PFORTIFIER: Mitigating PHP Object Injection through Automatic Patch Generation

Bo Pang*
School of Cyber Science and Engineering, Sichuan University
hach.chp@gmail.com

Yiheng Zhang*
School of Cyber Science and Engineering, Sichuan University
x1angf3ngwan@gmail.com

Mingzhe Gao
Alibaba Cloud Computing
mzgao@njnet.edu.cn

Junzhe Zhang
National University of Singapore
junzhe.zhang@u.nus.edu

Ligeng Chen
Honor Device Co., Ltd
Nanjing University
chenlg@smail.nju.edu.cn

Mingxue Zhang†
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
mxzhang97@zju.edu.cn

Gang Liang†
School of Cyber Science and Engineering, Sichuan University
lianggang@scu.edu.cn

***Abstract*—PHP Object Injection (POI) vulnerabilities enable unexpected execution of class methods in PHP applications, resulting in various attacks. In the meanwhile, designing effective patches for POI vulnerabilities demands substantial engineering efforts. Existing research mostly focused on the detection of POI gadget chains, whereas the automatic patch generation remains an under-explored problem.**

In this work, we empirically study known gadget chains, and discover that adversaries usually construct gadget chains by diverging the execution to paths that developers never considered. The methods that get unexpectedly jump into (*i.e.*, executed) are referred to as possible methods (PM). Based on the observation, we propose PFORTIFIER, a framework for automatic POI patch generation. PFORTIFIER operates in two stages: *(i)* the gadget chain detection phase, in which PFORTIFIER simulates the execution of PHP applications, and detects gadget chains that pass attacker controlled objects to dangerous sinks, and *(ii)* the patch generation phase, in which PFORTIFIER automatically generates POI patches by restricting PM jumps detected in the first phase. We evaluate PFORTIFIER on 31 PHP applications and frameworks. The experiment results demonstrate the effectiveness of PFORTIFIER: it generates precise patches for 52.53% of gadget chains, and suggests potential patches for 45.45% chains, resulting in a total chain coverage of 97.98%.

1. Introduction

With a dominant market share of 77.4% among all websites [1], PHP solidifies its position as the leading server-side programming language for web services. Vulnerabilities in PHP-based applications thus become attractive to adversaries. One typical example is the PHP Object Injection (POI) vulnerability, which allows attackers to execute class methods in an unexpected way to developers, leading to various types of attacks, *e.g.*, SQL injection, Denial of Service (DoS), *etc* [2]. The unexpected execution paths are called gadget chains. In particular, according to the statistics in PHPGGC [3], more than 75% of gadget chains lead to remote code execution, compromising the reliability and security of PHP applications. Hence, curbing the impact of POI on PHP applications remains a pressing issue in need of resolution.

One common method to mitigate vulnerabilities is patch generation. There have been numerous studies [4]–[16] on patching various vulnerabilities. However, these approaches cannot be readily applied for patching POI vulnerabilities due to the unpredictable nature of gadget chain execution and the difficulty in designing effective sanitization. While several works have focused on detecting gadget chains [17]–[21], they lack a patch generation method specifically tailored for PHP gadget chains.

Due to the complexity in exploiting POI vulnerabilities, developers usually opt to manually design the patches, which requires significant engineering efforts. Automatically generating patches for the vulnerabilities, on the other hand, tends to be a challenging problem. Specifically, in this work, we face the following challenges in automatic POI patch generation: *C1: Infinite combinations.* The unpredictable execution of gadget chains may involve arbitrary combinations of class methods. It is difficult to apply sanitizers on all possible execution paths. *C2: Incomplete and inefficient chain detection.* Current gadget chain detection methods suffer from inadequate coverage and analysis efficiency, restricting the comprehensiveness and effectiveness of the patching mechanism. Improving chain coverage and efficiency is also difficult. *C3: Impact minimization.* We aim to patch POI vulnerabilities while maintaining normal functionalities in

∗. These authors contributed equally to this work.

†. Corresponding authors. Mingxue Zhang is also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou 310051, China.

the patched applications, which is another challenge to be addressed.

To address the above challenges, we propose a framework, PFORTIFIER, for automatically generating gadget chain patches. It is worth noting that without the internal knowledge, it is very difficult to design one exact patch that does not affect the intended functionalities for each vulnerability. For instance, a method can be called unexpectedly by an attacker through magic methods, yet it is also possible that the magic methods are intended to be invoked. Therefore, instead of suggesting exact patches for all vulnerability, PFORTIFIER aims to provide developers with **a minimal number of potential patches** while **preserving the normal code functionalities** in the best effort manner. Although PFORTIFIER does not always generate the exact patches, automatic generation of minimal number of patches would still significantly ease the auditing burden on developers. Specifically, to address *C1*, we observe that adversaries use PM jumps to divert code execution to unexpected paths. Based on this, PFORTIFIER patches the gadget chains by limiting PM jumps according to 8 pre-defined heuristic rules. To address *C2*, PFORTIFIER performs a greedy simulation to abstract the code semantics and detect gadget chains. Compared with prior works, such a design achieves a higher chain coverage. PFORTIFIER further caches the PM analysis results to speed up the analysis. Finally, to address *C3*, PFORTIFIER constructs POI patches by adding only one *if* branch to minimize its impact on the application's normal functionality.

We evaluated PFORTIFIER on 31 PHP applications and frameworks. The results suggest that PFORTIFIER performs well in both chain detection and patching. In terms of gadget chain detection, PFORTIFIER outperforms the state-of-the-art with 38.18% higher chain coverage, and detects 56 new chains[1], while achieving a speedup of over 10X. For patch generation, PFORTIFIER successfully patched 52.53% of all detected chains, and generated patch suggestions for the remaining 45.45% of chains, achieving a total chain coverage of 97.98%.

Our primary contributions are summarized as follows:

- **The first POI patch generation method.** We proposed a novel PHP gadget chain patch generation method, and implemented a prototype framework, PFORTIFIER. To the best of our knowledge, we are the first to systematically study the POI patching problem.
- **Fast analysis with high chain coverage.** We instrumented PFORTIFIER with an innovative coverage-and-speed-oriented approach to detecting gadget chains. Compared with existing detection tools, PFORTIFIER improves the chain coverage by 38.18% with a speedup of over 10X.
- **Comprehensive evaluation.** Our thorough evaluation on 31 applications demonstrate the effectiveness of PFORTIFIER in generating POI patches.
- **Availability.** To benefit future studies, we have released the implementation of PFORTIFIER on GitHub[2].

The rest of this paper is organized as follows. Section 2 describes the background about POI vulnerabilities, along with the main challenges involved in patching POI. Section 3 presents an empirical analysis of known PHP gadget chains, which inspires the design of PFORTIFIER in Section 4. Section 5 demonstrates the implementation details. The evaluation results on 31 PHP applications are presented in Section 6. We then discuss the limitations and related works in Section 7 and 8, respectively. Finally, Section 9 concludes the paper.

2. Background

2.1. Magic Methods

One prerequisite for exploiting POI vulnerabilities is to construct a chain of methods, *i.e.*, the gadget chain, and invoke its execution starting from one entry method. To achieve this, one possible way is to leverage PHP magic methods [22]. In this section, we first introduce PHP magic methods.

PHP magic methods are special methods that are automatically triggered in specific scenarios. Classes implementing various PHP magic methods can be used within native PHP statements, allowing them to override the default behavior of the object [22]. For instance, when printing an instance of a user-defined class, the magic method `__toString()` will be automatically invoked to convert the object to a string. The same applies when an object is being concatenated to a string or encoded as a JSON object. This may lead to unpredictable runtime behaviors or security issues, because such methods need not to be explicitly invoked by developers and are capable to alter object properties and program execution states. We list all magic methods relevant to POI vulnerabilities in Table 7 in the Appendix.

2.2. (De)serialization and Object Injection Vulnerabilities

Serialization and deserialization are two fundamental mechanisms in PHP that allow developers to convert objects into a storable format, and vice versa. During serialization, an object is converted into a string that can be stored in a database or transmitted over the network. Deserialization, on the other hand, refers to the process of reconstructing objects from the serialized strings.

It is worth noting that the deserialization process is dynamic, *i.e.*, strings for reconstructing objects are processed at runtime, making it possible for attackers to manipulate serialized data and control the deserialization results [23]. This allows attackers to inject arbitrary PHP objects, which

1. 10 of them have been approved by PHPGGC by the time of writing.

2. https://github.com/HACHp1/PFortifier

in turn trigger a sequence of class methods, forming a gadget chain. Such vulnerabilities are referred to as PHP Object Injection (POI) vulnerabilities [2].

```
1  <?php
2  class PendingBroadcast{
3   public function __destruct(){
4    $this->events->dispatch($this->event);
5   }
6  }
7
8  class Generator{
9   public function __call($method, $attributes){
10   return $this->format($method, $attributes);
11  }
12
13  public function format($format, $arguments = []){
14   return call_user_func_array(
15    $this->getFormatter($format), $arguments);
16  }
17
18  public function getFormatter($format){
19   return $this->formatters[$format];
20  }
21
22  public function __wakeup(){
23   $this->formatters = []; // official patch
24  }
25 }
```

Listing 1. Vulnerable classes and official patch in Laravel 8.6.12.

