecker. Additionally, the taint is propagated in a `foreach` loop at line #15, which is also ignored by TChecker.

Comparison with Psalms. Psalm and Psalm† detect 79 true positive paths (48 SSRFs) with only built-in sources and sinks, which is 61.8% fewer than Artemis. With third-party sources and sinks, Psalm* detects 132 true positive paths (65 SSRFs), still 36.2% fewer. The main reason is that Psalm fails to handle implicit call graph connection, particularly in cases where type inference fails. Psalm relies on type annotations in comments to determine variable types. If the type of a variable cannot be inferred, method calls on that variable are ignored, cutting off the taint propagation path. For example, in Figure 8, Psalm fails to infer the type of `$this->Product`, ignoring the `changeImage` method call, leading to a false negative.

Comparison with Phans. Phan and Phan† produce 88 true positive paths (50 SSRFs) with only built-in sources and sinks, and Phan* produces 172 true positive paths (85 SSRFs) when third-party sources and sinks are added. Although surpassing other tools, Phan's detection rate falls short of Artemis by 57.5% and 16.9%, respectively. Like Psalm, Phan fails to create implicit call graph connections and cannot infer the type of `$this->Product` in Figure 8, missing the implicit call target on line #22 and thus failing to propagate the taint. Additionally, Phan does not model implicit

Table 5. Detection results of Artemises, TCheckers, Phans, Ripses, PHPJoerns, and Psalms.

Approaches	TP CVE Nums		TP Path Nums		FP Path Nums		Precision
Artemis	106		207		15		93.2%
Rips	27	-74.5%	45	-78.3%	139	+8.27x	24.5%
Rips*	31	-70.8%	58	-72.0%	149	+8.93x	28.0%
PHPJoern	30	-71.7%	51	-75.4%	100	+5.67x	33.8%
PHPJoern*	49	-53.8%	80	-61.4%	113	+6.53x	41.5%
TChecker	25	-76.4%	43	-79.2%	57	+2.80x	43.0%
TChecker*	28	-73.6%	52	-74.9%	63	+3.20x	45.2%
TChecker†	25	-76.4%	43	-79.2%	36	+1.40x	54.4%
Psalm	48	-54.7%	79	-61.8%	138	+8.20x	36.4%
Psalm*	65	-38.7%	132	-36.2%	149	+8.93x	47.0%
Psalm†	48	-54.7%	79	-61.8%	92	+5.13x	46.2%
Phan	50	-52.8%	88	-57.5%	156	+9.40x	36.1%
Phan*	85	-19.8%	172	-16.9%	166	+10.07x	50.9%
Phan†	50	-52.8%	88	-57.5%	124	+7.27x	41.5%
Artemis^{a}	53	-50.0%	97	-53.1%	11	-26.7%	89.8%
Artemis^{c}	89	-16.0%	181	-12.6%	15	+0%	92.3%
Artemis^{t}	102	-3.8%	198	-4.3%	153	+9.20x	56.4%
Artemis^{p}	106	+0%	207	+0%	35	+1.33x	85.5%

data flow relationships, such as those established by the `extract` function, leading to further false negatives.

As shown in the preceding comparisons, while existing approaches boost their true positive rates by including *third-party sources and sinks*, they still suffer low detection coverage (16.9% to 74.9% fewer than Artemis) due to their lack of support for implicit call targets and implicit data flows.

Comparison with Ablated Artemises. Removing the *third-party source and sink identification* module from Artemis results in only 97 true positive paths (53 SSRFs), 53.1% fewer than the full version. The low number of true positive paths highlights the widespread use of third-party sources and sinks in modern PHP applications and the critical importance of this module.

Removing *statically inferred call graph construction* results in 181 true positive paths (89 SSRFs), 12.6% fewer than Artemis. The 26 false negatives are due to lack of support for implicit call targets: 4 from missing magic methods, 18 from known method names with variable class names, 3 from known class names with variable method names, and 2 from both class and method names being unknown. Although implicit call targets are not frequently used, ignoring them causes false negatives. One case (CVE-2023-40969) involves both magic methods and known class names with variable method names.

By replacing the *rule-based taint analysis* module with generic propagation rules from Phan, 198 TPs are produced, which is 4.3% fewer than Artemis due to implicit data flow.

Removing the *false positive pruning* module does not impact detection coverage, because this module is specifically designed to reduce false positives deterministically, and conditions that cannot be statically verified are ignored.

