TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications

Changhua Luo
Chinese University of Hong Kong
Hong Kong SAR, China
chluo@cse.cuhk.edu.hk

Penghui Li
Chinese University of Hong Kong
Hong Kong SAR, China
phli@cse.cuhk.edu.hk

Wei Meng
Chinese University of Hong Kong
Hong Kong SAR, China
wei@cse.cuhk.edu.hk

ABSTRACT

PHP applications provide various interfaces for end-users to interact with on the Web. They thus are prone to taint-style vulnerabilities such as SQL injection and cross-site scripting. For its high efficiency, static taint analysis is widely adopted to detect taint-style vulnerabilities before application deployment. Unfortunately, due to the high complexity of the PHP language, implementing a precise static taint analysis is difficult. The existing taint analysis solutions suffer from both high false positives and high false negatives because of their incomprehensive inter-procedural analysis and a variety of implementation issues.

In this work, we present TChecker, a context-sensitive inter-procedural static taint analysis tool to detect taint-style vulnerabilities in PHP applications. We identify that supporting objects and type systems is critical for statically analyzing programs written in the dynamic language PHP. We first carefully model the PHP objects and the related object-oriented programming features in TChecker. It then iteratively performs an inter-procedural data-flow analysis on PHP objects to refine object types, thus could precisely identify the call targets. We also take a considerable amount of effort in supporting other dynamic features of PHP such as dynamic includes.

We comprehensively evaluated TChecker on a diverse set of modern PHP applications and demonstrated its high effectiveness in vulnerability detection. Specifically, TChecker successfully detected 18 previously unknown vulnerabilities in these PHP applications. We compared TChecker with the related static analysis tools and found that it significantly outperformed them by detecting more vulnerabilities. TChecker could also find all the vulnerabilities the existing tools detect with a relatively good precision. We release the source code of our prototype implementation to facilitate future research.

CCS CONCEPTS

• **Security and privacy** → **Web application security**.

KEYWORDS

PHP; Taint Analysis; Inter-Procedural Analysis; Taint-Style Vulnerabilities

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3559391

ACM Reference Format:
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 (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3559391

1 INTRODUCTION

PHP is the most popular server-side language. According to a recent study [48], it is used by around 78% of websites today. PHP applications are commonly deployed on web servers and they directly react towards user interactions and requests, *e.g.*, user clicks. Such an interaction mode opens a huge surface for various attacks where attackers can exploit the vulnerabilities in the applications. In particular, PHP applications are prone to *taint-style vulnerabilities*—a class of vulnerabilities that occur when the user-supplied data is used in critical operations without sufficient sanitization.

Taint-style vulnerabilities are fatal. An attacker might exploit a taint-style vulnerability to extract sensitive data, escape her privilege, or compromise a web server, *etc.* [2–4, 10]. In the real world, reports have shown that 64% of companies experienced web-based attacks [45]; an attack against commercial banks could even cost over 100K USD per hour to each bank [25].

Due to the high severity and prevalence of taint-style vulnerabilities, it is important to apply defensive techniques to mitigate such threats and protect the relevant parties. Many approaches have been proposed to detect taint-style vulnerabilities in the wild [16, 18, 21, 22, 26, 28, 29, 34]. They can generally be classified into dynamic approaches and static approaches. The dynamic approaches inject attack payloads and check the corresponding outputs in deployed web applications [16, 22, 29]. However, because of the dynamic features and navigational complexities in modern web applications [18], dynamic approaches usually have a limited code coverage. Consequently, they cannot reveal the vulnerabilities in application code that they fail to reach and suffer from high false negatives. Besides, dynamic approaches usually require manually configuring the applications, thus having low scalability and cannot be easily applied to a large number of applications. Static taint analysis, on the other hand, is fully automated and scalable, and it can achieve a high code coverage. As a result, static analysis has been widely adopted and has shown great promise in detecting taint-style vulnerabilities [21, 26, 28, 36, 46].

We systematically study the existing static taint analysis solutions for PHP applications, and find they also suffer from high false positives and high false negatives. We identify three inherent limitations in existing solutions. First, the taint analysis in prior works is not context-sensitive. Many taint-style vulnerabilities in modern PHP applications are context-dependent. They are only manifested when the corresponding functions or methods are called in specific contexts. However, existing works fail to implement a comprehensive context-sensitive analysis because they cannot precisely infer the target functions of method calls. Specifically, finding target functions of method calls requires identifying the class of the receiver objects (*e.g.*, `$obj` in `$obj->f()`) in the call sites. This is challenging because PHP is a dynamically-typed language and the receiver objects are usually defined without specifying any type annotations. As a result, the type of an object in PHP can hardly be determined before runtime execution. The commonly used approach to finding target functions is to match the called function/method names (`f()` in `$obj->f()`) with the function definitions. However, this approach leads to either over-tainting (by overly matching call targets) [1, 26, 28, 46] or under-tainting (by conservatively matching no targets) [21, 36].

Second, because of the difficulties in statically analyzing PHP code, the existing tools simplify taint analysis with some assumptions that might not (always) hold. For instance, the state-of-the-art commercial tool RIPS-A [28] assumes a call site in PHP applications always invokes the same target function. However, the call sites in PHP applications can invoke different functions in different calling contexts. Other open-source taint analysis tools make strong assumptions that the return values of call sites are tainted if any of the arguments are tainted [1]. Such simplifications could lead to an imprecise taint analysis and introduce false positives and false negatives in vulnerability detection.

Third, the existing tools do not support commonly used advanced PHP features. PHP supports a few complex features, including dynamic constants (*e.g.*, `constant($con)`, variable variables (*e.g.*, `$$a`), *etc.* Although those important features have been well discussed in prior works [26], we find that no open-source taint analysis tool has modeled these advanced features. Failure in modeling these features could lead to the early termination of a data flow analysis and thus cause false negatives.

In this work, we aim to address the above-mentioned limitations of the existing works and develop a precise static inter-procedural taint analysis. However, the high complexity of PHP language brings several technical challenges. First, it is challenging to precisely determine the call targets of method calls in PHP applications. Specifically, analyzing call targets requires an inter-procedural data flow analysis on the receiver objects to infer their types. However, performing the inter-procedural data flow analysis in turn requires a comprehensive call relationship analysis. Second, implementing a taint tracking algorithm to overcome the limitations of existing taint tracking implementation requires non-trivial engineering efforts. In particular, the taints could be propagated from a call site to different target functions in different calling contexts. We should not only implement an inter-procedural taint analysis but also consider the calling contexts when propagating taints into different target functions. As a function or a method can be called in various contexts at runtime, to statically support multiple calling contexts in a scalable way is very challenging. Third, we need to design a taint analysis tool that is able to model the complex features of PHP applications. Otherwise, even that we could develop a good call relationship analysis and a good inter-procedural taint analysis, the tool might still not be able to propagate the taints correctly because of the inability to model the semantics of a few complex PHP features or operations.

We present TChecker, a context-sensitive inter-procedural static taint analysis tool for PHP applications. TChecker overcomes the above-mentioned challenges with several new techniques. Specifically, it *iteratively* performs an inter-procedural data-flow analysis on PHP objects to infer their types or values. This helps it precisely identify the call targets of the method calls to incrementally build a precise call graph, which further benefits the inter-procedural data flow analysis. Furthermore, since a call site might invoke different callee functions in different contexts, TChecker analyzes the call sites in a context-sensitive manner. Instead of always regarding all the possible target functions as the target functions of a call site, it determines the target function based on the calling context and propagates the taints to the correct target function. We also spend a considerable amount of effort on modeling the complex and common PHP features in TChecker and addressing several implementation challenges in taint analysis of PHP applications. Specifically, TChecker tracks taints in object properties, analyzes a few commonly used dynamic features such as dynamic includes, and encodes the semantics of PHP built-in functions, *etc.* This would enable TChecker to comprehensively track the taint propagation in complex PHP applications.

We implemented a prototype of TChecker using 7.3K lines of Java code. We thoroughly evaluated TChecker on a dataset of 17 representative PHP applications. TChecker detected 131 true positive taint-style vulnerabilities in the dataset, including 18 previously unknown vulnerabilities. Our comparison to the state-of-the-art tools further shows that TChecker significantly outperforms them by detecting 50 more vulnerabilities. TChecker could also find all the vulnerabilities the existing tools detected with a relatively good precision. We characterized the 18 new vulnerabilities and found that they could be potentially exploited for severe security consequences. For example, a new XSS vulnerability in osCommerce2 (v2.3.4.1) could allow unprivileged attackers to compromise the victim user accounts. We have responsibly reported the new vulnerabilities to the relevant vendors. At the time of writing, 5 vulnerabilities, including 2 CVEs, have been acknowledged or patched.

In summary, this paper makes the following contributions:

- We designed and implemented TChecker, a precise context-sensitive inter-procedural static taint analysis framework with object-oriented programming support for PHP applications.
- TChecker significantly outperformed the state-of-the-art tools and detected 18 previously unknown vulnerabilities.
- We release the source code of TChecker at https://github.com/cuhk-seclab/TChecker to facilitate future research.