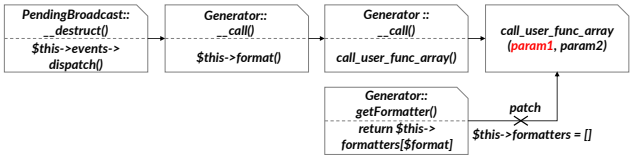

Figure 1. Call stack and official patch of PHPGGC Laravel/RCE1.

Listing 1 presents the gadget classes in Laravel 8.6.12 [24] and the corresponding official patch. We collect the example from PHPGGC [3], which is an open-source library that provides a collection of known PHP gadget chains, along with a payload generation tool. The gadget chain in this example is identified as `Laravel/RCE1` in PHPGGC. Figure 1 shows the associated call stack. Exploiting the PHP Object Injection (POI) vulnerability in `Laravel/RCE1` grants attackers the control over all fields in the root object `this`. Specifically, to construct the gadget chain, the attacker manipulates the `events` field of a `PendingBroadcast` object to a `Generator` object. Then, since the `Generator` class does not implement the `dispatch` method, an unexpected execution path is triggered to call the magic method `Generator::__call()`. The chain ultimately reaches the sink function `call_user_func_array()`, leading to an *arbitrary function execution* vulnerability.

In the chain provided in Listing 1, the attacker utilizes the `formatters` field of the `Generator` class to obtain a controllable method name. This method name is then passed as the first parameter to the `call_user_func_array()` function within the `Generator::format()` method, causing the vulnerability. Given the crucial role of `formatters` field in this gadget chain, Laravel developers patched the vulnerability by overwriting `this->formatters` as an empty array in the `__wakeup()` method. As a result, attackers can not invoke arbitrary methods in Line 14. The example demonstrates that for developers to patch gadget chains, they must have a comprehensive understanding of its construction and execution, which requires significant engineering efforts [21].

3. Insights

We systematically analyzed the known gadget chains in PHPGGC. In this section, we use examples to demonstrate our insights derived from the study. We further demonstrate how our insights inspire the design of PFORTIFIER.

3.1. Possible Method Calls

Before delving into the design of PFORTIFIER, it is imperative to understand how the gadget chains are constructed, and how the POI vulnerabilities can be exploited.

```
1  <?php
2  class LogPrinter{
3   public function __destruct(){ // entry method
4     echo $this->logger->getLog(); // PM call site
5   }
6  }
7
8  class Logger{
9   public function getLog(){ // expected method
10   return $this->log;
11  }
12 }
13
14 class SystemInfo{
15  public function __call(){ // unexpected method
16   $info = system($this->cmd); // sink method: cmd
         injection vul
17   return $info;
18  }
19 }
20
21 $maliciousObject = unserialize($_COOKIE['object']); //
       POI Vul
22 // $maliciousObject = O:10:"SystemInfo":1:{s:3:"cmd";s
       :6:"whoami";
```

Listing 2. An example of POI vulnerability exploitation.

Listing 2 shows an example of POI vulnerability exploitation. Consider a situation where the developer is building an application that prints logs upon application exit. The `LogPrinter` and `Logger` classes provide the function for handling the log information. During the destruction of `LogPrinter`, the `getLog()` method is invoked through `$this->logger` for log printing. However, when the POI vulnerability in Line 21 is exploited, `$this->logger` can be unexpectedly set to a `SystemInfo` object, unbeknownst to the developer. After that, the application attempts to invoke `getLog()` through `SystemInfo`, which implicitly invokes the magic method `SystemInfo::__call()` for handling the undefined method call. As `$this->cmd` is also manipulated by the attacker, the exploit eventually triggers a perilous command injection vulnerability.

The underlying cause of the vulnerability lies in the execution path that deviates from the flow intended by

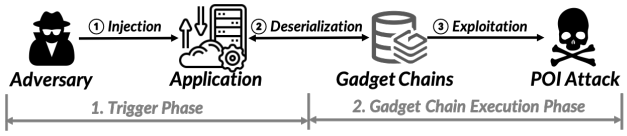

Figure 2. The POI exploitation model.

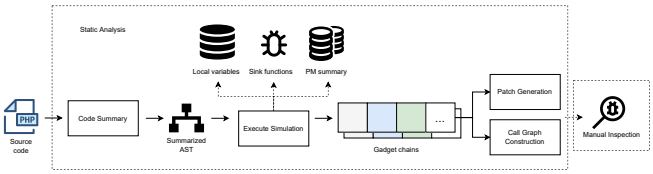

Figure 3. The workflow of PFORTIFIER

the developer. This allows POI exploits to direct the code to unforeseen class methods, triggering the execution of a gadget chain. In the aforementioned example, the command injection vulnerability is triggered by the initial call to `__destruct()` in Line 4. In this work, we refer to the triggering method like `__destruct()` as an *entry method*. We further define class methods that may get triggered through an attacker controllable object as *Possible Methods (PM)* (*i.e.*, `getLog()` and `__call()` in Line 9 and 15), and the security-sensitive methods as *sink methods* (*e.g.*, `system()` in line 16). As gadget chains are connected through PM calls, the POI vulnerabilities can be effectively patched by identifying and restricting the PM calls. In particular, we designate the jumps triggered by the PM call site as *unsafe jumps*.

It is worth noting that in this work, objects like `$this->attr` in Listing 2 are considered "controllable", whereas `$this` is not. Although attackers may direct execution to a specific class and control `$this`, they can not exploit the methods in any other classes through it.

3.2. Patching Strategy

As illustrated above, POI exploitation involves two key stages: payload injection and gadget chain execution (Figure 2). In the payload injection stage, the attacker gains control over a serialized string, allowing him/her to insert a meticulously crafted object through the deserialization method [23]. In the gadget chain execution stage, the attackers invoke the entry method, steering the PHP applications toward security-sensitive sinks. Although the POI vulnerabilities can be mitigated in either stage, we observe that patching the second stage is more feasible and poses minor impact on the normal functionalities. The reasons are as follows.

❶ **Varying ways for deserialization.** In addition to explicitly calling `unserialize()`, there are also other possible ways to trigger deserialization for object injection. For instance, the PHAR protocol in PHP supports autonomous deserialization [25], which implicitly deserializes the metadata of files accessed via the PHAR protocol. Consequently, it becomes difficult to restrict all deserialization operations in the first exploit stage.

❷ **Security awareness.** Web developers prevalently use deserialization for customized functionalities, *e.g.*, reading files and restoring cookies, *etc*. Due to the limited security awareness of secondary developers, the custom code is more likely to be vulnerable thus difficult to be effectively patched by limiting object injections.

❸ **Patching efficiency.** According to PHPGGC, a well-acknowledged tool that provides a collection of 143 known gadget chains, the vulnerable chains and classes mostly reside in the PHP frameworks and libraries. Note that PHPGGC is not framework-exclusive, *e.g.*, it also includes chains in content management systems like WordPress. Therefore, the distribution of vulnerabilities demonstrates the prevalence of gadget chains in frameworks and libraries. As a single framework or library may be adopted by multiple secondary developers, patching POI by restricting gadget chains enhances the security of PHP applications in a more efficient way.

Impact on normal functionalities. Different from other patch frameworks that aim to mitigate abnormal functionalities [10], [12], patching POI vulnerabilities is more challenging, as it is difficult to distinguish exploits from normal behaviors. Therefore, we do not aim to generate precise patches that completely mitigate the security risks while preserving all normal functionalities. This, if not impossible, is very challenging due to the limited internal knowledge of intended functionalities. Instead, we generate patches for obviously abnormal behaviors, and provide patch suggestions in other cases, so that developers can verify the compatibility. In the meanwhile, we attempt to introduce minimal changes to avoid affecting normal functionalities.

3.3. Summary

To summarize our observations, firstly, we find that in POI exploits, the PM calls play a critical role in connecting the gadget chains. Besides, although the POI vulnerabilities can be patched by mitigating object injection risks or restricting the execution of gadget chains, the latter is a more feasible and efficient choice. Therefore, we decided to patch the POI vulnerabilities by detecting gadget chains and restricting the related PM calls. Finally, given the impracticality of devising precise and secure patches for all vulnerabilities, we aim to generate as few simple patches as possible, instead of always constructing one exact patch.

4. Design

4.1. Overview

In this section, we describe PFORTIFIER, a framework for automatic generation of POI patches. The workflow of PFORTIFIER is illustrated in Figure 3, which comprises three analysis steps: code summarization, simulated execution,

and patch generation. The information collected during the first two steps can be utilized to verify the automatically generated patches. More detailed descriptions about the three steps can be found below.