4.2.2 False Positives. The detection precision of Artemis, Rips, PHPJoern, TChecker, Psalm, Phan, and the ablated versions of Artemis are presented in Table 5. Artemis produces 15 false positives, achieving a precision rate of 93.2%.

Comparison with Ripses. Rips and Rips* generate 149 false positives with third-party sources and sinks, and 139 without, having the lowest precision rates of 28.0% and 24.5%. This is mainly due to Rips's over-tainting, assuming tainted parameters make return values tainted. Additionally, Rips lacks support for safety string analysis and considers any string constructed with a tainted variable as tainted, regardless of whether the tainted input controls the critical parts of the string. These over-tainting issues lead to a significant number of false positives.

Comparison with PHPJoerns. PHPJoern and PHPJoern* generate 100 false positives with third-party sources and sinks and 113 without, with a precision rate of 41.5% and 33.8%, respectively. PHPJoern has high false positive rates mainly because of the lack of safety string analysis and false positive pruning.

Comparison with TCheckers. TChecker and TChecker* produce 63 false positives with third-party sources and sinks and 57 without, resulting in a precision rate of 45.2% and 43.0%, respectively. The relatively low number of false positives is due to many functions (that are not explicitly invoked by the top-level functions) not being analyzed by TChecker, which also results in fewer true positive paths. Consequently, TChecker remains less precise compared to Artemis. Compared to TChecker, TChecker† reduces false positives to 36, a decrease of 36.8%.

Comparison with Psalms. Psalm and Psalm* produce 149 false positives with third-party sources and sinks and 138 without, achieving a precision rate of 47.0% and 36.4%. Psalm† produces 92 false positives, a 33. 3% decrease compared to Psalm.

Comparison with Phans. Phan and Phan* have the most false positives, 166 with and 156 without, resulting in precision rates of 50.9% and 36.1%. With the false positive pruning module, Phan† produces 124 false positives, resulting in a 20.5% decrease.

TCheckers, Psalms, and Phans have high false positive rates due to their generic propagation rules. Without *safety string analysis*, they over-taint URL strings when tainted input does not affect critical URL parts. The absence of *path condition analysis* also leads to reporting infeasible paths, further increasing their false positives. We have observed that integrating the *path condition analysis* module into existing approaches significantly reduces false positives, for example, TChecker† reduces the false positives reported by TChecker by 36.8%. However, without the augmented taint propagation and clearance rules in Artemis, existing approaches relying solely on post-processing for pruning false positives are imprecise.

Comparison with Ablated Artemises. Removing the *third-party source and sink identification* module results in 11 false positives, slightly reducing the number of false positives, but at the cost of 53.1% fewer true positive paths, highlighting the importance of this module.

Replacing the *rule-based taint analysis* module with propagation rules from Phan results in 153 false positives, 9.2 times more than the full version of Artemis. This sharp increase is due to the lack of safety string assurance rules, which prevents proper handling of cases where user-controlled input cannot control critical parts of request-sending URLs, such as query parameters or parts of file-accessing URLs.

Removing the *false positive pruning* module leads to 35 false positives, an increase of 133.3% compared to the full version of Artemis. The increase in false positives suggests the importance of handling path conditions in SSRF detection.

Discussion. Artemis has 15 false positives due to three key factors. First, four false positives arise from always unsatisfied conditions involving array variables, which Artemis cannot accurately recognize. Tracking individual elements in PHP arrays is challenging [43] because of PHP's lack of formal semantics and the dynamic nature of arrays. Second, six false positives occur due to user-defined functions that remove slashes (/) from an input string, making it no longer a valid URL and preventing SSRF. Third, five false positives arise from intended features where developers intentionally allow arbitrary requests for debugging purposes. These features/functions are protected by file-based authentication, requiring prior access to the server's file system before debugging can occur. Pruning these cases requires domain-specific knowledge, making false positive pruning challenging.