2 BACKGROUND AND MOTIVATION

In this work, we focus on PHP—the most popular server-side programming language, used by 78.4% of websites [48]. We introduce first the PHP features (§2.1) and PHP program analysis (§2.2), then taint-style vulnerabilities, and existing detection methods and their limitations (§2.3).

2.1 PHP Features

As a language especially on the web, PHP is designed with complex and dynamic features [26, 33]. In this section, we introduce several important PHP features for program analysis.

Method Calls. PHP applications usually declare lots of methods. The methods are invoked through an instantiated receiver object, *i.e.*, `$o->f()`, or a static keyword, *e.g.*, `parent->f()`. A method call site can invoke different methods depending on the types of receivers at runtime.

Variable Functions. The name of the callee function at a call site in PHP applications can be built dynamically. Specifically, PHP evaluates a variable and then executes the function with the name the variable evaluates to. For instance, `$f = "g"; $f()` would invoke a function named as `g`. Callbacks are common instances of variable functions. In the following sections, we call the variables used in variable functions as *function name variables*.

Dynamic Includes. File inclusion statements in PHP open a specified file and evaluate its PHP code. The file names can also be built dynamically, *e.g.*, `require $a`.

Object Properties. Object properties (*e.g.*, `$obj->prop`) are commonly used in PHP applications. Similar to global variables, object properties can be defined and then used in different function scopes. The types of receivers in object properties (*i.e.*, `$obj` in `$obj->prop`) are known mostly at runtime.

2.2 PHP Program Analysis

In this subsection, we introduce the challenges in static PHP program analysis.

Call Relationship Analysis. As pointed out by prior works, variables (especially, tainted variables) are often defined and used in different functions [21, 51]. Performing inter-procedural program analysis is thus important for tracking the data flow of these variables. Analyzing call relationships is the prerequisite of inter-procedural program analysis. Finding the target functions is straightforward if no variable is used in the call sites (*e.g.*, `a()`), because the target functions can be directly inferred based on the called function names. However, inferring the target functions of method calls (*e.g.*, `$obj->f()`) and variable functions (*e.g.*, `$f()`) is very hard, because the types of the receiver objects (*e.g.*, `$obj`) or the values of the function name variables (*e.g.*, `$f`) are determined by PHP at runtime.

The following features make it even more challenging to precisely reason about the call relationships in PHP programs. First, PHP is a weakly-typed language and the variables are usually declared without specifying any type annotations. As a result, one has to perform data flow analysis on a receiver object to find the class that it is instantiated into. Second, PHP is dynamically-typed; the type of the receiver object is checked and determined at runtime and can change in different calling contexts. Therefore, instead of assuming that the target functions of call sites are deterministic, analyzing call relationships requires a *context-sensitive* data flow analysis on the receiver objects in call sites. Third, PHP offers the `spl_autoload_register` function that is used to automatically load external classes and interfaces. The developers could invoke the methods in external classes without explicitly including the corresponding PHP files. This increases the difficulties in type inference as the objects might be in any type declared in the applications.

Data Flow Analysis. Data flow analysis allows us to obtain the possible set of values of the variables that we are interested in. While it is the basic technique and is well studied in other languages, data flow analysis on PHP programs brings new challenges. First, we need to perform call relationship analysis to track the data flow across function boundaries. However, performing call relationship analysis on PHP programs requires a data flow analysis on the receiver objects in call sites. The analysis thus becomes a cycle and making it very difficult to perform data flow analysis precisely because of the difficulty in doing the call relationship analysis. Besides, PHP supports a few unique features, including dynamic constants (*e.g.*, `constant($con)`), variable variables (*e.g.*, `$$var`), *etc.* Failure in modeling such features would lead to early termination of the data flow analysis. Yet statically analyzing the above features is an open challenge because of the dynamic nature of PHP code [26].

2.3 Taint-Style Vulnerabilities and Taint Analysis

2.3.1 *Taint-Style Vulnerabilities.*

Web applications often provide many features for end-users and act correspondingly on user inputs and interactions, *e.g.*, form submissions or clicks. A security vulnerability occurs when the user-supplied data (*i.e.*, taint) is not sufficiently sanitized and is used in critical operations (*i.e.*, sinks) of the application. Such vulnerabilities are known as *taint-style vulnerabilities* [26]. Taint-style vulnerabilities have been a persistent security threat to web applications. Common types of vulnerabilities, such as cross-site-scripting (XSS), SQL injection (SQLi), *etc.*, are instances of taint-style vulnerabilities. An attacker might exploit such a flaw by providing malicious inputs to change the expected behavior of the application, *e.g.*, injecting malicious code.

2.3.2 *Taint Analysis.*

Taint analysis is the *de facto* approach to finding taint-style vulnerabilities in practice. It tracks the propagation of *taints* originated from external *sources* (*e.g.*, untrusted user-supplied data) along the program execution, checks if the tainted data could flow to the critical program locations (*sinks*), and finally reports the sinks that can potentially be manipulated by attackers.

Generally, there are both dynamic and static taint analysis approaches. Dynamic methods [22, 29] often inject special payloads and check their reappearance in a black-box manner on deployed web applications. However, due to the complexity of modern web applications [18], such methods can only reach and test a small proportion of an application. The rest of the application is left unchecked and the vulnerabilities in it thus cannot be revealed. Besides, dynamic methods are not scalable because they require manual efforts to deploy and configure an application.

Static methods analyze the source code to report potential taint-style vulnerabilities [18, 21, 26, 36, 46]. They can achieve a high code coverage, and are more efficient and scalable.

2.3.3 *Existing Solutions.*

Many static taint analysis tools have been proposed to detect taint-style vulnerabilities. Since call graph and data flow analysis are essential components in static taint analysis, researchers have proposed different designs to handle the challenges we discussed in §2.2. We summarize the key ideas below.

(Partial) Matching of Function Names. Analyzing call relationships is necessary for tracking the taints propagated across function boundaries. To address the challenge of determining method call targets, one common approach is to ignore the receiver objects in call sites and match the called method names only. There are different design choices to this end. RIPS [1] compares the called method names `f()` with function definitions in the whole PHP program and regards the matched ones as the target functions of the call sites `$obj->f()`. This approach can find all the target functions but has many false positives because the methods in other class scopes might be mistakenly regarded as the call target functions. PHPJoern instead considers only the function calls with unique called function names. For example, if there is only one method definition with method name `f()`, then the target function of call site `$obj->f()` can be determined. However, PHPJoern might suffer from high false negatives—it maps only 29% of call sites in Joomla according to our experiment results (see §6.2.1).

Data Flow Analysis. An advanced approach to resolving call relationships is to leverage data flow analysis. The proprietary version of RIPS (we call it RIPS-A) is a state-of-the-art tool in doing data flow analysis for PHP programs. It performs an intra-procedural data flow analysis on a receiver object to infer its type, then identifies the target function of a method call. It also performs the same analysis on some function name variables (*e.g.*, `$a` in `$a()`) to find the target functions in variable function calls. However, because of the lack of an inter-procedural analysis, RIPS-A cannot infer the types or values of the variables (*e.g.*, function parameters) that are assigned in other function scopes. Therefore, it also fails to support a few common call patterns in PHP applications. Moreover, RIPS-A is not open-sourced. Researchers have to implement the static analysis techniques (*e.g.*, the call graph) on their own or upon open-source tools such as PHPJoern [18].

Function Summary. Performing inter-procedural data flow analysis requires analyzing the target function of a call site. To avoid duplicated analysis, a function summary is created to summarize the data flow within a function [28, 36]. Because a function might contain call sites, the prior works assume that each call site invokes the same function(s) in whatever calling contexts and then merge the function summaries of all the called functions as the effect of one call site [28]. Finally, each function is analyzed once and the function summary is used when the function is invoked again. The use of function summaries greatly improves analysis efficiency, as a function needs not to be repeatedly analyzed many times. However, it sacrifices analysis precision for performance, because the use of function summaries implies that the data flow of each function is deterministic regardless of the calling contexts.

2.3.4 A Motivating Example. We illustrate the limitations of existing taint analysis tools with an example. Listing 1 has an XSS vulnerability. The `src` property in the IMG tag is defined as a return value of the `base_url()` function. The user-controlled input `$_POST['image']` is used as an argument of that function call and it is finally outputted without undergoing sanitization. Therefore, the attackers can craft payloads through the `$_POST['image']` parameter to perform XSS attacks.