❶ **Code Summarization**: In this step, PFORTIFIER parses the source code into abstract syntax trees (ASTs) and extracts essential information from them, including the fields, methods, class inheritances, interface inheritances, interface implementations, and trait inheritances.

❷ **Simulated Execution**: In this step, PFORTIFIER simulates the execution of ASTs, enabling the identification of potential gadget chains.

❸ **Patch Generation**: In this step, PFORTIFIER attempts to patch the detected gadget chains by restricting the corresponding unsafe jumps caused by PM calls. It iteratively searches for the first patchable jump in the sequence of method calls and adds related checks to prevent unsafe jumps.

As our primary objective is to maximize the patching of gadget chains, PFORTIFIER has been meticulously designed to achieve high chain coverage while ensuring fast analysis speed. In the following sections, we introduce the design of PFORTIFIER in detail.

4.2. Code Summarization

PFORTIFIER initiates the code summarization process by constructing and parsing ASTs, which serve as the foundation for subsequent simulated execution. PFORTIFIER in particular extracts all classes, traits, interfaces, and method information to construct a map from the class name to the ASTs of its methods. Additionally, PFORTIFIER constructs another map from method names to the classes that implement the method, which we call *attr_func_dict*. As methods of the same name can be implemented multiple times in varying ways, PFORTIFIER maintains a list of methods in the map to enable a conservative analysis.

The code summarization process assumes that source code of all classes are available to attackers. We acknowledge that such an assumption may lead to occasional report of vulnerabilities that are very difficult to exploit in a blackbox setting. However, this significantly improves the coverage of gadget chains by analyzing all classes, traits, and interfaces, and is thus suitable for the patch generation purpose.

4.3. Simulated Execution

PFORTIFIER relies on a simulated execution to identify potential gadget chains. It starts the execution from four most common entry methods, *i.e.*, __unserialize(), __destruct(), __wakeup(), and __toString(), following the structure of their ASTs. The entry methods are extracted from known gadget chains in PHPGGC. Nonetheless, PFORTIFIER can also be easily extended to initiate the simulation from other methods. Simultaneously, it marks attacker-controllable objects as tainted, and propagates the taint throughout the simulation process. Once such objects flow into the sink functions or statements in Table 2, a gadget chain is detected.

However, the simulation is not trivial. For instance, as local variables may be controllable to an attacker, we need a way to precisely track their values and point-to relation. The branches caused by PM calls also need to be carefully handled to avoid missing gadget chains. Additionally, we aim to optimize the analysis to improve the efficiency. In the following, we explain the simulation process in detail.

a) Local variable abstraction. One straightforward way to handle local variables is to record their values, namespaces, and use statements under the current scope. Since the main purpose of simulation is to propagate the taint, for variables whose values cannot be determined statically, PFORTIFIER can use a predefined default value, *e.g.*, an empty string for strings. However, as also mentioned in Section 4.2, the simulation may encounter branches, *e.g.*, when processing a PM call site (*e.g.*, `$this->logger->getLog()`) and multiple methods (*e.g.*, `Logger::getLog()` and `SystemInfo::__call()`) could be invoked. To fully capture all possible execution effects, PFORTIFIER employs forced execution of all branches, cloning and merging local variables before and after the branch points. The details can be found below.

Cloning local variables. Upon branches, PFORTIFIER creates a copy of local variables for each branch, which we call *cloning*. For cloning local variables, it is crucial to preserve their original point-to relation, especially when dealing with controllable objects. Nonetheless, deep copy of the local variables may result in object duplication, which in turn mixes up the point-to relation. Prior works [26]–[28] maintain a pointer flow graph to address this issue, which incurs additional overhead. Therefore, PFORTIFIER records with each controllable object its index relative to the root object instead. During a variable cloning, PFORTIFIER locates the corresponding controllable object from the root object through the index chain, and restores the structure of the objects accordingly. For other variables, PFORTIFIER simply records the values.

Merging local variables. After simulating the execution of branches, PFORTIFIER chooses from all branches one copy of local variables as the execution results, and continues the simulation. We call such process as *merging*. For the merging of local variables, PFORTIFIER adopts a greedy strategy, by prioritizing the branch that returns a controllable object. This allows PFORTIFIER to further propagate attacker-controlled objects and thus improve detection coverage of gadget chains. If no such branch is available, PFORTIFIER selects local variables from the first branch.

b) PM summarization. During the simulated execution, whenever a method has multiple implementations with identical names, PFORTIFIER executes all of them for conservativeness. We adopt such a design, because this process aims to achieve high chain coverage instead of being precise. However, this also incurs high computational cost, due to repetitive executions. To improve the analysis speed, PFORTIFIER employs a map called *PM_summary* to cache

the analysis results. The basic assumption is that if the method names and controllabilities of parameters remain consistent, the execution effects would also be the same. PFORTIFIER specifically caches the merged local variables, return values, and gadget chains involved, in all PM call branches. Once PFORTIFIER completes the analysis of the first PM call site, the execution results will be stored in *PM_summary*. Subsequently, when a method with the same name and parameter controllability is encountered, PFORTIFIER retrieves the results directly from the cache, thereby expediting the simulation process.

c) Branch and loop optimization. As previously mentioned, PFORTIFIER forces the execution of all branches to improve chain coverage. However, in case where a controllable object is overwritten as non-controllable in one feasible branch, this design choice could incur false positives. To improve the precision of simulation, PFORTIFIER is equipped with a filtering module that identifies "safe" objects that have been properly sanitized. The rationale is if the controllable object is checked against strict conditions, it is highly possible that developers are aware of the risk that such objects might be controlled by attackers, and the object can be deemed safe.

To this end, PFORTIFIER stores the indexes of sanitized objects in a *condition stack* during the simulation of conditional branches. For nested branches, each branch scope constitutes a stack frame. We define the sanitized objects as the receiver objects on which the checks in Table 1 have been applied. The checks will be explained in detail in Section 4.4. Objects present in the condition stack are considered sanitized and ignored by PFORTIFIER, as they cannot be manipulated by an attacker. Once the conditional node (e.g., if statements) is processed, the corresponding condition is popped out of the stack. Additionally, if the current scope contains a `die()` statement, the corresponding indexes will be promoted one level up in the stack. This ensures proper handling of the statement's impact on the surrounding scope.

To further improve the efficiency of loops, PFORTIFIER performs analysis on each loop only once. Although this could incur inaccuracies if the taint status of an object is changed only in certain loops, we did not observe in practice any such cases. In other words, the inaccuracies are very rare if not inexistent.

d) Early termination. PFORTIFIER sets a threshold *max_pop_length* of the call chain length, and *max_pm_length* for the maximum number of PM calls. If the number of method calls exceeds *max_pop_length*, or the number of PM calls exceeds *max_pm_length*, PFORTIFIER terminates the simulated execution as a tradeoff between the analysis speed and comprehensiveness. In this work, we empirically set the thresholds as 9 and 4, respectively. We demonstrate in Section 6 that PFORTIFIER achieves a high chain coverage with the early termination strategy.

4.4. Patch Generation

As discussed in Section 3, mitigating object injection is very difficult, due to the heavy functional reliance on deserialization in modern PHP applications. Instead, restricting the execution of gadget chains would be more viable. Following this principle, we devised different patching strategies for different categories of gadget chains, by restricting the corresponding PM calls. PFORTIFIER detects PM calls by checking all methods that can be invoked through an attacker-controllable object, *e.g.*, `$this->attr` in Listing 2. When a gadget chain reaching the sinks is identified, a list of PM calls is extracted so that PFORTIFIER can select a suitable one to patch.

Next, we demonstrate the definition and examples of each category of gadget chains as follows.

❶ **Magic method chains.** PMs invoked in this category of gadget chains are magic methods.

❷ **Possible call chains.** These gadget chains do not involve magic method calls but contain other PM call sites.

❸ **Vanilla chains.** This type of gadget chains does not contain any PM call site from the entry method to the sink.

```
1 <?php
2 class C1{
3  public function __destruct(){ //entry
4   $this->attr->vmethod();
5  }
6 }
7
8 class C2{
9  public function __call($methodname, $params){ //sink
10  echo $this->attr;  //xss vulnerability
11  }
12 }
```

Listing 3. An example of a Magic Method chain.

```
1 <?php
2 class C1{
3  public function __destruct(){ //entry
4   $this->attr->vmethod();
5  }
6 }
7
8 class C2{
9  public function vmethod(){ //sink
10  echo $this->attr; //xss vulnerability
11  }
12 }
```

Listing 4. An example of a Possible Call chain.

```
1 <?php
2 class C1{
3  public function __destruct(){ //entry
4   $this->vmethod();
5  }
6  public function vmethod(){ //sink
7   echo $this->attr; //xss vulnerability
8  }
9 }
```

Listing 5. An example of a Vanilla chain.