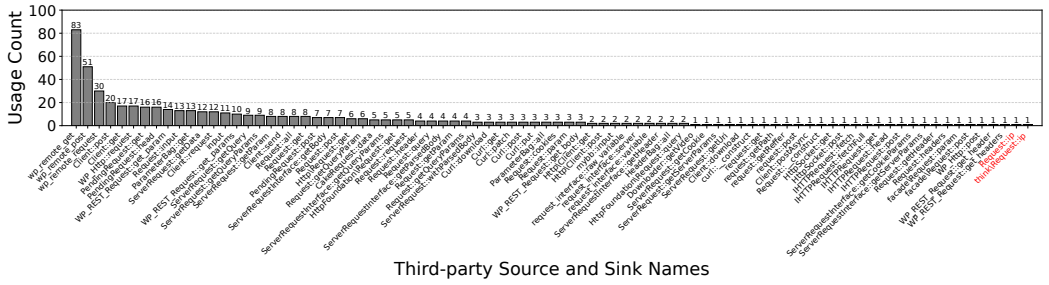


Fig. 9. Usage frequency of third-party sources and sinks. The two third-party sources missed by GPT-4o are marked in red.

4.3 Source and Sink Identification Results

4.3.1 Generalization Analysis. We evaluate the generalizability of our LLM-based source and sink identification by extending our analysis to multiple state-of-the-art models, including GPT-4o, LLaMA-3.1 [63], and Claude-3.5-Sonnet [21], without human refinement.

First, we manually create the ground truth by examining PHPDocs and code examples, classifying all 7,406 third-party APIs from 250 applications as sources, sinks, or neither. This *manual* process yields 44 sources and 40 sinks as our ground truth. We then measure precision and recall by comparing the identification outputs of state-of-the-art models with our ground truth. We present the precision, recall, and cost of GPT-4o, Claude-3.5, and LLaMA-3.1 in Table 6. We select GPT-4o as our main assistant model primarily because (1) it has a balanced precision and recall with the highest F1 score; (2) it is relatively cost-effective; and (3) its standard API format enables easy extraction of raw model outputs in JSON. Although GPT-4o misses two sources that are associated with the MipCMS application, they do not lead to any SSRFs. GPT-4o misses the two sources due to insufficient context in PHPDocs.

Table 6. Precision, recall, f1-score and cost of GPT-4o, LLaMA-3.1, and Claude-3.5.

LLM	Precision	Recall	F1	Cost ($)
GPT-4o	91.1%	97.6%	**94.3%**	21
Claude-3.5	**92.0%**	96.4%	94.2%	40
LLaMA-3.1	87.3%	**98.8%**	92.7%	**19**

4.3.2 Sensitivity Analysis. To evaluate how scalable Artemis is in identifying third-party sources and sinks for use in detecting SSRFs in other PHP applications, we conduct a sensitivity analysis on all 84 third-party sources and sinks. First, we calculate how many applications each third-party source and sink appears in, as shown in Figure 9. We conducted the experiment 11 times, progressively incorporating 0%, 10%, ..., up to 100% of the most frequently used sources and sinks in each iteration to identify SSRFs across all applications. [7]

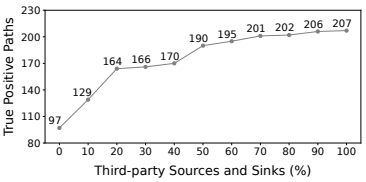


Fig. 10. Number of true positive paths detected with 0% to 100% of frequently-used third-party sources and sinks.

As shown in Figure 10, the number of detected true positive paths increases logarithmically from 97 to 207 as the inclusion of third-party sources and sinks rises from 0% to 100%, converging sublinearly. With an inclusion of 0%, 97 true positive paths involving only PHP built-in sources and sinks are detected. We also observe that with 50% of inclusion, we achieve 84.5% coverage. This aligns with the frequency

[7] The two sources missed by GPT-4o are added in the last 10% iteration and do not impact our result.

Table 7. Precision, recall, and construction time of call graphs in Artemis, Rips, PHPJoern, TChecker, Psalm, and Phan on 10 sample applications.