To detect such a vulnerability, a taint analysis tool shall analyze the function calls in lines 2, 5, 13 and propagate taints from arguments (*e.g.*, `$uri`) to local variables (*e.g.*, `$uri` in line 5, 13, and 17) and finally the return value of function `base_url()` in line 2. We study how the existing open-sourced taint analysis tools handle such a case. PHPJoern fails to report this vulnerability because it does not propagate taints from function returns (*i.e.*, the taints in return statements in lines 13 and 17 are not propagated). RIPS can detect this vulnerability while it also reports a lot of false positives. In addition to the problem of over-estimated call graph (by partially matching function names), RIPS performs taint analysis in a context-insensitive way—it taints the return value of a function call if any argument is tainted. As a result, RIPS taints the return value of function `base_url()` even `uri` has been properly sanitized in `base_url()`. We are unable to apply RIPS-A because its source code is not publicly available. However, detecting such a vulnerability is challenging because performing intra-procedural analysis is insufficient to infer the type of the receiver object in line 5. RIPS-A could mistakenly propagate the taints to the incorrect target functions, thus it could also suffer from false negatives.

3 PROBLEM STATEMENT

In this section, we first present the research scope and research goals of this work. We then discuss the research challenges we encounter.

3.1 Research Scope and Research Goals

In this work, we aim to improve static analysis techniques to assist detection of taint-style vulnerabilities in PHP applications. We focus on static approaches because of its wide adoption in both academia [21, 36, 41, 42, 46] and industry [12, 14]. We have identified several limitations in the existing static taint analysis tools. In this work, we aim to address these limitations by developing a comprehensive inter-procedural static taint analysis tool for detecting taint-style vulnerabilities in PHP applications.

3.2 Research Challenges

Designing and implementing a static taint analysis tool for PHP applications has been known to be challenging. First, it is difficult to handle the call relationship inference and inter-procedural data flow analysis. Specifically, the interleaving of inter-procedural analysis and call relationship analysis makes it notoriously difficult to address both problems. As an example, to identify the target function of the call site `get_instance()->config->base_url()` in line 5 in Listing 1, we need to infer the type of the receiver object `get_instance()->config`. To this end, we perform data flow analysis to find the class(es) it can be instantiated into; then infer all the functions that invoke the function `base_url()` in line 3 because the object can be instantiated in different calling contexts. However, the callers of `base_url()` can only be determined after we complete call relationship analysis, which is not the case because we are still in the process of finding target functions of the call site in line 5.

Besides, we need to propose a new inter-procedural taint tracking algorithm. The taint tracking algorithm should be precise—the

```
1  <?php
2  <img ... src="<?= base_url('attachments/blog_images/'
       . $_POST['image']) ?>">
3  function base_url($uri = '', $protocol = NULL) {
4      ...
5      return get_instance()->config->base_url($uri,
        $protocol);
6  }
7
8  //system/core/Config.php
9  class CI_Config {
10     ...
11     public function base_url($uri = '', $protocol =
        NULL){
12         ...
13         return $base_url.$this->_uri_string($uri);
14     }
15     protected function _uri_string($uri) {
16         ...
17         return $uri;
18     }
19 }
```

Listing 1: An XSS vulnerability found by TChecker.

taints should only be propagated to the target functions that are invoked in the current calling contexts, and efficient—the algorithm shall efficiently determine the target function of a call site in a specified calling context because the call site might be executed many times.

Third, we need to model a few complex yet commonly-used features in PHP programs (*e.g.*, dynamic includes and object properties) to address the implementation issues of existing tools. Modeling the dynamic features in PHP is known to be hard, yet it allows us to understand the semantics of more PHP operations and to find the vulnerabilities that reside deeply in the applications.

4 TCHECKER

We present TChecker, a precise context-sensitive inter-procedural static taint-analysis tool with object-oriented feature support to identify taint-style vulnerabilities in PHP applications. We first introduce the high level design of TChecker in §4.1, then its two major components in §4.2 and §4.3.

4.1 Overview

As discussed earlier, it is important to perform an inter-procedural taint analysis to detect the vulnerabilities in PHP applications. The main challenge of doing inter-procedural taint analysis on PHP programs is to analyze call relationships. To this end, TChecker first builds a call graph, which helps it later perform more accurate taint propagation across function calls. The benefit of building call graph before propagating taints is twofold. First, the call graph is built to contain already all the possible target functions of a call site in a *context-sensitive* way. Thus, given a specific calling context, TChecker can directly select the correct target function of a call site from the possible target functions. The selection is a mapping process from calling contexts to target functions (see §4.3.2) and it allows TChecker to obtain the target functions efficiently. Without constructing the call graph in advance, TChecker has to repeatedly infer the call target of a call site each time it is met (even in the same context) in the taint analysis. Second, the call graph allows TChecker to perform selective taint analysis. For instance, TChecker could skip the taint analysis for a call site if all its target functions do not use any tainted variable. This could avoid unnecessary analysis on irrelevant functions and thus improve the overall analysis efficiency.

The architecture of TChecker is depicted in Figure 1. Given the source code of a PHP application, TChecker incrementally constructs its call graph by inferring the call targets at each call site; it then performs a whole-program context-sensitive inter-procedural taint analysis to identify taint-style vulnerabilities.

4.2 Constructing Call Graph

While call graph is the basic component in many security applications such as vulnerability detection [26, 28], sandboxing [23], *etc.*, finding the target functions of method calls and variable functions has been an open problem. Analyzing target functions of call sites in a context-insensitive way could lead to the over-approximation problem [28]. As discussed in [28], RIPS-A cannot find target functions of call sites whose receiver objects are defined in other functions because it employs only an intra-procedural analysis. RIPS-A would still suffer from inaccurate call target inference even in some cases where it could correctly identify the possible call target functions of a call site, because it merges the function summaries of all the target functions at a call site whereas the call site invokes only one function in a specific context.

However, it would also be very inefficient if we analyze a call site each time it is encountered, because the call site might be analyzed many times in similar contexts. To tackle this problem, we design a new algorithm that analyzes a call statement only when it is executed in a *new* context.

Specifically, TChecker takes two stages to incrementally construct a call graph. In the first stage, TChecker performs a backward data-flow analysis on the receiver objects and variable function names to find the *function name* of each call site (§4.2.1). In the second stage, it connects one call site to its call targets (§4.2.2). More importantly, TChecker performs the two steps iteratively. By adding new call target functions (in new contexts) to the call graph, new data flow relationships can be discovered, allowing TChecker to determine the values of more function name variables or classes of more receiver objects based on the new data flow relationship and then to further find corresponding new call targets.

We use the example in Listing 1 to further illustrate the idea. The `base_url()` function at line 3 is invoked by multiple call sites in the application. When TChecker connects a new call edge from the call site in line 2 to the function definition in line 3, it checks if the newly introduced caller functions (*i.e.*, all the function nodes reaching the `base_url()` function node through the newly connected call edge) contain assignments to `get_instance()->config`. It there is one such assignment in function `p()`, TChecker performs a backward data flow analysis on `get_instance()->config` starting from function `p()` to infer its type. TChecker then updates the target functions of the call site `get_instance()->config->base_url()` in line 4.

4.2.1 Identifying Function Names of Call Sites.

Finding the call target of a call site requires identifying the name of its called function. Besides directly invoking a function using its literal name, call sites

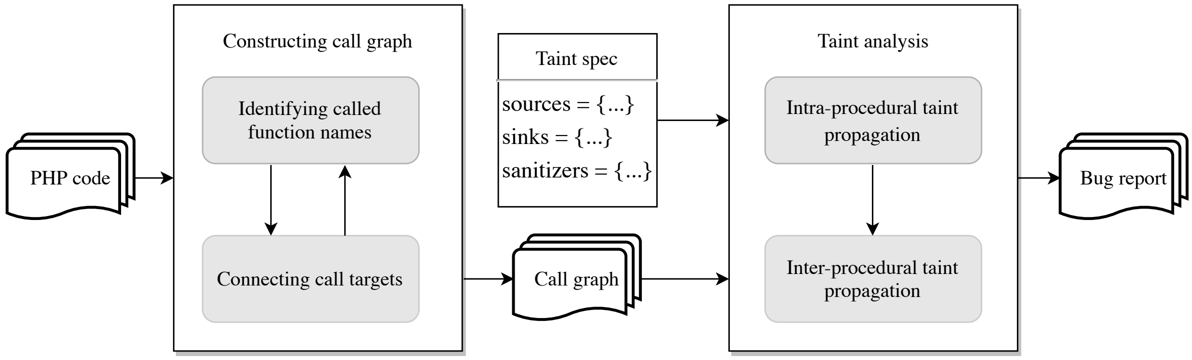

Figure 1: The overall architecture of TChecker.

in PHP include method calls and variable function calls. Therefore, TChecker infers the type of the receiver object of a method call, and the value of the variable of a variable function call.