Listing 3 shows an example of a *magic method* chain. Specifically, upon execution of `__destruct()`, if `$this->attr` is controllable and the class C2 does not implement `vmethod()`, the magic method C2::__call() will be invoked. Eventually, the XSS vulnerability in Line 10 will be triggered.

Listing 4 demonstrates an example of *possible call* chain. Compared with in Listing 3, class C2 now implements the `vmethod`. Therefore, when

`$this->attr->vmethod` is called and `$this->attr` is controllable, `C2::vmethod` will be invoked, triggering the XSS vulnerability.

Listing 5 illustrates an example of a *Vanilla chain*. This chain starts from `C1::__destruct()` and ends at Line 7. It is relatively shorter, as it does not involve any PM call from one attacker-specified class to another.

We have thoroughly analyzed the features of each type of gadget chains. In the following, we explain the patch generation strategy in detail.

Patching magic method chains. These chains are connected by magic method calls, which are only invoked if the methods or attributes accessed is not available or in an unexpected type. Therefore, PFORTIFIER can feasibly restrict the method calls by checking the presence and types of the corresponding methods or attributes. Inspired by the existing pattern-based patch generation methods [10]–[12], PFORTIFIER applies the patching rules in Table 1. For example, for a gadget chain that triggers `__call()` by invoking an non-existent method, the patch adds an if branch, checking whether the method has been defined on the receiver object. If not, a `die()` statement will be executed. We refer to such patches as "restrictive patches". This simple patch is not only effective, but also incurs minimal changes. Furthermore, our patches solely implement conditional checks at specific magic method calls, inherently avoiding direct introduction of new vulnerabilities. For the rest magic methods, *e.g.*, `__construct()`, `__wakeup()`, `__destruct()`, and `__unserialize()`, *etc*, it is infeasible to automatically devise similar patches. For example, when `__clone()` is invoked, it is difficult to determine whether the cloned object is safe or desired. Therefore, PFORTIFIER searches for prior method calls that can be feasibly patched, along the detected gadget chains. If no such calls can be found, PFORTIFIER instead generates patch suggestions for them, which will be explained later.The patching strategy is summarized in Algorithm 1.

Specifically, `jmpNodes` denotes the list of PM call nodes in a gadget chain. To generate a patch, PFORTIFIER iterates through each PM call node, and extracts information such as the method call statement, the method name, and the involved classes (Line 2-4). PFORTIFIER then generates the corresponding patches (Line 5) for the PM call based on the relevant rules. If there is no rule defined for the node, PFORTIFIER attempts to generate the patch for the next node. If a patch is successfully generated, the resulting patch and its location will be returned.

Patching possible call chains and vanilla chains. If the gadget chain is a possible call chain or vanilla chain, PFORTIFIER may not be able to suggest an exact patch that preserves all intended functionalities. This is because such chains do not involve magic method calls that are obviously unexpected, *e.g.*, caused by undefined method calls or incorrect attribute types. Therefore, in such cases, PFORTIFIER generates patch suggestions instead of directly creating patches. This also applies to several magic methods, as described before. To design an effective approach to generating patch suggestions, we thoroughly studied the known gadget chains and summarized the corresponding possible patches. Specifically, for possible call chains, PFORTIFIER suggests setting the controllable fields to *NULL*. As for the vanilla chains, PFORTIFIER suggests limiting the deserialization of the entry class.

Algorithm 1: Rule-based patch generation via found gadget chain

```
   Input: jmpNodes
   Output: the patch for the gadget chain
1  for jnode ∈ jmpNodes do
2      jmpStatement ← getStatement(jnode);
3      method ← getCurrentMethod(jnode);
4      class ← getCurrentClass(jnode);
5      patch ←
         getPatchByRules(jmpStatement, method, class);
6      if patch != False then
7          break;
8      end
9  end
10 if patch != False then
11     finalPatch ← patch;
12 else
13     finalPatch ← None;
14 end
```

Note that the patch suggestions are also automatically generated, as PFORTIFIER can identify the controllable fields, and the entry classes. The generated suggestions ensure the vulnerabilities can be mitigated, yet the functional compatibility needs to be confirmed. In Section 6, we further demonstrate that PFORTIFIER can apply restrictive patches to 48.5% detected gadget chains, and the rest chains are shorter and easier to audit. Therefore, we believe the above patching strategy can significantly reduce the manual efforts required in mitigating POI vulnerabilities.

4.5. Minimal Working Example

Below, we use the code in Listing 2 as an example to demonstrate the workflow of PFORTIFIER.

Firstly, in the code summarization phase, PFORTIFIER extracts classes and the corresponding ASTs from the source code, specifically, {"LogPrinter": ASTs of __destruct(), "Logger": ASTs of getLog(), "SystemInfo": ASTs of __call()}. Based on that, the `attr_func_dict` map is constructed as: {"__destruct": ["LogPrinter"], "getLog": ["Logger"], "__call": ["SystemInfo"]}.

Secondly, in the simulated execution phase, PFORTIFIER initiates the execution from the entry method `LogPrinter::__destruct()`. In this context, all attributes of `$this` are controllable, because the `LogPrinter` object is deserialized from an attacker-controllable string. When executing `$this->logger->getLog()`, since `$this->logger` is controllable, PFORTIFIER finds the corresponding classes through attr_func_dict["getLog"] and

TABLE 1. MAGIC METHOD CALL SITES AND CORRESPONDING PATCH GENERATION METHODS

Magic Method Call Site	Magic Method Call Site Example	Patch
__call	`$this->attr->method()`	`if(!method_exists($this->attr,'method')){die();}`
__get	`$this->attr1->attr2`	`if(!property_exists($this->attr1,'attr2')){die();}`
__set	`$this->attr1->attr2="something"`	`if(!property_exists($this->attr1,'attr2')){die();}`
__isset	`isset($this->attr1->attr2)`	`if(!property_exists($this->attr1,'attr2')){die();}`
__unset	`unset($this->attr1->attr2)`	`if(!property_exists($this->attr1,'attr2')){die();}`
__toString	`echo $this->attr`	`if(!is_string($this->attr)){die();}`
Iterator related methods	`foreach($this->attr as $key=>$value)`	`if($this->attr instanceof Iterator){die();}`
ArrayAccess related methods	`$this->attr["key"]`	`if($this->attr instanceof ArrayAccess){die();}`

attr_func_dict["__call"], which will be class `Logger` and `SystemInfo`. Then, `$this->logger->getLog()` is recorded as a PM call node, and the PM call branches are analyzed. For the `SystemInfo::__call()` branch, upon analyzing up to Line 16, a command injection vulnerability is detected, because the argument of `system()` is controllable. Therefore, the chain is stored. However, no vulnerability is found in the `Logger::getLog()` branch. After analyzing the branches, the results are merged. Since the `Logger::getLog()` branch returns a controllable object, it is retained as the merged result according to the greedy strategy. Upon returning to the `echo()` statement in Line 4, an XSS vulnerability is detected, with the chain also stored.

Lastly, PFORTIFIER generates patches for the vulnerabilities.For the command injection chain, where the PM call node is `$this->logger->getLog()`, the first patch rule in Table 1 is matched, because the sink function is invoked through magic method `__call()`. Therefore, PFORTIFIER generates a patch `if(!method_exists($this->logger, 'getLog')){die();}`. For the XSS chain, since no PM is present in the call chain at Line 4, PFORTIFIER provides a patch suggestion by adding `public function __wakeup(){die();}` to the LogPrinter class. At this point, developers can refer to the generated patch and suggestion to repair the code accordingly.

5. Implementation

PFORTIFIER leverages PHPLY [29] to extract the AST nodes from PHP source code and conducts a recursive simulation execution on each type of node. We implemented the simulator that tracks taint propagation in PHP applications from scratch, in over 5K lines of code. We will open source the implementation to benefit future research.

6. Evaluation

We have conducted a comprehensive evaluation of PFORTIFIER, aiming to answer the following research questions:

RQ1: Gadget chain coverage. Can PFORTIFIER effectively detect known gadget chains? Can PFORTIFIER detect previously unknown chains? How does the result compare with the state-of-the-art?