App	Artemis	Rips	PHPJoern	TChecker	Psalm	Phan
	P / R / Time (ms)	P / R / Time (ms)	P / R / Time (ms)	P / R / Time (ms)	P / R / Time (ms)	P / R / Time (ms)
Cockpit	70.9% / **99.0%** / 19.8	98.1% / 10.9% / 1.1	**99.8%** / 11.9% / 11.7	95.9% / 38.9% / 233.2	98.7% / 64.4% / 17.8	96.3% / 41.6% / 7.9
i-librarian	96.4% / **100%** / 48.5	**100%** / 76.1% / 4.4	**100%** / 80.1% / 26.7	99.4% / 92.9% / 47.3	99.0% / 94.1% / 169.1	99.1% / 93.2% / 35.5
leadin	**100%** / **100%** / 9.5	**100%** / 72.6% / 1.9	**100%** / 85.1% / 7.1	**100%** / **100%** / 16.9	**100%** / **100%** / 43.1	**100%** / **100%** / 9.3
LibreY	**100%** / **100%** / 4.2	**100%** / 89.4% / 0.6	**100%** / 98.3% / 6.9	**100%** / **100%** / 6.1	**100%** / **100%** / 13.7	**100%** / 95.0% / 3.4
LinkAce	66.9% / **98.3%** / 168.1	**98.2%** / 5.5% / 1.4	98.1% / 5.4% / 22.5	96.3% / 86.0% / 451.8	96.7% / 87.3% / 77.1	96.5% / 86.8% / 110.2
NoneCms	67.4% / **98.5%** / 17.3	99.5% / 50.5% / 1.5	**99.7%** / 51.0% / 12.2	98.0% / 71.2% / 951.1	98.3% / 75.5% / 33.8	98.2% / 73.5% / 9.2
rconfig	97.4% / **100%** / 37.1	**100%** / 79.8% / 7.5	**100%** / 82.7% / 23.4	99.8% / 92.9% / 536.7	99.8% / 97.4% / 301.3	**100%** / 95.5% / 30.7
WeBid	99.1% / **99.5%** / 67.9	**99.8%** / 50.6% / 16.5	**99.8%** / 52.1% / 28.3	99.5% / 96.1% / 282.5	99.1% / 96.6% / 94.2	99.6% / 96.4% / 57.1
wp-fastest-cache	98.3% / **100%** / 44.5	**100%** / 83.7% / 1.6	**100%** / 87.7% / 12.4	98.9% / 95.8% / 18.4	99.4% / 96.5% / 127.3	**100%** / 94.6% / 31.3
yzmcms	78.7% / **98.8%** / 124.3	99.6% / 32.7% / 8.1	**99.8%** / 33.9% / 29.7	99.1% / 39.9% / 329.7	99.2% / 41.8% / 215.9	99.4% / 48.5% / 104.9

distribution in Figure 9, where infrequently used sources and sinks appear only once or twice, having minimal impact on the overall detection rate. Therefore, we believe that our identified set of third-party sources and sinks can be applied to future application SSRF detection, offering a solid coverage rate.

4.4 Call Graph Construction Results

In this section, we conduct a small-scale experiment to evaluate the effectiveness of our call graph construction approach in Section 3.2. We compare Artemis with alternative methods, including the unique name-matching approach used in Rips and PhpJoern, and the type inference-based approach used in TChecker, Psalm, and Phan. We select ten moderately sized applications from our benchmark and manually extract and verify their caller-callee pairs as the ground truth.

Table 7 presents the precision, recall, and speed of different call graph construction approaches. Among all tools, Artemis consistently achieves the highest recall, ranging from 98.3% to 100%, indicating superior call graph coverage. However, existing tools exhibit poor recall, significantly lower than Artemis in some cases. Among them, the name matching-based call graphs (Rips and PhpJoern) show the lowest recall, with a minimum of just 5.4%. The reason Artemis cannot reach 100% recall is due to incorrect type inference, which is an open problem [25, 78]. On the other hand, Artemis has the lowest precision due to our over-approximation strategy to identify as many implicit calls as possible. This relaxation is then tightened by our safety string rules in Section 3.3 and false positive pruning in Section 3.4 to prevent excessive false positives, as many of them are either untainted or never reach the sink. Therefore, the final SSRF detection in Artemis maintains high precision despite the lower precision of its call graph module. As shown in Table 5, when including implicit call graph construction, Artemis does not produce any additional false positives.

Rips and PHPJoern are the fastest because they use simple unique name matching. TChecker, Psalm, Phan, and Artemis take longer due to type inference. Although Artemis is about 30% slower than Phan (on which Artemis is based) for implicit call targets, it still completes in less than 1 second for all cases, making the extra construction time worthwhile for better SSRF coverage.