Inferring Receiver Object Type. Method calls are invoked with receiver objects that can be in multiple classes, which might all define a function with the same name. Therefore, TChecker needs to infer the types of receivers to find the correct called methods. To this end, it iteratively analyzes the data flow of the receiver, `$rec`. `$rec` might be assigned by a local variable or a variable defined outside the current function (*e.g.*, a global variable). For simplicity, we call the variables that are defined/assigned in other function scopes as *external variables*. If `$rec` is assigned by a local variable `$loc`, TChecker backward analyzes the data flow of `$loc`. If `$loc` is instantiated from a `new` statement, *i.e.*, `new A()`, TChecker directly returns `A` as the type of `recv`. Otherwise, TChecker backward traces `$loc` till it finds the type of `$loc` or `$loc` is assigned by an external variable, `$ev`. TChecker then performs inter-procedural data flow analysis on `$ev` in the following ways.

(1) If `$ev` is a function parameter with a type declaration, TChecker returns the declared type as the type of `$ev`. If `$ev` is a function parameter without a type declaration, TChecker searches the call sites of the current function, `f()`. It then backward traces the corresponding argument in each call site to find the possible types of the function parameter `$ev`. When a new call site of `f()` is added, TChecker iteratively repeats the above analysis.
(2) If `$ev` is the return value of a function call (*e.g.*, `$ev=$obj->func()`), TChecker enters the call target functions (could be multiple) to analyze the data flow. Specifically, it performs the same backward data flow analysis at each return statement in each call target function. The function call might also use variables, *i.e.*, `$obj->func()` or `$func()`. If the type or value of the corresponding variable is unknown, TChecker also performs this data flow analysis on it.
(3) If `$ev` is an object property, *e.g.*, `$ev=$o->p`, TChecker infers the type of the parent object `$o` for further determining the type of the property `p`. TChecker first searches the type `C` of the object `$o` based on two constraints: 1) the class `C` must contain an object property with name `p`; 2) the object property `C::p` must be instantiated from a `new` statement or be assigned by other variables that are instantiated from a `new` statement. If there is only one candidate, the type of `$o` can be known directly. Otherwise, TChecker continues the backward data-flow analysis further on `$o`. Next, it searches all classes that are instantiated to `$o->p` to determine the class of `p` hence that of `$ev`.
(4) If `$ev` is a global variable used in the current function `f()`, TChecker backward tracks the data flow to `$ev` in `f()` and its caller functions. TChecker also enters call targets of sibling function calls in `f()` and in its caller functions to ensure it does not miss any assignment to `$ev`. Similarly, `$ev` might be assigned by another variable. TChecker thus identifies the classes instantiated to the relevant variables to determine the type of `$ev`. The process is also iterative. When TChecker finds a new caller function of `f()`, it repeats the analysis in the new caller function.

Analyzing Variable Functions. TChecker attempts to statically infer the value of the variables in a variable function (*e.g.*, `$func()`) to identify the function name. It models the semantics of relevant statements and performs also the backward data-flow analysis. Different from receiver object type inference, inferring the variable values is much more challenging. Many complex PHP language features, including the numerous string operations, dynamic typing (*e.g.*, `'test'+1='test1'`), and dynamic constants (*e.g.*, `constant($const)`), *etc.*, are involved during the analysis to construct the *strings* representing variable function names. To the best of our knowledge, only the proprietary RIPS-A partially supports such complex features with an intra-procedural data analysis.

Although TChecker performs an inter-procedural analysis on the variable functions, it might not be able to statically infer the values of a few variables because of the high complexity of modern PHP applications. It thus ignores the call sites it fails to analyze, following the common practice [28]. This could lead to false negatives because it might fail to find target functions for a few call sites. In our evaluation, TChecker fails to analyze up to tens of variable functions among the tens of thousands of call sites. We believe that such a low failure rate is acceptable especially compared with other open-source taint analysis tools that do not support variable functions.

TChecker might also infer part of the function name for a few call sites. For example, the called function names of the call site `$y()` where `$y` is previously assigned with `'get'.$x`, might not be fully inferred statically if `$x` is unknown. However, its data flow points out that the called function names shall start with `'get'`. TChecker thus uses a wildcard to replace the unknown variables, *e.g.*, `$x` in `('get'.$x)()`. Such partial values could help match with the target functions in practice.

4.2.2 Connecting Call Targets. TCHECKER then connects one call site with the corresponding call targets. Specifically, it first matches the called function name of one call site with all function definitions. It then filters out the invalid call target candidates that are introduced when matching called function names containing a wildcard.

Determining Called Function Names. Because of class inheritance, the called function names may not be literally equal to that of the target function definition. To this end, TCHECKER parses the `extends` keywords and builds an inheritance tree. It then extracts the class from the called function name. If the class name is a static keyword (*e.g.*, `parent`) or `$this`, TCHECKER replaces it with the corresponding type. Afterward, TCHECKER searches the target function by matching available function definitions (*i.e.*, namespace\(class_name::)function_name) with the called function name *i.e.*, `N\C::p`. If TCHECKER does not find a match in class C, it iteratively replaces the class name with its parent class name. It stops the analysis until it finds a matching function definition or the class has no parent class.

Validating Call Targets. TCHECKER performs function prototype checks to remove the invalid call target candidates. It determines a call target candidate as invalid in the following situations: 1) the static function call invokes non-static functions; 2) the number of arguments in a call site is less than the number of required parameters in the candidate; 3) the access modifier of a candidate method is `private` and the call site is not in the class scope of that method; 4) the access modifier of a candidate method is `protected` and the call site is not in the scope of a class inheriting the class of that method. Call relationships that meet the above conditions are filtered out because they violate PHP coding standards [5].

4.2.3 Simulating File Includes. TCHECKER also analyzes the inclusion statements (*e.g.*, `require`) because they introduce new code defined in other PHP files. It handles the inclusion statements as call sites and the corresponding included files as user-defined functions in the call graph. We face the similar challenges of determining variable names when analyzing dynamic includes. In the case where the file name is not a static string, TCHECKER performs a similar data-flow analysis to reconstruct the file name. Prior works suggest to use a PHP profiler to track file dependencies of a PHP application [18, 20]. In the case where profiling is allowed, TCHECKER is also able to interpret the profiling outputs[1]. Dynamic approaches such as crawling are required when profiling PHP applications. We tested a few PHP applications with a spider (*i.e.*, a crawler) and found the spider could help discover additionally less than 1% of all call edges. Therefore, profiling is only provided as an option to complement static analysis.

4.2.4 An Example. We finally illustrate how TCHECKER constructs a call graph with the example in Listing 1. There are three call sites in this example. TCHECKER can determine the target function of call site `base_url()` in line 2 by simply matching the called function name `base_url`. To get the target function of the call site `get_instance()->config->base_url()`, TCHECKER first determines the type of the object property `get_instance()->config`. It finds that `CI_Config` is the only class that 1) has been instantiated to an object property with property name `config` and 2) has a method with name `base_url()`. Please note in such a case we do not need to perform type analysis for the return value of the `get_instance()` function call. Therefore, TCHECKER connects the function in line 11 as the only target function invoked by the call site in line 5. Finally, to obtain the target functions of the call site `$this->_uri_string()` in line 13, TCHECKER performs an object analysis on the keyword `$this`. Instead of simply identifying `CI_Config` as the class of `$this` where `CI_Config` is the class defining the caller method of the call site `$this->_uri_string()`, TCHECKER infers the class of `$this` based on the current object class to handle polymorphism. For instance, the function `base_url()` containing call site `$this->_uri_string` could be invoked by a method defined in the class `Sub`, where `Sub` is the child class of `CI_Config`. In such a case, TCHECKER would identify the called function name to be `Sub::_uri_string()` and find the overridden target function accordingly.

4.3 Taint Analysis

TCHECKER performs a static context-sensitive inter-procedural taint analysis to find the sensitive operations that can potentially be controlled and exploited by attackers. Similar to other works [21, 26, 36], TCHECKER treats external inputs (*e.g.*, `$_GET`) as taint *sources*. It reports a vulnerability if the tainted data could be used in a sensitive operation, *i.e.*, the *sink*. In our current implementation, TCHECKER detects three common types of taint-style vulnerabilities—XSS, SQLi and Denial-of-Service (DoS) vulnerabilities. It regards database (*e.g.*, `mysql_query()`) operations, content generation (*e.g.*, `echo()`) operations, and variables in loop termination conditions as sinks, and considers the corresponding sanitizers (*e.g.*, `htmlspecialchars()`, `mysql_real_escape_string()`). TCHECKER is implemented as an multi-tag taint analysis tool. It can separately propagate and maintain the taints for different types of vulnerabilities. For instance, it maintains taint status separately for XSS, thus the XSS sanitizer functions remove only the corresponding XSS taint tag. One can easily extend TCHECKER to detect other types of taint-style vulnerabilities by including new sink functions and sanitization rules.

TCHECKER starts its analysis from the top-level function of each PHP file. It propagates the taints and reports a vulnerability if any tainted variables are used in *sinks* without sanitization. It completes the analysis after all functions on the call graph have been analyzed. We describe the details below.