TABLE 2. SINKS OF PFORTIFIER

Vulnerability Type	Sink Functions or Statements
PHPINFO called	phpinfo
Arbitrary file reading	show_source, highlight_file, file_get_contents, readfile, fopen, file, fread
Arbitrary file deletion	unlink
File sensitive operation	rmdir, mkdir, chmod, chown, chgrp, touch, copy, rename, link, symlink
Arbitrary code execution	array_map, create_function, call_user_func, call_user_func_array, assert, dl, register_tick_function, register_shutdown_function
Preg_replace arbitrary code execution	preg_replace
Preg_replace_callback arbitrary code execution	preg_replace_callback
Command injection	system, exec, passthru, shell_exec, pcntl_exec, proc_open, popen, escapeshellcmd
Mail() options injection	mail
Arbitrary file write	file_put_contents, fputs, fwrite
XXE	simplexml_load_string, simplexml_load_file
SSRF	get_headers, curl_exec, mysqli::query
SQL injection	mysql_db_query, mysqli_query, pg_execute, pg_insert, pg_query, pg_select, pg_update, sqlite_query, msql_query, mssql_query, odbc_exec, fbsql_query, sybase_query, ibase_query, dbx_query, ingres_query, ifx_query, oci_parse, sqlsrv_query, maxdb_query, db2_exec, sqlite_exec, mysql_query
XSS	print_r, printf, vprintf, trigger_error, user_error, odbc_result_all, ovrimos_result_all, ifx_htmltbl_result, ECHO, PRINT, EXIT
File uploading	move_uploaded_file
Eval code execution	eval
File inclusion	include, include_once, require, require_once

RQ2: Precision of gadget chain detection. Does PFORTIFIER report less false positives of gadget chains, compared with the state-of-the-art?

RQ3: Efficiency of gadget chain detection. How efficient is PFORTIFIER in detecting gadget chains? How does the efficiency compare with the state-of-the-art?

RQ4: Performance for patch generation. What is PFORTIFIER's performance in patching gadget chains? Does it generate faulty patches? How effective is it in providing patch suggestions?

RQ5: Impact of gadget chain categories. Do all the three categories of gadget chains prevalently exist? How does that affect the patching effectiveness?

6.1. Experiment Setup

Baselines and dataset. In this study, we select FUGIO [21] and PHPGGC as the baseline. FUGIO is a state-of-the-art tool for gadget chain detection and exploit generation. It firstly conducts a static analysis by tracking back the data flow from sinks to attacker controlled deserialization sources. FUGIO then verifies the results and generates exploits through a fuzzing process, in which it tries to trigger the execution of detected gadget chains. PHPGGC is a public library of known PHP gadget chains as previously introduced. This allows us to evaluate the superiority of PFORTIFIER in gadget chain detection. As the first to develop automatic POI patching frameworks, however, we were not able to compare PFORTIFIER against other tools in terms of patch generation capabilities.

As FUGIO does not disclose the specific gadget chains detected from its dataset, in this work, we excluded the applications evaluated in FUGIO that are not present in PHPGGC, resulting in a dataset of 25 PHP applications. To further evaluate the extensibility and scalability of PFORTIFIER, we augmented the dataset with 6 other popular applications. In total, we derive a dataset of 31 applications for our evaluation. Our dataset covers mainstream frameworks (e.g., Zend Framework, CodeIgniter), popular libraries (e.g., PHPExcel, Omnipay), and applications (e.g., Drupal, WordPress) with known PHPGGC chains, ensuring the representativeness.

Deduplication of gadget chains. Currently, there is no standard definition of gadget chains. FUGIO identifies gadget chains as distinct execution chains, *i.e.*, if a method is executed once and twice respectively in two runs, they will be counted as two distinct chains. However, such chains usually exhibit the same semantics, and the presence of long and complex chains will pose significant challenges for manual auditing [21]. Therefore, we propose a specialized chain identification method. Our approach is based on two key observations: first, different entries branch into different gadget chains, making the entry method a critical element in the chain. Second, the effect of an attack is determined by the sinks. Thus, we define the gadget chains as an entry-sink pair. This helps minimize the number of chains to be manually verified, and ensures that semantics of a gadget chain can be preserved.

Time budget. In this work, we limited the maximum execution time of FUGIO according to Equation 1. Basically, compared with PFORTIFIER, we allocate more time budget to FUGIO, as the fuzzing verification process can be time-consuming. The upper bound for FUGIO execution is set to 3 hours to manage the experiment cost on 31 applications. Although FUGIO might be able to cover more chains when executed longer, our experiments have proved that PFORTIFIER significantly outperforms FUGIO in gadget chain detection with much less time budget.

$$t_{fugio} = \begin{cases} 100t_{\text{PFORTIFIER}}, & t_{\text{PFORTIFIER}} \leq 10s, \\ 10t_{\text{PFORTIFIER}}, & t_{\text{PFORTIFIER}} > 10s \quad and \\ & t_{fugio} \leq 10800s, \\ 10800s, & t_{fugio} > 10800s. \end{cases} \tag{1}$$

Experiment environment. The experiments were conducted on a machine with an Intel Xeon Silver 4210 CPU operating at 2.20 GHz and 64 GB RAM.

6.2. Experimental Evaluation

To ensure the accuracy and reliability of our results, we manually verified the detected gadget chains and the generated patches. In particular, for applications that underwent patching, we meticulously inspected the the official documentation of the patched sections to check whether the functionalities remained unaffected.

6.2.1. Gadget chain coverage (RQ1). As FUGIO first statically detects the gadget chains and then verifies them dynamically, we compared FUGIO and PFORTIFIER in terms of both the static and dynamic detection. The results are listed in Table 3 and Table 4, respectively. Note that the dynamic verification leads to lower chain coverage by filtering the chains that cannot be executed during fuzzing. Therefore, we believe the comparison fairly demonstrate the superiority of PFORTIFIER.

Failure marks. 11 applications in our dataset cannot be analyzed by FUGIO. For fairness, we excluded these applications from the comparison. Specifically, three applications require a PHP version exceeding 7.2, which has not been supported by FUGIO. We marked them as *VersionOut*. During the static summary generation phase of FUGIO, the analysis of seven other applications got stuck or encountered queue delivery errors, and were marked as *Unsupported*. Furthermore, Typo3 9.3.0 threw a memory exhausted error when triggering the POI, which is marked as *MemoryOut*.

Static detection results. In Table 3, the "PHPGGC" column displays the number of known gadget chains in PHPGGC. The "Detected" columns represent the number of PHPGGC chains detected by FUGIO and PFORTIFIER. Additionally, the "New" columns indicate the number of new gadget chains detected by FUGIO and PFORTIFIER.

As shown in Table 3, among the 20 applications without failure marks, PFORTIFIER achieves a total gadget chain coverage of 92.73% ((29+22)/(33+22)), outperforming FUGIO's coverage of 54.55% ((21+9)/(33+22)). Compared to the state-of-the-art, PFORTIFIER showcases a remarkable 38.18% higher total gadget chain coverage. PFORTIFIER achieves higher coverage for two primary reasons. Firstly, PFORTIFIER optimizes the speed of static analysis, enabling it to comprehensively analyze the entire application within a restricted timeframe. Secondly, PFORTIFIER adopts a simulation-based execution, which precisely tracks the point-to relation and taint status across method calls. In contrast, according to our case studies, FUGIO sometimes cannot precisely record controllable objects when long

TABLE 3. EVALUATION RESULTS OF PFORTIFIER AND FUGIO STATIC ANALYSIS FOR GADGET CHAIN DETECTION OF EACH APPLICATION. (WP: WORDPRESS)

Applications	PHPGGC	FUGIO			PFORTIFIER				
	Chains	Detected	New	Time	Detected	New	Time	Patch Coverage	Patch Suggestion Coverage
TCPDF 6.3.4	1	1	0	3m 20s	1	0	2m 29s	0	100%
Drupal7	2	1	0	2m 10s	1	0	13s	0	50%
Laminas 2.11.2	1	1	1	13m 20s	1	1	8s	50%	50%
SwiftMailer 5.4.12	2	2	2	10m	2	2	6s	0	100%
SwiftMailer 6.0.0	1	1	0	15m	1	1	9s	0	100%
Monolog 1.7.0	2	2	1	3m 20s	2	1	2s	0	100%
Monolog 1.18.0	1	1	1	5m	1	1	3s	0	100%
Monolog 2.0.0	1	1	2	5m	1	2	3s	0	100%
PHPExcel 1.8.1 (WP)	5	4	1	7m 50s	5	3	47s	75%	25%
Dompdf 0.8.0 (WP)	1	0	0	7m 20s	1	0	44s	100%	0
Guzzle (WP)	5	1	0	5m 40s	4	1	34s	50%	
WooCommerce 2.6.0 (WP)	1	1	0	5m 6s	1	0	47s	100%	0
WooCommerce 3.4.0 (WP)	1	0	0	9m 30s	1	0	57s	100%	0
Emailsubscribers (WP)	1	1	0	5m 40s	1	0	34s	100%	0
EverestForms (WP)	1	1	0	5m 40s	1	0	34s	100%	0
Smarty	2	1	0	2m 20s	1	0	4s	0	50%
SwiftMailer 5.0.1	1	1	0	2m 10s	1	0	13s	0	100%
ZendFramework 1.12.20	4	1	0	2h 35m	3	5	15m 40s	44.44%	55.56%
Omnipay	0	0	0	13m 20s	0	1	8s	100%	0
ThinkPHP 6.1.0	0	0	1	8m 30s	0	4	51s	75%	25%
TYPO3 9.3.0	1	0	0	MemoryOut	1	2	1m 35s	33.33%	66.67%
Yii 1.1.20	1	0	0	Unsupported	1	4	2m 49s	60%	40%
CodeIgniter 4.1.3	1	0	0	VersionOut	1	3	3m 7s	100%	0
Swoft 2.0.11	0	0	0	Unsupported	0	4	43s	25%	75%
PHPCSFixer 2.17.3	2	0	0	Unsupported	2	2	35s	0	100%
Spiral 2.8	0	0	0	Unsupported	0	7	55s	71.43%	28.57%
Yii 2.0.37	2	0	0	Unsupported	1	3	3m 25s	80%	20%
PopPHP 4.7.0	0	0	0	VersionOut	0	2	3m 23s	50%	50%
CakePHP 3.9.6	2	0	0	Unsupported	0	4	2m 35s	100%	0
Slim 3.8.1	1	0	0	Unsupported	1	2	17s	100%	0
Slim 4.11.0	0	0	0	VersionOut	0	1	30s	100%	0
Total	43	21	9		36	56		52.53%	45.45%