4.5 Detection Time

Artemis executes in 10.4 to 328.1 seconds, averaging 69.5 seconds. Rips completes analysis in 0.2 to 2045.2 seconds, averaging 34.3 seconds. TChecker analyzes projects in 0.4 to 652.1 seconds, with an average of 33.7 seconds. PHPJoern takes 0.5 to 210.7 seconds, averaging 22.4 seconds. Psalm requires 0.7 to 921.3 seconds, with an average of 28.1 seconds. Phan finishes analysis in 0.6

to 157.4 seconds, averaging 20.9 seconds. Compared to other tools, Artemis on average requires twice as much time to analyze a project primarily because the call graph generated by Artemis contains implicit calls ignored by other tools. Each connected implicit call site uncovers additional hidden call chains, thereby increasing the workload for taint propagation tracking and false positive pruning.

Rips, TChecker, and PHPJoern cannot complete the analysis on some projects because they are designed for PHP 7.0 and lack forward compatibility. For example, in Rips, PHPJoern, and TChecker, the iterative and recursive analysis of PHP 7.4 arrow syntax in lambdas [20] leads to repeated evaluations of the same functions or string elements, with intermediate results stored in memory during each iteration or recursion. This continuous accumulation of data without termination eventually results in an out-of-memory (OOM) condition. The detailed detection time for each application with all detection tools can be found in the full version of this paper [48].

4.6 Limitations

False Negatives. Artemis cannot detect second-order SSRF where user input is first stored in a database and later retrieved. For example, in Figure 11, user input is saved in the database on line 3, read from the database on line #8, and finally used in a sink function on line #14. To detect such vulnerabilities, we need to model database APIs, infer database schemas, and identify the affected columns during data transfer, which is outside the scope of our current work. Past studies [40, 75] focusing on application-specific database operations in second-order vulnerabilities can help improve Artemis to detect such SSRF vulnerabilities.

```
1  $f = esc_sql($_POST['file']);
2  $q = "INSERT INTO $t VALUES('$f',...)";
3  $wpdb->query($q);
4  ...
5  $r = $wpdb->get_row("SELECT...FROM $t...");
6  readfile($r->file);
```

Fig. 11. A second-order SSRF (CVE-2020-24141) missed by Artemis.

Language Features. So far, Artemis does not support all dynamic PHP features, such as argument unpacking, which involves spreading array elements into argument lists. Integrating this feature into Artemis requires inferring all array elements when constructing implicit call graphs and updating our taint propagation rules accordingly. To make matters worse, nested arrays, where an array element is another array, further complicate this feature. Although we have not observed these features being exploited in existing vulnerabilities, they could lead to undetected vulnerabilities.

5 Conclusion

In this paper, we introduce Artemis, a static taint analysis tool to effectively identify SSRF vulnerabilities. Artemis achieves enhanced detection coverage and accuracy by integrating LLM-based source and sink identification, explicit and implicit call graph construction, augmented rule-based taint analysis, and false positive pruning based on path conditions. Artemis operates entirely automatically without any application-specific domain knowledge. We have implemented a prototype of Artemis and tested it with 250 open-source applications. The results reveal that Artemis successfully identifies 207 true vulnerable paths (106 true SSRFs) and 15 false positives, significantly outperforming existing tools. Of the 106 SSRFs, 35 are newly discovered. We have reported the 35 new SSRFs to the developers, and 24 SSRFs have been confirmed by the developers and assigned CVE IDs.

Acknowledgments

We would like to thank the anonymous reviewers for their insightful feedback and valuable comments. This work was supported by the Shanghai Sailing Program 22YF1428600. Yutian Tang's

work was supported in part by the National Natural Science Foundation of China under grant 62202306.

Data-Availability Statement

Artemis's source code and detailed reports of newly detected SSRFs are available at https://zenodo.org/records/13353039.