4.3.1 Intra-Procedural Taint Propagation. TCHECKER propagates taint information from the right hand operands to the left hand operands for each assignment-like statement. A simple statement might contain nested assignment-like statements. For instance, in a function call `$a = f(func($b))`, there are three assignment-like statements: the right hand operands are `$b`, return values of `func()`, and return values of `f()`; the left hand operands are the parameter of `func()`, the parameter of `f()`, and `$a`, respectively. To this end, TCHECKER first identifies assignment-like statements based on the

[1] TCHECKER currently supports XDebug.

predefined symbols (*e.g.*, operator `=`, function calls, *etc.*). It recursively parses the nested assignment-like statements until it finally analyzes the outermost one (*i.e.*, `$a = f()`).

To parse an assignment-like statement, TChecker first analyzes the taint status of the assignment operands. If an operand is sanitized, TChecker cleans its taint tag. Otherwise, TChecker considers an operand tainted in the following situations: 1) the operand is a *source*; 2) the operand is literally equal to a tainted variable; 3) the operand might be a tainted array element (see §5); 4) the operand is a tainted return value. Since TChecker tracks data flow in object properties, the taints are also maintained for individual properties. TChecker then propagates taints. If the right hand operands are tainted, TChecker taints the left hand operands; otherwise, if the untainted right hand operands are assigned to a tainted left hand operand, TChecker cleans the taint tag in the left hand operand.

4.3.2 Inter-Procedural Taint Propagation. TChecker analyzes call statements and propagates taints across function boundaries. It should be noted that a precise inter-procedural taint analysis is very expensive. First, the taints might be propagated through multiple function boundaries. Besides, each call statement shall be analyzed because its call targets might define new sources or use tainted global variables. To address this issue, TChecker skips the analysis of the call targets that are not likely to be called in current calling contexts or use no tainted variable.

Selecting Target Functions. TChecker first selects the target functions likely to be called in the current calling context. The call site might have many call targets in the call graph. However, it invokes one call target in a specific calling context. TChecker takes heuristics to narrow the potential call targets down. We observe that the ambiguous call targets are mostly methods, because call sites having multiple call targets are mostly non-static method calls. To call a method, the application code first invokes the *class constructor*. Also, the methods using object properties (*i.e.*, `$this->prop`) usually have data dependency relationships with other methods in the same class scope. Therefore, TChecker prioritizes a call target `f()` if other functions (including the constructors) in the same file of `f()` have ever been called. If there is no such a function, TChecker selects call targets in the sub-folders until it finds at least one. Finally, TChecker obtains a subset of call targets from call graph. This approach avoids over-tainting and improves analysis efficiency. However, it could lead to under-tainting because TChecker may miss a true call target. Nevertheless, the results in §6 show its effectiveness: TChecker could find all the vulnerabilities that existing tools detect.

Preprocessing Target Functions. TChecker preprocesses a call target, *e.g.*, `f()`, to find whether it uses any tainted variables. If not, the taint analysis for such callee functions could be skipped. Specifically, TChecker finds all the assignment-like statements using external variables into the set `asg_global`. It then collects the functions of these statements into the set `func_global`. It also collects all the functions that include assignment-like statements using *source* into the set `func_source`. This search process takes a one-time cost when TChecker parses PHP code into ASTs. Afterward, given a call target `f()`, TChecker searches `f()` and its callee functions in `func_global` and `func_source`. TChecker skips the analysis for function `f()` directly if it finds no match and no argument of `f()` is tainted. The idea of selective taint analysis has been discussed and shown to be able to improve the analysis efficiency in other languages such as C [24].

Analyzing Target Functions. TChecker finally enters the target functions of a call statement. Before that, it identifies if a target function is recursively called, *e.g.*, a function `f()` calls itself. TChecker skips the analysis for recursive function calls, following the common practices [26, 28]. TChecker simulates context switches during function invocation and initializes the tainted local variables to be any tainted function parameters. It then performs the standard intra-procedural taint analysis in the target function. A call site might have multiple call targets. If any of the target functions at a call site returns a tainted value, the left hand operand in the corresponding assignment statement in the caller is tainted.

5 IMPLEMENTATION

We implemented a prototype of TChecker upon PHPJoern [21], with 7.3K lines of Java code. We use php-ast [8] to parse PHP source code into ASTs, and use PHPJoern to build the Control Flow Graphs (CFGs) and Data Dependency Graphs (DDGs) for the intra-procedural data flow analysis. The original DDG cannot distinguish different object properties (*e.g.*, `$obj->p1` and `$obj->p2`). To address this problem, TChecker additionally uses the property name (*e.g.*, `p1` and `p2`) to identify an object property.

Built-in Functions. TChecker models the following built-in functions. First, it models string functions (*e.g.*, `substr()`) to resolve the variable function names and file names. Second, it identifies callbacks in built-in functions (*e.g.*, `array_walk()`) and resolves the callbacks. TChecker also analyzes the effects of built-in functions on taint propagation. For instance, the return value of `substr()` is tainted if its first argument is tainted.

Conditional Statements and Loops. TChecker merges the taints from all branches of a conditional statement following the common practices [26, 28]. This would introduce over-tainting. One solution to tackle this problem is modeling the constraints of a conditional statement [18]. Since constraint solving is orthogonal to our design, we leave it as a further work. TChecker treats loop statements as conditional statements and unrolls them only once.

Arrays. Arrays in PHP are hash tables that map keys to values. The array keys and values can both be variables, *e.g.*, `$arr[$key] = $val`. The array structure can also be dynamically changed using the built-in functions such as `array_push()`. Modeling the dynamic array structure and dynamic array access is known to be hard [26]. The common practice is to only model the array elements with concrete key values [35, 36]. We adopt this approach in our implementation. Additionally, we taint array elements in two cases. First, we taint an array element (*e.g.*, `$arr['a'][$var]`) if it is an item of another tainted array element (*e.g.*, `$arr['a']`). Second, since an array element indexed by a variable key might be any array element, we taint an array element (*e.g.*, `$arr[$i]`) if it could be another tainted array element (*e.g.*, `$arr['a']`). The reason we taint `$a[$i]` without actually analyzing the value of `$i` is that we observe the array elements are usually iteratively accessed in a loop structure. The index `$i` can be any key value in many cases.

6 EVALUATION

In this section, we evaluate the effectiveness of TChecker in detecting taint-style vulnerabilities. We apply TChecker to a diverse set of PHP applications and also compare it to existing taint analysis tools. The rest of this section introduces the dataset and setup (§6.1), then presents the detection results (§6.2) and detection performance (§6.3), and last shows several case studies (§6.4).

6.1 Dataset and Setup

We select 17 PHP applications with over 1M total LLoC as our evaluation dataset. They are listed in the first column of Table 1. Our evaluation dataset consists of two categories of applications. First, it includes popular and well-known real-world PHP applications such as WordPress, MediaWiki, and Joomla. Using the same standard in [36], we choose 9 PHP applications that either have over 0.1% market share [13] or have over 2K stars on GitHub, and they are tagged with superscript ‡ in Table 1. Second, it includes relatively less popular applications. We thoroughly include 8 such applications that were evaluated by prior works [18, 26, 28].

We compare TChecker with the latest open-source versions of PHPJoern[21] and RIPS [1], which are two advanced static analysis tools on vulnerability detection in PHP applications. We are unable to compare with other taint analysis tools because either they are not open-sourced [28, 46] or the code is incomplete [6]. For a fair comparison, we apply the same definitions of sources, sinks, and sanitization rules to TChecker, RIPS and PHPJoern.

6.2 Detection Results

We apply TChecker to detect SQLi, XSS, and DoS vulnerabilities in the dataset and present the results in Table 1. We use the superscript T to denote the results of TChecker. In total, TChecker reported 284 vulnerabilities (including false positives).

To understand the true positives and false positives, we conduct a systematic validation for each reported vulnerability. Though Navex has demonstrated the feasibility of automated vulnerability validation using symbolic execution [18], unfortunately, we cannot apply it in this work because of its incomplete released code. We thus validate the vulnerabilities manually. Specifically, we have designed TChecker to dump the complete call stack for each reported vulnerability. We first apply code review to filter out obvious false positives, *i.e.*, the vulnerabilities that are reported due to explicit flaws in detection tools. For instance, we could quickly identify the vulnerabilities found in the test files as false positives. Afterward, we attempt to construct exploits for the reported vulnerabilities. For vulnerabilities in old versions of applications, their proof-of-concept (PoC) exploits are usually publicly available. Therefore, we simply search online for the sink locations of reported vulnerabilities, and then exploit the applications with the existing PoC exploits. If we could not find a PoC directly, we manually analyze the call stack and related data flows to construct exploits ourselves. We leverage the similar characteristics shared by several vulnerabilities to ease the effort to develop exploits for the vulnerabilities. For instance, we could usually use the same payload to perform XSS attacks on multiple web pages of an application. Finally, we attribute a reported vulnerability to a false positive if it cannot be exploited or is considered as a legitimate feature of an application. The manual code review and exploitation experiment were conducted by one of the authors with about 50 hours.