and complex method call chains exist. For all 31 applications, PFORTIFIER achieves an impressive total coverage of 92.93% ((36+56)/(43+56)) for gadget chains. Additionally, PFORTIFIER surpasses the state-of-the-art by detecting 13 more new chains from the 20 applications without failure marks, and a total of 56 new chains from all applications. This highlights PFORTIFIER's effective chain detection capabilities. To facilitate future research, we released all new chains on GitHub[3].

Dynamic detection results. As presented in Table 4, FUGIO's dynamic analysis achieves a total chain coverage of only 27.27% compared to PFORTIFIER's impressive 92.73%. The dynamic verification in FUGIO improves the precision yet harms the coverage of gadget chains, causing many chains undetected and cannot be patched.

6.2.2. Precision of gadget chain detection (RQ2). Table 5 presents the false positive rates for each tool. Specifically, the false positive rate is calculated as n_{FP}/n_{all}, where n_{FP} represents the number of deduplicated false positive chains or patches, and n_{all} denotes the total number of deduplicated detected chains or generated patches.

As shown in Table 5, among the 20 applications without failure marks, PFORTIFIER achieves a total false positive rate of 68.69%, while FUGIO's false positive rate is 77.56% in the static detection stage. Comparatively, PFORTIFIER demonstrates an 8.87% lower false positive rate than the

3. https://github.com/CyanM0un/PFortifier_DataSet

TABLE 4. FUGIO DYNAMIC DETECTION COVERAGE AND PFORTIFIER COVERAGE. (WP: WORDPRESS)

Applications	FUGIO Dynamic Coverage	PFORTIFIER Coverage
TCPDF 6.3.4	1 / 1	1 / 1
Drupal7	1 / 2	1 / 2
Laminas	1 / 2	2 / 2
SwiftMailer 5.4.12	1 / 4	4 / 4
SwiftMailer 6.0.0	0 / 2	2 / 2
Monolog 1.7.0	0 / 3	3 / 3
Monolog 1.18.0	0 / 2	2 / 2
Monolog 2.0.0	0 / 3	3 / 3
PHPExcel 1.8.1 (WP)	5 / 8	8 / 8
Dompdf 0.8.0 (WP)	0 / 1	1 / 1
Guzzle (WP)	1 / 6	5 / 6
WooCommerce 2.6.0 (WP)	1 / 1	1 / 1
WooCommerce 3.4.0 (WP)	0 / 1	1 / 1
Emailsubscribers (WP)	1 / 1	1 / 1
EverestForms (WP)	1 / 1	1 / 1
Smarty	1 / 2	1 / 2
SwiftMailer 5.0.1	0 / 1	1 / 1
ZendFramework 1.12.20	1 / 9	8 / 9
Omnipay	0 / 1	1 / 1
ThinkPHP 6.1.0	0 / 4	4 / 4
Total	27.27% (15 / 55)	92.73% (51 / 55)

TABLE 5. FALSE POSITIVE RATES OF PFORTIFIER, STATIC ANALYSIS OF FUGIO, AND PFORTIFIER PATCH GENERATION. (WP: WORDPRESS)

Applications	False Positive Rate of PFORTIFIER	False Positive Rate of FUGIO Static Analysis	False Positive Rate of PFORTIFIER Patch Generation	False Positive Rate of PFORTIFIER Patch Suggestion
TCPDF 6.3.4	0 (0 / 5)	0 (0 / 2)	No Patch Generated (0 / 0)	0 (0 / 1)
Drupal7	50% (1 / 2)	80% (4 / 5)	No Patch Generated (0 / 0)	50% (1 / 2)
Laminas 2.11.2	40% (2 / 5)	0 (0 / 2)	0 (0 / 1)	0 (0 / 1)
SwiftMailer 5.4.12	54.55% (6 / 11)	64.71% (11 / 17)	0 (0 / 2)	0 (0 / 3)
SwiftMailer 6.0.0	55.56% (10 / 18)	68.42% (26 / 38)	0 (0 / 1)	14.29% (1 / 7)
Monolog 1.7.0	33.33% (3 / 9)	14.29% (1 / 7)	No Patch Generated (0 / 0)	0 (0 / 3)
Monolog 1.18.0	42.86% (6 / 14)	42.86% (6 / 14)	No Patch Generated (0 / 0)	0 (0 / 4)
Monolog 2.0.0	50% (7 / 14)	66.67% (10 / 15)	No Patch Generated (0 / 0)	0 (0 / 5)
PHPExcel 1.8.1 (WP)	74.42% (32 / 43)	82.86% (29 / 35)	20% (2 / 10)	62.5% (5 / 8)
Dompdf 0.8.0 (WP)	90.91% (10 / 11)	100% (5 / 5)	50% (2 / 4)	100% (1 / 1)
Guzzle (WP)	58.33% (14 / 24)	75% (12 / 16)	68.75% (11 / 16)	50% (3 / 6)
WooCommerce 2.6.0 (WP)	75% (3 / 4)	0 (0 / 1)	50% (1 / 2)	100% (2 / 2)
WooCommerce 3.4.0 (WP)	77.78% (7 / 9)	100% (4 / 4)	55.56% (5 / 9)	33.33% (1 / 3)
Emailsubscribers (WP)	75% (3 / 4)	50% (1 / 2)	0 (0 / 2)	100% (2 / 2)
EverestForms (WP)	75% (3 / 4)	50% (1 / 2)	0 (0 / 2)	100% (2 / 2)
Smarty	83.33% (5 / 6)	66.67% (2 / 3)	0 (0 / 1)	66.67% (2 / 3)
SwiftMailer 5.0.1	69.70% (23 / 33)	75% (27 / 36)	0 (0 / 5)	0 (0 / 6)
ZendFramework 1.12.20	72.54% (177 / 244)	98.78% (81 / 82)	7.89% (3 / 38)	12.5% (1 / 8)
Omnipay	83.33% (5 / 6)	80% (12 / 15)	0 (0 / 1)	No Suggestion (0 / 0)
ThinkPHP 6.1.0	79.31% (23 / 29)	92.86% (13 / 14)	7.69% (1 / 13)	0 (0 / 4)
TYPO3 9.3.0	50% (41 / 82)	MemoryOut	21.05% (4 / 19)	40% (4 / 10)
Yii 1.1.20	76.19% (16 / 21)	Unsupported	33.33% (2 / 6)	0 (0 / 2)
CodeIgniter 4.1.3	86.07% (105 / 122)	VersionOut	20% (4 / 20)	33.33% (2 / 6)
Swoft 2.0.11	67.24% (39 / 58)	Unsupported	0 (0 / 4)	0 (0 / 5)
PHPCSFixer 2.17.3	61.90% (13 / 21)	Unsupported	100% (6 / 6)	0 (0 / 4)
Spiral 2.8	75.58% (65 / 86)	Unsupported	8.33% (1 / 12)	16.67% (1 / 6)
Yii 2.0.37	55.84% (43 / 77)	Unsupported	14.29% (2 / 14)	0 (0 / 5)
PopPHP 4.7.0	52.17% (12 / 23)	VersionOut	50% (1 / 2)	0 (0 / 2)
CakePHP 3.9.6	74.83% (110 / 147)	Unsupported	0 (0 / 5)	50% (1 / 2)
Slim 3.8.1	54% (27 / 50)	Unsupported	14.29% (1 / 7)	0 (0 / 17)
Slim 4.11.0	35% (14 / 40)	Unsupported	0 (0 / 6)	13.64% (3 / 22)
Total	68.69% (340 / 495)	77.56% (242 / 312)	21.60% (46 / 213)	19.08% (29 / 152)

state-of-the-art. As described before, this can be attributed to the precise tracking of controllable objects in complex call chains. Although FUGIO may avoid false positives in the verification stage, this also leaves many chains undetected. As the ultimate goal of PFORTIFIER is to be comprehensive in patching the vulnerabilities, we think the false positive rate is acceptable.