References

[1] 2019. What We Can Learn from the Capital One Hack. https://krebsonsecurity.com/2019/08/what-we-can-learn-from-the-capital-one-hack.
[2] 2022. CakePHP. https://cakephp.org/.
[3] 2022. Curl Class. https://www.phpcurlclass.com/.
[4] 2022. Guzzle, PHP HTTP client. https://github.com/guzzle/guzzle.
[5] 2022. Laravel Request. https://laravel.com/api/11.x/Illuminate/Http/Request.html.
[6] 2022. phan. https://github.com/phan/phan.
[7] 2022. phan-plugin. https://github.com/wikimedia/mediawiki-tools-phan-SecurityCheckPlugin.
[8] 2022. Yii Request. https://www.yiiframework.com/doc/api/2.0/yii-web-request.
[9] 2023. Awesome-Selfhosted. https://github.com/awesome-selfhosted/awesome-selfhosted.
[10] 2023. CVE database. https://cve.mitre.org/index.html.
[11] 2023. CWE-918: Server-Side Request Forgery (SSRF). https://cwe.mitre.org/data/definitions/918.html.
[12] 2023. function.extract. https://www.php.net/manual/en/function.extract.php.
[13] 2023. Joern. https://github.com/joernio/joern.
[14] 2023. Magic Methods. https://www.php.net/manual/en/language.oop5.overloading.php.
[15] 2023. php-ast. https://github.com/nikic/php-ast.
[16] 2023. Popular Plugins. https://wordpress.org/plugins/browse/popular.
[17] 2023. PSR-5: PHPDoc. https://github.com/php-fig/fig-standards/blob/master/proposed/phpdoc.md.
[18] 2023. Superglobals. https://www.php.net/manual/en/language.variables.superglobals.php.
[19] 2023. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.
[20] 2024. Arrow Functions. https://www.php.net/manual/en/functions.arrow.php.
[21] 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet.
[22] 2024. GPT-4o. https://platform.openai.com/docs/models/gpt-4o.
[23] 2024. Object Inheritance. https://www.php.net/manual/en/language.oop5.inheritance.php.
[24] 2024. OWASP Top 10 - 2021. https://owasp.org/Top10/.
[25] 2024. Phan Type Inference Wiki. https://github.com/phan/phan/wiki/Phan-Config-Settings#allow_overriding_vague_return_types.
[26] 2024. Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering.
[27] 2024. psalm. https://github.com/vimeo/psalm/.
[28] 2024. Reflection. https://www.php.net/manual/en/intro.reflection.php.
[29] 2024. Type System. https://www.php.net/manual/en/language.types.intro.php.
[30] 2024. Variable Functions. https://www.php.net/manual/en/functions.variable-functions.php/.
[31] 2024. Variable-length argument lists. https://www.php.net/manual/en/functions.arguments.php#functions.variable-arg-list.
[32] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 377–392. doi:10.5555/3277203.3277232
[33] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE european symposium on security and privacy (EuroS&P)*. IEEE, 334–349. doi:10.1109/EuroSP.2017.14
[34] Souphiane Bensalim, David Klein, Thomas Barber, and Martin Johns. 2021. 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* (Online, United Kingdom) *(EuroSec '21)*. Association for Computing Machinery, New York, NY, USA, 27–33. doi:10.1145/3447852.3458718
[35] Tim Berners-Lee, Larry M Masinter, and Mark P. McCahill. 1994. Uniform Resource Locators (URL). RFC 1738. https://www.rfc-editor.org/info/rfc1738
[36] Stefano Calzavara, Michele Bugliesi, Silvia Crafa, and Enrico Steffinlongo. 2015. Fine-grained detection of privilege escalation attacks on browser extensions. In *Programming Languages and Systems: 24th European Symposium on*

Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 24. Springer, 510–534. doi:10.1007/978-3-662-46669-8_21

[37] Bofei Chen, Lei Zhang, Xinyou Huang, Yinzhi Cao, Keke Lian, Yuan Zhang, and Min Yang. 2024. Efficient Detection of Java Deserialization Gadget Chains via Bottom-up Gadget Search and Dataflow-aided Payload Construction. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 150–150. doi:10.1109/SP54263.2024.00150

[38] Yi Chen, Luyi Xing, Yue Qin, Xiaojing Liao, XiaoFeng Wang, Kai Chen, and Wei Zou. 2019. Devils in the guidance: predicting logic vulnerabilities in payment syndication services through automated documentation analysis. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 747–764. doi:10.5555/3361338.3361390

[39] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis.. In *NDSS*, Vol. 14. 23–26.