We use TP and FP in Table 1 to represent the numbers of true positives and false positives. Overall, TChecker detected 131 true positive vulnerabilities and 153 false positives. Its precision is 46.1%, which is very close to the highest (47.9% of PHPJoern) among the three tools. Perhaps unsurprisingly, most (about 80%) vulnerabilities TChecker found were from non-popular applications. Nevertheless, TChecker is capable of finding vulnerabilities in well known applications like Joomla due to its precise inter-procedural analysis.

Statically detecting vulnerabilities in complex PHP programs requires the support of multiple language features. However, in our current design, it would be difficult to separate TChecker's individual components, which have strong dependencies with the others. For instance, the call graph construction and the inter-procedural data flow analysis help each other. We thus present TChecker as a whole and do not separately evaluate each component (*e.g.*, call graph). In a case study, we found that TChecker was able to correctly analyze a few complex features including dynamic includes and object-oriented features, and tens of non-static method calls to reveal one SQL injection vulnerability in Joomla. This shows that the good detection performance of TChecker results from its strong support of modern PHP features.

We finally evaluate TChecker's ability to find new vulnerabilities. We consider a true positive as new (*i.e.*, NP in Table 1) if 1) we cannot find its bug report on the Internet; and 2) it exists in the latest version of the application. TChecker reported 18 new vulnerabilities—7 SQLi, 10 XSS, and 1 DoS vulnerabilities. The vulnerabilities could severely affect the applications and their users. In our experiment, all the 7 SQLi vulnerabilities allow the unprivileged users to manipulate the database; 5 XSS vulnerabilities allow the unprivileged users to steal session cookies; and the DoS vulnerability allows the attackers to *directly* control the iteration times of a loop statement. Fortunately, only one new vulnerability was found in a popular application (*i.e.*, osCommerce2 (2.3.4.1)). This vulnerability could allow the unprivileged users to hijack the user's session and it has been promptly fixed because of our report. We have responsibly disclosed our new findings to the relevant vendors. At the time of writing, 5 vulnerabilities, including 2 new CVEs[2], have been acknowledged or patched.

6.2.1 Comparison with Related Works. In this section, we compare TChecker with related works and discuss how the different design choices affect the detection results.

In Table 1, we use the superscripts J and R to denote the results of PHPJoern and RIPS, respectively. UP denotes the vulnerabilities found by only TChecker and UN denotes the ones found by the other tools but not by TChecker. UP and UN could indicate how TChecker and other static analysis tools could potentially complement each other. In general, TChecker significantly outperformed PHPJoern and RIPS, and detected all the vulnerabilities reported by them (*i.e.*, UN = 0) and 50 vulnerabilities (*i.e.*, UP) they did not find. We present more detailed characterization below.

Comparison with RIPS. TChecker detected 50 vulnerabilities RIPS failed to find. In 27 out of the 50 vulnerabilities, the taints were

[2]CVE-2022-35212, CVE-2022-35213.

Table 1: Evaluation results of vulnerability detection. TP denotes true positives; FP denotes false positives; The superscripts T, J, R denote the results of TChecker, PHPJoern, and RIPS. UP denotes the number of vulnerabilities found by TChecker but not PHPJoern and RIPS. UN denotes the number of vulnerabilities found by PHPJoern and RIPS but not TChecker. NP denotes the number of new vulnerabilities TChecker finds. A dash means we failed to apply a tool to the application. ‡ denotes popular and well-known real-world PHP applications in the first category; other untagged applications are in the second category.

Application	LLoC	TP^T	FP^T	$Time^T$	TP^J	FP^J	$Time^J$	TP^R	FP^R	$Time^R$	UP	UN	NP
MediaWiki (1.36.2)‡	178,616	0	8	91 m	0	6	11 m	0	54	4 m	0	0	0
Collabtive (3.1)	71,705	1	2	46 m	1	0	3 m	1	19	1 m	0	0	0
Ecommerce-CodeIgniter-Bootstrap	25,421	14	0	5 m	8	0	1 m	10	10	1 m	4	0	4
WeBid (1.2.2)	19,595	18	10	42 m	1	0	2 m	1	0	5 m	17	0	2
WordPress (5.4.8)‡	98,099	0	7	68 m	0	0	6 m	-	-	-	0	0	0
CPG (1.6.12)	30,553	1	1	8 m	1	0	3 m	1	1	6 m	0	0	0
Webchess (0.9)	1,569	27	9	3 m	27	9	3 m	27	12	2 m	0	0	0
Joomla (3.7.0)‡	134,692	3	7	86 m	0	0	13 m	0	1	5 m	3	0	0
Joomla (3.10.3)‡	156,172	0	7	93 m	0	0	13 m	0	1	5 m	0	0	0
osCommerce2 (2.3.4.1)‡	27,284	15	12	12 m	0	10	4 m	4	156	3 m	11	0	1
phpBB (3.3.3)‡	194,312	0	3	72 m	0	1	3 m	0	9	7 m	0	0	0
stock-management	21,466	7	0	3 m	0	0	2 m	0	0	0 m	7	0	7
PHPLiteAdmin (1.9.8.2)‡	3,288	1	0	1 m	1	0	1 m	1	41	1 m	0	0	0
Zen-Cart (1.3.8)‡	46,804	4	45	55 m	0	9	4 m	3	519	2 m	1	0	0
Zen-Cart (1.5.5)‡	51,099	1	42	57 m	0	42	9 m	1	529	3 m	0	0	0
Codiad (2.8.4)	2,275	33	0	9 m	32	0	3 m	32	3	1 m	1	0	0
monstra (3.0.4)	6,827	6	0	10 m	0	0	4 m	0	1	2 m	6	0	3
Total	1,069,777	131	153	661 m	71	77	85 m	81	1356	47 m	50	0	18

propagated through the *complex object properties* before reaching the sinks. The object property support of TChecker allows it to detect them, whereas RIPS could not. RIPS could not identify the other 23 vulnerabilities because of implementation issues, *e.g.*, it failed to resolve some assignment-like statements and the dynamic includes when propagating taints.

TChecker has a much higher precision (46.1%) compared with RIPS (about 6%). Many vulnerabilities reported by RIPS were false positives, mainly because it adopts an over-tainting strategy when handling function calls. Unlike TChecker that performs a precise context-sensitive inter-procedural data flow analysis, RIPS simply considers that the return value is tainted if one of the arguments is tainted without analyzing the actual behaviors of the callee function. Such a strategy might label a return value as tainted even if all data flows to it are sanitized. As a result, many false positives reported by RIPS were found in applications (*e.g.*, Zen-Cart) that extensively used tainted arguments in function calls. Besides, RIPS did not analyze the class type of receivers. It over-approximated calling relationships by regarding all the methods whose names matched with the called method names as call targets, resulting in tens of false positives. Note it is very time-consuming to manually validate if a call edge is valid. This further demonstrates the need and benefit of building a precise call graph.

Comparison with PHPJoern. PHPJoern adopts a different strategy from RIPS. Instead of using an over-tainting strategy, PHPJoern selectively propagates taints—it only tracks taints in local variables and function parameters. As a result, PHPJoern reported the fewest false positives (77) among the three tools and had the highest precision (47.9%). Most false positives in PHPJoern were caused by the inherent limitations of static analysis. For example, PHPJoern reported vulnerabilities in dead code.

Nevertheless, PHPJoern reported also much fewer true positives (TPs), *i.e.*, it found 60 fewer TPs than TChecker and 10 fewer TPs than RIPS. PHPJoern exhibits similar limitations as RIPS, *e.g.*, it does not support object properties. Additionally, it supports only method calls with unique names. For instance, several applications declared multiple `query()` functions in their database modules. These `query()` functions were invoked through one method call `$db->query()`. PHPJoern could not find the correct call targets of the function call `$db->query()` and thus failed to detect SQLi vulnerabilities in these `query()` functions.

6.2.2 False Positives. TChecker reported 153 false positives. We attribute its false positives to four aspects.

Incomplete Sanitizers. TChecker does not model all sanitizers. For example, MySQLi query supports typecasting by invoking the `CAST()` function. TChecker does not analyze the semantics of arguments in query function calls, which causes several false positives.

Intended Features. Some applications allow the administrator users to send arbitrary inputs to the sink functions. For example, the Zen-Cart implements a debug feature that executes arbitrary MySQLi query statements provided by the administrators. Our communication with the developers helped us learn that 9 false positives were actually the intended application features.

Implementation Issues. Some false positives were introduced because of implementation issues of TChecker. Specifically, TChecker over-approximates arrays—it taints an array when one of the array elements is tainted. This introduces a few false positives. TChecker does not solve the constraints of conditional statements but merges the taint information from all branches, which leads to another tens of false positives.

Dead Code. TChecker also reports vulnerabilities in dead code (*e.g.*, the unused components of third-party libraries and test files). The vulnerabilities in dead code are not exploitable and lead to over half of the total false positives of TChecker. Note the vulnerabilities in third-party libraries might lead to vulnerabilities in other PHP programs [11]. However, we did not report these vulnerabilities to third-party vendors as we cannot exploit them in our experiments.