6.2.3. Efficiency of gadget chain detection (RQ3). PFORTIFIER achieves notable performance in terms of gadget chain detection on the 31 applications. The longest execution time recorded for PFORTIFIER is 15 minutes and 40 seconds, while the average execution time is 1 minute and 27 seconds. In contrast, FUGIO reaches the maximum execution time specified in Section 6.1 for most applications. Overall , PFORTIFIER demonstrates remarkable efficiency by outperforming FUGIO with an average speedup of more than 10 times, clearly demonstrating the innovative advantage of PFORTIFIER.

6.2.4. Performance for patch generation (RQ4). To understand how effective is PFORTIFIER in generating patches, we analyzed the coverage and precision of patched gadget chains. The results are described below.

Coverage. The last two columns in Table 3 present the patch coverage and patch suggestion coverage. Specifically, the patch coverage is calculated as P/GC_{all}, where GC_{all} denotes the total number of all PHPGGC and newly detected chains, and P represents the number of restrictively patched chains. The patch suggestion coverage is S/GC_{all}, where S represents the number of chains for which PFORTIFIER successfully generates patch suggestions.

PFORTIFIER successfully patched 52.53% of all gadget chains and provided patch suggestions for the remaining 45.45% of chains, resulting in an impressive total chain coverage of 97.98%. Importantly, rigorous manual checks confirmed that all generated patches maintained the applications' functionality intact. The high chain coverage achieved in the patch and patch suggestion generation strongly demonstrates the effective patch generation capabilities of PFORTIFIER. It is important to note the total number of patches and patch suggestions is less than the number of chains. This is because multiple chains can be patched by restricting one single PM call. For instance, in ZendFramework 1.12.20, which comprises 472,015 lines of code, PFORTIFIER identifies 244 chains and generates 46 patches and patch suggestions. Conducting a manual review of the entire application and the chains would be a daunting process. Nevertheless, PFORTIFIER enables developers to validate the generated patches instead, substantially enhancing the efficiency.

Precision. The last two columns in Table 5 indicate the false positive rates of the patches and patch suggestions generated by PFORTIFIER. Lower false positive rate for patches is desirable for developers to confirm and apply the patches.

Out of the 213 restrictive patches, 46 are false positives, resulting in a false positive rate of 21.60%. Similarly, for patch suggestion generation, 29 out of 152 suggestions are false positives, yielding a false positive rate of 19.08%. The false positive are caused by inaccurate gadget chain detection, leading to the patch of non-exploitable chains. Nevertheless, the number of false positives is still low, rendering manual checks affordable for developers.

6.2.5. Impact of gadget chain categories (RQ5). Table 6 provides an overview of the number and average length of the three types of gadget chains. As shown, the number of possible call chains and vanilla chains is smaller than that of the magic method chains, and their average chain lengths are also shorter. In essence, the construction of a gadget chain resembles privilege escalation. As the chain length increases, exploitation also becomes easier. Although PFORTIFIER cannot generate restrictive patches for the possible call chains and vanilla chains, as such chains are also less prevalent and easier to review, we believe our patching strategy is effective for practical use.

6.3. Case Studies

We now present our case studies of several real-world examples of restrictive patches and patch suggestions. Furthermore, we demonstrate a case of an undetected chain that was still successfully patched by PFORTIFIER, when generating patches for other chains that involve the same PM call. This illustrates the effectiveness and benefit of our patching strategy.

```
1  <?php
2  namespace Spiral\Composer {
3   class Downloader {
4    public function __destruct() {
5     if ($this->dir === null) {
6      return;
7     }
8     if(!is_string($this->dir)){die();}
9     // patch generated by PFORTIFIER
10    $files = new \RecursiveIteratorIterator(
11     new \RecursiveDirectoryIterator(
12      $this->dir, \RecursiveDirectoryIterator::SKIP_DOTS)
            ,
13     \RecursiveIteratorIterator::CHILD_FIRST);
14    // trigger __toString
15    ...
16   }
17  }
18 }
19 namespace Spiral\Reactor {
20  class FileDeclaration {
21   public function __toString() {
22    return $this->render(0);
23   }
24   public function render(int $indentLevel = 0) {
25    $result = "<?php\n";
26    if (!$this->docComment->isEmpty()) {
27     // call PhpOption\LazyOption::isEmpty
28     $result .= $this->docComment->render($indentLevel) .
           "\n";
29    }
30    ...
31   }
32  }
33 }
34 namespace PhpOption {
35  class LazyOption {
36   public function isEmpty() {
37    return $this->option()->isEmpty();
38   }
39   private function option() {
40    if (null === $this->option) {
41     $option = call_user_func_array(
42     $this->callback, $this->arguments); // sink
43     ...
44    }
45   }
46  }
47 }
```

Listing 6. A magic method chain in Spiral 2.8 and the patch generated by PFORTIFIER

Spirl 2.8. Listing 6 demonstrates a magic method chain detected in Spiral 2.8. The gadget chain starts from `Downloader::__destruct()`. In Line 12, the first parameter of the `RecursiveDirectoryIterator` constructor is treated as a string, leading to the call of `FileDeclaration::__toString()`. Subsequently, at Line 26, `LazyOption::isEmpty()` is invoked, eventually resulting in an *arbitrary function call* in Line 41-42. To mitigate this vulnerability, PFORTIFIER developed a patch for the chain at Line 8, which checks the type of `$this->dir` and prevents it from proceeding to the next node.

```
1  <?php
2  namespace Monolog\Handler {
3   class RollbarHandler {
4    public function __destruct() {
5     $this->close();
6    }
7    public function close() {
8     $this->flush();
9    }
10   public function flush() {
11    if ($this->hasRecords) {
12     $this->rollbarLogger->flush();
13     $this->hasRecords = false;
14    }
15   }
16   public function __wakeup(){$this->rollbarLogger = NULL
         ;}
17   // patch suggestion generated by PFORTIFIER
18  }
19 }
20 namespace PHPUnit\Runner{
21  class ResultCacheExtension {
22   public function flush() {
23    $this->cache->persist();
24   }
25  }
26  class DefaultTestResultCache {
27   public function persist() {
28    ...
29    file_put_contents($this->cacheFilename, json_encode(
30     ['version' => self::VERSION, 'defects' => $this->
           defects,
31     'times'   => $this->times,]), LOCK_EX); // sink
32   }
33  }
34 }
```

Listing 7. A possible call chain in Spiral 2.8 and the patch suggestion generated by PFORTIFIER

Listing 7 illustrates a possible call chain in Spiral 2.8. This chain originates from `RollbarHandler::__destruct()`. In Line 12, `ResultCacheExtension::flush()` is called, followed by `DefaultTestResultCache::persist()` in Line 23, resulting in an *arbitrary file write* in Lines

TABLE 6. THE NUMBER AND AVERAGE LENGTH OF DIFFERENT TYPES OF CHAINS

Type of Gadget Chain	Magic Method Chain	Possible Call Chain	Vanilla Chain
Number of Gadget Chain	48	31	20
Average Chain Length	4.7 (224 / 48)	4.0 (125 / 31)	1.7 (34 / 20)

29-31. For gadget chains without magic method calls, PFORTIFIER generates a patch suggestion at Line 16, which overwrites the controllable object to `NULL`. In this case, a manual review is required to further confirm the patch suggestion. As mentioned in Section 6.2.5, checking the suggestions does not require significant engineering efforts.

```
1  <?php
2  namespace Symfony\Component\Process {
3   class Process {
4    public function __destruct() {
5     if(!method_exists($this->processPipes,'close')){die()
          ;}
6      // patch generated by PFORTIFIER
7      $this->processPipes->close();
8    }
9   }
10 }
11 namespace Cake\ORM {
12  class Table {
13   public function __call($method, $args) {
14    ...
15    return $this->_behaviors->call($method, $args);
16    ...
17   }
18  }
19 }
20 namespace Cake\ORM {
21  class BehaviorRegistry {
22   public function call($method, array $args = []) {
23    ...
24    list($behavior, $callMethod) = $this->_methodMap[
          $method];
25    return call_user_func_array([$this->_loaded[$behavior
          ],
26                  $callMethod], $args); // springboard
27   }
28  }
29 }
30 namespace Cake\Shell {
31  class ServerShell {
32   public function main(){
33    $command = sprintf('php -S %s:%d -t %s',
34     $this->_host, $this->_port,
35     escapeshellarg($this->_documentRoot));
36    ...
37    system($command); //final sink
38   }
39  }
40 }
```

Listing 8. An exploitable gadget chain in CakePHP 3.9.6 and the patch generated by PFORTIFIER

CakePHP 3.9.6. Listing 8 showcases a magic call chain in CakePHP 3.9.6, and PFORTIFIER identifies an *arbitrary method call* in Line 25. However, as the first argument of `call_user_func_array` is an array, an attacker can direct the execution to arbitrary method of any class. Hence, PFORTIFIER cannot continue the simulated execution and failed to detect another chain that ends at `ServerShell::main()` in Line 37. However, PFORTIFIER can still generate a patch at the entry method in Line 5, which restricts the next jump in the chain, *i.e.*, `Table::__call()`. As a result, although the second chain is not detected, PFORTIFIER still successfully patched the vulnerability.