[40] Johannes Dahse and Thorsten Holz. 2014. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 989–1003. doi:10.5555/2671225.2671288

[41] Benjamin Eriksson, Amanda Stjerna, Riccardo De Masellis, Philipp Rüemmer, and Andrei Sabelfeld. 2023. Black Ostrich: Web Application Scanning with String Solvers. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 549–563. doi:10.1145/3576915.3616582

[42] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 1789–1804. doi:10.1145/3460120.3484745

[43] Daniele Filaretti and Sergio Maffeis. 2014. An executable formal semantics of PHP. In *ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28*. Springer, 567–592. doi:10.1007/978-3-662-44202-9_23

[44] Riccardo Focardi, Flaminia L. Luccio, and Marco Squarcina. 2012. Fast SQL blind injections in high latency networks. In *2012 IEEE First AESS European Conference on Satellite Telecommunications (ESTEL)*. 1–6. doi:10.1109/ESTEL.2012.6400112

[45] Mohammad Ghasemisharif, Chris Kanich, and Jason Polakis. 2022. Towards Automated Auditing for Account and Session Management Flaws in Single Sign-On Deployments. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1774–1790. doi:10.1109/SP46214.2022.9833753

[46] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: effective fuzzing of web applications for server-side vulnerabilities. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 267, 18 pages. doi:10.5555/3698900.3699167

[47] Jin Huang, Junjie Zhang, Jialun Liu, Chuang Li, and Rui Dai. 2021. UFuzzer: Lightweight Detection of PHP-Based Unrestricted File Upload Vulnerabilities Via Static-Fuzzing Co-Analysis. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*. 78–90. doi:10.1145/3471621.3471859

[48] Yuchen Ji, Ting Dai, Zhichao Zhou, Yutian Tang, and Jingzhu He. 2025. *Artemis: Toward Accurate Detection of Server-Side Request Forgeries through LLM-Assisted Inter-Procedural Path-Sensitive Taint Analysis*. Technical Report. arXiv:2502.21026 [cs.CL]

[49] Martin Johns, Björn Engelmann, and Joachim Posegga. 2008. XSSDS: Server-Side Detection of Cross-Site Scripting Attacks. In *2008 Annual Computer Security Applications Conference (ACSAC)*. 335–344. doi:10.1109/ACSAC.2008.36

[50] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, V. N. Venkatakrishnan, and Yinzhi Cao. 2023. Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1059–1076. doi:10.1109/SP46215.2023.10179352

[51] Zifeng Kang, Song Li, and Yinzhi Cao. 2022. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites.. In *NDSS*.

[52] Sojhal Ismail Khan, Dominika C Woszczyk, Chengzeng You, Soteris Demetriou, and Muhammad Naveed. 2021. Characterizing Improper Input Validation Vulnerabilities of Mobile Crowdsourcing Services. In *Proceedings of the 37th Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC '21)*. Association for Computing Machinery, New York, NY, USA, 944–956. doi:10.1145/3485832.3485888

[53] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. 2024. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web, In 45th IEEE Symposium on Security and Privacy. *Proceedings of 45th IEEE Symposium on Security and Privacy*. doi:10.1109/SP54263.2024.00098

[54] Soheil Khodayari and Giancarlo Pellegrino. 2021. JAW: Studying Client-side CSRF with Hybrid Property Graphs and Declarative Traversals. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2525–2542.

[55] Soheil Khodayari and Giancarlo Pellegrino. 2023. It's (DOM) Clobbering Time: Attack Techniques, Prevalence, and Defenses. In *2023 IEEE Symposium on Security and Privacy (SP)*. 1041–1058. doi:10.1109/SP46215.2023.10179403

[56] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding client-side business flow tampering vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 222–233. doi:10.1145/3377811.3380355

[57] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. 2020. FUSE: Finding File Upload Bugs via Penetration Testing.. In *NDSS*.

[58] Sebastian Lekies, Ben Stock, and Martin Johns. 2013. 25 million flows later: large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 1193–1204. doi:10.1145/2508859.2516703

[59] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2021. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 268–279. doi:10.1145/3468264.3468542

[60] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 143–160.

[61] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1468–1484. doi:10.1109/SP40001.2021.00062

[62] Z. Liu, K. An, and Y. Cao. 2024. Undefined-oriented Programming: Detecting and Chaining Prototype Pollution Gadgets in Node.js Template Engines for Malicious Consequences. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 120–120. doi:10.1109/SP54263.2024.00121

[63] Team Llama3. 2024. The Llama 3 Herd of Models. arXiv:2407.21783 [cs.AI] https://arxiv.org/abs/2407.21783

[64] Changhua Luo, Penghui Li, and Wei Meng. 2022. 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* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2175–2188. doi:10.1145/3548606.3559391

[65] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia. 2014. Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives. In *Proceedings of the 23rd International Conference on World Wide Web* (Seoul, Korea) *(WWW '14)*. Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/2566486.2568024

[66] Marius Musch, Robin Kirchner, Max Boll, and Martin Johns. 2022. Server-Side Browsers: Exploring the Web's Hidden Attack Surface. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security* (Nagasaki, Japan) *(ASIA CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1168–1181. doi:10.1145/3488932.3517414

[67] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. 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* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 616–628. doi:10.1145/2810103.2813680

[68] OpenAI. 2023. GPT-4 Technical Report. https://arxiv.org/abs/2303.08774.