6.2.3 False Negatives. Although TChecker identifies all the vulnerabilities RIPS and PHPJoern detect, it still has false negatives in reporting all the taint-style vulnerabilities. For instance, the `mysql_real_escape_string()` adds preceding backslash to escape special characters in a string to be used in an SQL query. However, if the user input is not embedded into quotes within an SQL query, such a sanitization is insufficient and the application becomes vulnerable. TChecker does not sufficiently model the sanitization code and cleans taints in such (incompletely) sanitized variables, thus could have false negatives.

To understand the false negatives of TChecker, we collect from the CVE database[3] all the known vulnerabilities in our evaluated versions of applications to obtain the ground truth. Note that this is an underestimation of false negatives as the CVE database might not contain all known vulnerabilities of an application. We also exclude second-order vulnerabilities as they are outside the scope of TChecker. Overall, we found that TChecker was able to detect all known vulnerabilities in 7 applications including Webchess 0.9 and osCommerce2 2.3.4.1. Yet it also had false negatives in several other applications. For instance, it failed to detect 6 known vulnerabilities in Joomla 3.7.0. We manually analyzed a few vulnerabilities TChecker failed to detect and found that they were caused by insufficient sanitization. This indicated that by better modeling the (custom) sanitization code in PHP programs [34], we could further reduce the false negatives. As this is orthogonal to this work, we leave it as a future work.

6.3 Performance

In this subsection, we discuss the performance of each detection tool. We list the analysis time for each tool in columns Time^T, Time^J, and Time^R. The analysis time of TChecker includes the time spent on call graph construction and taint analysis. In particular, TChecker used 661 minutes to finish all the analysis. Compared with RIPS and PHPJoern, TChecker spent more time on analyzing a PHP application because of its comprehensive data flow analysis. However, given TChecker could report more true positive vulnerabilities with a relatively good precision, we believe that the additional analysis time remains acceptable and manageable.

6.4 Case Study

We further showcase the effectiveness of TChecker with two new vulnerabilities it detected in the current version of Stock-Management-System [9] and osCommerce2 [7].

Stock-Management-System. As shown in Listing 2, the user input `$_REQUEST` is saved into an object property in line 3 and is used as an argument of function call in line 5. The called function `ProductController::insert()` invokes the `productName()` function to check the validity of the argument `$data`, which is tainted. However, the `productName()` function does not sanitize `$data` against SQLi, although it invokes the custom `security()` function that calls a few sanitizers such as `htmlspecialchars()`. In line 8, the tainted variable `$data` is then used as an argument of the call to the `Product::insert()` function, which finally constructs a SQL query string using `$data` and executes the unsafe SQL query in line 22.

```
1  <?php
2  //bootstrap/app.php
3  self::$data = $_REQUEST;
4  //routes/ApiRoutes.php
5  ProductController::insert(self::$data);
6  //controllers/ProductController.php
7  if(ValidateParams::productName($data['name'])) {
8    $product->insert($data);
9  }
10 //helpers/ValidateParams.php
11 static function productName($name) {
12   $name = self::security($name);
13   ...
14   // return if $name is secure
15 }
16 private static function security($data) {
17   $data = trim($data);
18   $data = stripslashes($data);
19   $data = htmlspecialchars($data);
20   return $data;
21 }
22 //model/Product.php
23 function insert($data) {
24   $sql = "INSERT INTO products (`name`) VALUES ('" .
       $data['name'] . "')";
25    if ($this->conn->query($sql) === TRUE) {
26         ...
27    }
28 }
```

Listing 2: An SQLi vulnerability in Stock-Management-System. The code is simplified for demonstration purpose.

To detect such a vulnerability, TChecker tracks taints in object property `self::$data` and infers the call targets of the method call `$product->insert()` in line 8. Besides this application, we find several applications, including Joomla, save the user inputs into object properties before processing them. The existing tools fail to detect vulnerabilities in these applications as they cannot track taints through object properties. The ambiguous method names (*e.g.*, `inject()`) also hinder the detection performance of RIPS and PHPJoern.

osCommerce2. Listing 3 shows an example of XSS vulnerability TChecker found in osCommerce2 [7]. The application executes SQL statement in line 2 or echo the `$query` parameter back to users if function `mysqli_query()` returns error. The tainted `$query` parameter is sent from users to the `mysqli_query()` function and sanitized against SQL injection attacks. However, it is not sanitized against XSS vulnerabilities. If the unprivileged user sends an invalid `query` parameter that causes the function `mysqli()` to return error, the output function `die()` in line 6 will echo the tainted `$query`, leading to an XSS attack.

[3] CVE: https://cve.mitre.org/.

```
1 <?php
2 $result = mysqli_query($$link, $query) or tep_db_error($query, mysqli_errno($$link), mysqli_error($$link));
3
4 // includes/functions/database.php
5 function tep_db_error($query, $errno, $error) {
6   die('<font ...' . $query . '... </font>');
7 }
8
9 // admin/includes/functions/database.php
10 function tep_db_error($query, $errno, $error) {
11   die('<font ...' . $query . '... </font>');
12 }
```

Listing 3: An XSS vulnerability in oscommerce2 v2.3.4

The function `tep_db_error()` is declared twice in the osCommerce2 application. PHPJoern fails to identify the target function of function call in line 2 and misses this vulnerability. RIPS instead connects the function call to all the declared `tep_db_error()` functions. Further, it does not step into any target function but reports XSS vulnerability because of the tainted argument `$query`. Therefore, RIPS reports many false positives–it reports the same vulnerability even in the patched version in which `$query` is sanitized. TChecker determines the target function based on the calling contexts. It finds the `tep_db_error()` function in line 5 which is included in the current scope when propagating taints through the call sites in line 2.

7 DISCUSSION

In this section, we discuss the limitations of TChecker and the possible future works.

7.1 PHP Static Analysis

TChecker is designed to model the complex features in PHP applications. However, it still fails to fully support a few PHP features, including the dynamic arrays, the variable functions, *etc.* In our evaluation, TChecker suffers from a few false positives for it. We emphasize the challenges in statically modeling these PHP features. In the future, new algorithms and implementations might be developed upon TChecker to further improve the static techniques. Also, many works simplify static analysis with some assumptions. It would be interesting to study the effects of the simplified modeling on different features given different scenarios. In this work, we also demonstrate the importance of context-sensitive inter-procedural analysis on detecting taint-style vulnerabilities in PHP applications.

7.2 Applications of Call Graph

In this work, we demonstrate that a precise call graph could help find more taint-style vulnerabilities in PHP applications. In addition to the application in taint analysis, call graph has been used in many security scenarios. Saphire resolves dynamic includes to identify and filters the unused dangerous system calls [23]. Varis builds call graph to provide IDE extensions that allow the user to navigate between the caller functions and callee functions [41]. Torpedo models database statements of PHP applications to detect second-order vulnerabilities whereas call graph is the fundamental component of its static analysis [42]. Note that TChecker currently does not consider second-order vulnerabilities as analyzing database statements is orthogonal to this work. Nevertheless, we believe that the techniques we propose could help future research on PHP applications. For instance, it would be good to further implement database statement modeling upon TChecker for detecting second-order vulnerabilities. The static analysis can further work with dynamic approaches as demonstrated in [18]

7.3 Automated Validation

Static analysis usually reports false positives in vulnerability detection. Navex has demonstrated the feasibility of automated bug validation using symbolic execution [18]. Unfortunately, we cannot apply it in this work because of its incomplete released code. In the future, we plan to implement symbolic execution to validate and exploit the vulnerabilities found by TChecker. Automatically validating bugs found by TChecker is hard. In addition to implementing the symbolic execution algorithm on PHP applications, we need to address a few additional challenges. For instance, the call site in TChecker might have multiple call targets while the conventional symbolic engines consider only one target function for a call site [18, 37]. To address this issue, one solution is to model the path constraint of each target function and then feed the inputs satisfying different path constraints to the program. We leave it to future work.

8 RELATED WORK

In this section, we discuss TChecker with the related works.

8.1 Vulnerability Detection in PHP Applications

The detection of PHP vulnerabilities has drawn significant attention over the past years. Prior studies on static security analysis focus on data-flow (taint) analysis to identify cross-site scripting [15, 19, 21, 26, 39, 40], SQL injection [19, 21, 26, 40, 49], denial-of-service [42, 46], and other types of vulnerabilities [27, 28, 36, 40]. TChecker proposes several generic techniques such as precise inter-procedural data-flow analysis and object property handling, aiming to improve the state-of-the-art static analysis in PHP. The techniques are shown to bring many benefits in terms of precision and scalability. Although we currently implement them for PHP, we believe that they can be extended to other languages to further help the whole community in the field of static program analysis.