[69] Giancarlo Pellegrino, Onur Catakoglu, Davide Balzarotti, and Christian Rossow. 2016. Uses and abuses of server-side requests. In *Research in Attacks, Intrusions, and Defenses: 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings 19*. Springer, 393–414. doi:10.1007/978-3-319-45719-2_18

[70] Giancarlo Pellegrino, Martin Johns, Simon Koch, Michael Backes, and Christian Rossow. 2017. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1757–1771. doi:10.1145/3133956.3133959

[71] Joanna CS Santos, Mehdi Mirakhorli, and Ali Shokri. 2024. Seneca: Taint-Based Call Graph Construction for Java Object Deserialization. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 1125–1153. doi:10.1145/3649851

[72] Youkun Shi, Yuan Zhang, Tianhao Bai, Lei Zhang, Xin Tan, and Min Yang. 2024. RecurScan: Detecting Recurring Vulnerabilities in PHP Web Applications. In *Proceedings of the ACM Web Conference 2024* (Singapore, Singapore) *(WWW '24)*. Association for Computing Machinery, New York, NY, USA, 1746–1755. doi:10.1145/3589334.3645530

[73] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, Yinzhi Cao, Ziwen Wang, Yudi Zhao, Zongan Huang, and Min Yang. 2022. Backporting Security Patches of Web Applications: A Prototype Design and Implementation on Injection Vulnerability Patches. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 1993–2010.

[74] Marco Squarcina, Mauro Tempesta, Lorenzo Veronese, Stefano Calzavara, and Matteo Maffei. 2021. Can I Take Your Subdomain? Exploring Same-Site Attacks in the Modern Web. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2917–2934.

[75] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. 2023. Splendor: Static Detection of Stored XSS in Modern Web Applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1043–1054. doi:10.1145/3597926.3598116

[76] Karthika Subramani, Roberto Perdisci, and Maria Konte. 2021. Detecting and measuring in-the-wild DRDoS attacks at IXPs. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th International Conference, DIMVA 2021, Virtual Event, July 14–16, 2021, Proceedings 18*. Springer, 42–67. doi:10.1007/978-3-030-80825-9_3

[77] Avinash Sudhodanan, Soheil Khodayari, and Juan Caballero. 2020. Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks. (2020).

[78] Apil Tamang. 2015. *A constraint-based method for flow-sensitive static type analysis Of PHP using the Rascal meta-programming platform*. East Carolina University.

[79] Leon Trampert, Ben Stock, and Sebastian Roth. 2023. Honey, I Cached our Security Tokens Re-usage of Security Tokens in the Wild. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses* (Hong Kong, China) *(RAID '23)*. Association for Computing Machinery, New York, NY, USA, 714–726. doi:10.1145/3607199.3607223

[80] Erik Trickel, Fabio Pagani, Chang Zhu, Lukas Dresel, Giovanni Vigna, Christopher Kruegel, Ruoyu Wang, Tiffany Bao, Yan Shoshitaishvili, and Adam Doupé. 2023. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In *2023 IEEE symposium on security and privacy (SP)*. IEEE, 2658–2675. doi:10.1109/SP46215.2023.10179317

[81] Tom Van Goethem, Iskander Sanchez-Rola, and Wouter Joosen. 2023. Scripted Henchmen: Leveraging XS-Leaks for Cross-Site Vulnerability Detection. In *2023 IEEE Security and Privacy Workshops (SPW)*. 371–383. doi:10.1109/SPW59333.2023.00038

[82] E. Wang, J. Chen, W. Xie, C. Wang, Y. Gao, Z. Wang, H. Duan, Y. Liu, and B. Wang. 2024. Where URLs Become Weapons: Automated Discovery of SSRF Vulnerabilities in Web Applications. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 216–216. doi:10.1109/SP54263.2024.00198

Received 2024-10-16; accepted 2025-02-18