Orthogonal to our research, another line of studies utilizes dynamic scanning methods [16, 29–31, 43, 47] to detect vulnerabilities in PHP. Such tools drive the concrete execution of PHP code and thus can literally support the dynamic PHP language features. However, the key challenges like the inter-state dependencies [29] make these dynamic scanning tools less effective to reach deep code. Hybrid approaches like Navex [18] and Chainsaw [17] apply static symbolic execution to model the inter-state dependencies, which can produce seeds to guide the dynamic scanner to find vulnerabilities hidden deeply in code. Similarly, our refinement of call graphs can help improve the static symbolic execution.

8.2 Call Graph Analysis

Call graph analysis is a foundational program analysis technique used for various tasks such as vulnerability detection, control-flow integrity [20], *etc.* Besides PHP, call graph analysis has been an important topic for other programming languages [32, 38, 44, 50–52]. Yamaguchi *et al.* integrate call graph into code property graph to model common vulnerabilities in C/C++ programs [50] and further extend it with an inter-procedural analysis [51]. Most recently, Lu and Hu refine indirect-call targets with multi-layer type analysis for C/C++ programs based on structure types and pointer analysis [38]. However, unlike other languages, PHP, as a weakly-typed and dynamically-typed language, lacks necessary variable type information, making it even more challenging to perform call graph analysis. TChecker addresses those PHP-specific features and challenges, and achieves a precise call graph analysis.

9 CONCLUSION

Vulnerability detection in PHP is essential to web security. In this paper, we identified several fundamental limitations that hinder prior static approaches from a precise and practical analysis for detecting taint-style vulnerabilities. We presented TChecker, a precise context-sensitive inter-procedural static taint analysis system to detect taint-style vulnerabilities in PHP applications. TChecker incrementally builds a precise call graph by iteratively performing an inter-procedural data flow analysis. We also take a considerable amount of engineering effort to support a few complex object-oriented features in modern PHP applications. With these techniques, TChecker successfully identified 18 previously unknown vulnerabilities. The comparison with the related detection tools demonstrates that TChecker outperformed them with more vulnerabilities detected. The impressive evaluation results show that TChecker is highly effective in detecting taint-style vulnerabilities in modern PHP applications. We believe that the techniques can shed light on future research of PHP program analysis.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments. We also thank Yanting Chi for helping us validate the reported vulnerabilities. The work described in this paper was partly supported by a grant from the Research Grants Council of the Hong Kong SAR, China (Project No.: CUHK 14210219).

REFERENCES

[1] 2017. RIPS scanner. https://sourceforge.net/projects/rips-scanner//.

[2] 2020. SQL injection vulnerability in Joomla. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35613.

[3] 2020. XSS in Drupal Core. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-13688.

[4] 2020. XSS in OpenCart. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15478.

[5] 2021. Basic Coding Standard. https://www.php-fig.org/psr/psr-1/.

[6] 2021. Incomplete Navex source code. https://github.com/aalhuz/navex/issues/6.

[7] 2021. osCommerce Online Merchant. https://www.oscommerce.com.

[8] 2021. php-ast. https://github.com/nikic/php-ast.

[9] 2021. Stock-Management-System: An Introductory Stock Management System built on PHP, jQuery with AJAX in MVC pattern. https://github.com/haxxorsid/stock-management-system.

[10] 2021. XSS in WordPress. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-39202.

[11] 2022. CWP CentOS Web Panel – preauth RCE. https://octagon.net/blog/2022/01/22/cve-2021-45467-cwp-centos-web-panel-preauth-rce/.

[12] 2022. Pulse. https://www.pulse.codacy.com/?utm_source=codacy&utm_medium=referral&utm_campaign=codacy_link&utm_content=nav_dropdown.

[13] 2022. Usage statistics of content management systems. https://w3techs.com/technologies/overview/content_management.

[14] 2023. RIPS-tech. https://blog.sonarsource.com/.

[15] Wasef Abdalla, Zarul Marashdih, Zaaba Fitri, and Suwais Khaled. 2018. Cross Site Scripting: Investigations in PHP Web Application. In *International Conference on Promising Electronic Technologies*.

[16] Doupé Adam, Cavedon Ludovico, Kruegel Christopher, Vigna Giovanni, and Barbara Santa. 2014. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. Scottsdale, Arizona.

[17] Abeer Alhuzali, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. 2016. Chainsaw: Chained automated workflow-based exploit generation. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.

[18] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and VN Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.

[19] Algaith Areej, Nunes Paulo, Jose Fonseca, Gashi Ilir, and Vieira Marco. 2018. Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools. In *European Dependable Computing Conference*.

[20] Amin Azad Babak, Laperdrix Pierre, and Nikiforakis Nick. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.

[21] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. 2017. Efficient and flexible discovery of php application vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P)*. Paris, France.

[22] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. 2010. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.

[23] Alexander Bulekov and Manuel Egele. 2020. Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[24] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2020. SELECTIVETAINT: Efficient data flow tracking with static binary rewriting.. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.

[25] Penny Crosman. 2015. Banks Lose Up to $100K/Hour to Shorter, More Intense DDoS Attacks. https://www.americanbanker.com/news/banks-lose-up-to-100k-hour-to-shorter-more-intense-ddos-attacks.

[26] Johannes Dahse and Thorsten Holz. 2014. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[27] Johannes Dahse and Thorsten Holz. 2014. Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.

[28] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code Reuse Attacks in PHP : Automated POP Chain Generation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. Scottsdale, Arizona.

[29] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. 2021. Black Widow: Blackbox Data-driven Web Scanning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.

[30] Duchene Fabien, Rawat Sanjay, Richier Jean-Luc, and Groz Roland. 2013. Ligre: Reverse-engineering of control and data flow models for black-box xss detection. In *Working Conference on Reverse Engineering*.

[31] Duchene Fabien, Rawat Sanjay, Richier Jean-Luc, and Groz Roland. 2014. Kameleonfuzz: evolutionary fuzzing for black-box xss detection. In *Proceedings of the ACM conference on Data and application security and privacy*.

[32] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. In *Proceedings of the 12th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Atlanta, Georgia.

[33] Mark Hills1, Paul Klint, and Jurgen Vinju. 2013. An empirical study of PHP feature usage: a static analysis perspective. In *Proceedings of the 22nd International Symposium on Software Testing and Analysis (ISSTA)*. Lugano, Switzerland.

[34] Hough Katherine, Welearegai Gebrehiwet, Hammer Christian, and Bell Jonathan. 2020. Revealing Injection Vulnerabilities by Leveraging Existing Tests. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. Seoul, Korea.

[35] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. 2010. Phantm: PHP analyzer for type mismatch. In *Proceedings of the 18th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Santa Fe, NM.

[36] Penghui Li and Wei Meng. 2021. LChecker: Detecting Loose Comparison Bugs in PHP. In *Proceedings of the Web Conference (WWW)*. Ljubljana, Slovenia.

[37] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. 2020. On the feasibility of automated built-in function modeling for PHP symbolic execution. In *Proceedings of the Web Conference (WWW)*. Taipei, Taiwan.
[38] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*. London, UK.
[39] Kumar Mukesh, Mahesh Gupta, Govil Chandra, and Singh Girdhari. 2015. Predicting Cross-Site Scripting (XSS) Security Vulnerabilities in Web Applications. In *International Joint Conference on Computer Science and Software Engineering*.
[40] Jovanovic Nenad, Kruegel Christopher, and Kirda Engin. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (Oakland)*. Oakland, CA.
[41] Hung Viet Nguyen, Christian Kastner, and Tien N. Nguyen. 2015. Varis: IDE Support for Embedded Client Code in PHP Web Applications. In *Proceedings of the 37th International Conference on Software Engineering (ICSE)*. Florence, Italy.
[42] Olivo Oswaldo, Dillig Isil, and Lin Calvin. 2015. Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. Denver, Colorado.
[43] Giancarlo Pellegrino, Constantin Tschürtz, Eric Bodden, and Christian Rossow. 2015. jäk: Using dynamic analysis to crawl and test modern web applications. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Kyoto, Japan.
[44] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call graph construction for java libraries. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Seattle, WA.
[45] AAG IT Services. 2019. How often do Cyber Attacks occur? https://aag-it.com/how-often-do-cyber-attacks-occur/.
[46] Sooel Son and Vitaly Shmatikov. 2011. SAFERPHP: Finding semantic vulnerabilities in PHP applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*.
[47] Felmetsger Viktoria, Cavedon Ludovico, Kruegel Christopher, and Vigna Giovanni. 2010. Toward automated detection of logic vulnerabilities in web applications.. In *Proceedings of the 19th USENIX Security Symposium (Security)*. Washington, DC.
[48] W3Techs. 2021. Usage statistics of PHP for websites. https://w3techs.com/technologies/details/pl-php.
[49] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. San Diego, CA.
[50] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[51] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[52] Chao Zhang, Dawn Song, Scott A Carr, Mathias Payer, Tongxin Li, Yu Ding, and Chengyu Song. 2016. VTrust: Regaining Trust on Virtual Calls.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.