7. Discussion and Limitations

Our experiments have proved that PFORTIFIER exhibits the ability to generate patches and patch suggestions for the majority of gadget chains. In this section, we discuss several limitations for future improvement.

First, PFORTIFIER effectively preserves the functionality of applications that do not need to call magic methods in gadget chains, a common scenario in real-world applications. However, the patching strategy may also brings side effects in rare cases, *e.g.*, when the developers intend to call magic methods at entries to proceed the execution. Nonetheless, our experiments have demonstrated that restricting magic method calls did not break any functionality. As also discussed in Section 4.4, using magic methods can easily lead to gadget chains. Therefore, we believe it is not a good coding practice and should be avoided.

Secondly, PFORTIFIER generates patch suggestions for possible call chains and vanilla chains, instead of providing an exact patch. Different from the chains that invoke magic methods in obviously unexpected scenarios, in these chains, it is hard to automatically decide whether the patch may break critical functionalities. Nonetheless, the generated suggestions can always mitigate the vulnerabilities. PFORTIFIER just relies on the developers to confirm functional compatibility, as they have sufficient internal knowledge. We leave it as a future work to automatically verify the patch suggestions.

Finally, PFORTIFIER may yield more false positives compared with FUGIO, which verifies detected chains through fuzzing. However, it is essential to note the dynamic verification in FUGIO also caused a lower recall, leaving many chains unpatched. Instead, PFORTIFIER aims to maximize the chain coverage to provide comprehensive patches. Furture works may attempt to further reduce the false positives, *e.g.*, by applying symbolic execution for a more precise execution simulation.

8. Related Work

Web application analysis methods. Given the critical concern surrounding web vulnerabilities, extensive research has been conducted on vulnerability detection [30]–[53]. For instance, RIPS [31] facilitated PHP vulnerability detection through static taint analysis of PHP tokens. TChecker [36] improved the accuracy of static taint analysis for PHP by modeling the types of objects and other dynamic features. NAVEX [44] guided dynamic analysis using static

approaches, for automatic vulnerability verification and exploit generation. These tools are orthogonal to PFORTIFIER, which aims to automatically patch the POI vulnerabilities.

Deserialization vulnerability detection methods. Building upon the web application analysis methods, several works have focused on deserialization vulnerability detection [17]–[21], [54]. Dahse et al. [17] proposed the first automated PHP gadget chain detection method by performing a static analysis on the ASTs. FUGIO [21] introduced the first automatic exploit generation method for gadget chains. However, these methods primarily focus on gadget chain detection and do not support automatic patch generation. We also proved that PFORTIFIER outerforms state-of-the-art tools in terms of gadget chain detection through a precise simulated execution.

Automatic patch generation methods. Numerous efforts [4]–[16] have been dedicated to automating vulnerability patch generation. Kim et al. [10] summarized from human-written patches 10 templates for patching Java vulnerabilities. Huang et al. [11] proposed a property-based patching approach for automatic mitigation of buffer overflow, bad cast, and integer overflow vulnerabilities. AppSealer [12] mitigated component hijacking vulnerabilities in Android apps by statically analyzing the bytecode and blocking vulnerability sinks. In this work, we focused on POI vulnerabilities, of which the exploits heavily rely on property-oriented programming. This causes unpredictable execution paths, and existing patching approaches cannot be readily applied.

9. Conclusion

In this work, We introduce PFORTIFIER, the first automatic patch generation method for POI vulnerabilities. PFORTIFIER leverages an efficient and high-coverage static analysis module to detect gadget chains, and generates patches by imposing restrictions on related PM calls. Our comprehensive evaluation of PFORTIFIER demonstrates its remarkable performance. PFORTIFIER significantly outperforms state-of-the-art gadget chain detection tools, achieving higher chain coverage and analysis speed. It additionally identifies 56 previously unknown gadget chains, of which 10 have been validated by PHPGGC. In terms of patch generation, PFORTIFIER successfully generates patches for all detected chains, resulting in a total coverage of 97.98%.

Acknowledgments

The authors would like to thank our shepherd and the anonymous reviewers for their helpful suggestions. The research described in this paper is sponsored by the National Natural Science Foundation of China (No. 62402423) and the National Natural Science Foundation of Sichuan under Grant (No.2025ZNSFSC0509).

References

[1] W3Techs. (2023) Usage statistics of server-side programming languages for websites. [Online]. Available: https://w3techs.com/technologies/overview/programming_language

[2] T. O. Group. (2023) Php object injection — owasp foundation. [Online]. Available: https://owasp.org/www-community/vulnerabilities/PHP_Object_Injection

[3] A. Security. (2023) PHPGGC: PHP generic gadget chains. [Online]. Available: https://github.com/ambionics/phpggc

[4] Y. Shi, Y. Zhang, T. Luo, X. Mao, Y. Cao, Z. Wang, Y. Zhao, Z. Huang, and M. Yang, "Backporting security patches of web applications: A prototype design and implementation on injection vulnerability patches," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1993–2010.

[5] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic hot patch generation for android kernels," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2397–2414.

[6] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wang, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[7] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching." in *USENIX Security Symposium*, 2017, pp. 1253–1270.

[8] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries." in *NDSS*, 2019.

[9] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 981–992.

[10] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[11] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 539–554.

[12] M. Zhang and H. Yin, "Appsealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications." in *NDSS*, 2014.

[13] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[14] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 702–713.

[15] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1–11.

[16] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[17] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in php: Automated pop chain generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 42–53.

[18] M. Shcherbakov and M. Balliu, "Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web," in *Network and Distributed Systems Security (NDSS) Symposium 202121-24 February 2021*, 2021.

[19] H. Shahriar and H. Haddad, "Object injection vulnerability discovery based on latent semantic indexing," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 801–807.

[20] S. Rasheed and J. Dietrich, "A hybrid analysis to detect java serialisation vulnerabilities," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1209–1213.

[21] S. Park, D. Kim, S. Jana, and S. Son, "{FUGIO}: Automatic exploit generation for {PHP} object injection vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 197–214.

[22] T. P. Group. (2023) Php: Magic methods. [Online]. Available: http://php.net/manual/language.oop5.magic.php

[23] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma *et al.*, "Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing," *arXiv preprint arXiv:2304.04233*, 2023.

[24] T. L. Group. (2023) Laravel - The PHP Framework For Web Artisans. [Online]. Available: https://laravel.com/

[25] S. Thomas. (2018) File Operation Induced Unserialization via the "phar://" Stream Wrapper. [Online]. Available: https://i.blackhat.com/us-18/Thu-August-9/us-18-Thomas-Its-A-PHP-Unserialization-Vulnerability-Jim-But-\Not-As-We-Know-It-wp.pdf

[26] T. Tan, Y. Li, and J. Xue, "Making k-object-sensitive pointer analysis more precise with still k-limiting," in *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*. Springer, 2016, pp. 489–510.

[27] ——, "Efficient and precise points-to analysis: modeling the heap by merging equivalent automata," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 278–291.

[28] Y. Li, T. Tan, A. Møller, and Y. Smaragdakis, "Precision-guided context sensitivity for pointer analysis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[29] S. Pitucha. (2018) PHP parser written in Python using PLY . [Online]. Available: https://github.com/viraptor/phply

[30] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo, "Securing web application code by static analysis and runtime protection," in *Proceedings of the 13th international conference on World Wide Web*, 2004, pp. 40–52.

[31] J. Dahse and J. Schwenk, "Rips-a static source code analyser for vulnerabilities in php scripts," in *Seminar Work (Seminer Çalismasi). Horst Görtz Institute Ruhr-University Bochum*. Citeseer, 2010.

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

[33] ——, "Static detection of second-order vulnerabilities in web applications," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 989–1003.

[34] A. Algaith, P. Nunes, F. Jose, I. Gashi, and M. Vieira, "Finding sql injection and cross site scripting vulnerabilities with diverse static analysis tools," in *2018 14th European dependable computing conference (EDCC)*. IEEE, 2018, pp. 57–64.

[35] J. Huang, Y. Li, J. Zhang, and R. Dai, "Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 581–592.

[36] C. Luo, P. Li, and W. Meng, "Tchecker: Precise static inter-procedural analysis for detecting taint-style vulnerabilities in php applications," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2175–2188.

[37] T. Jensen, H. Pedersen, M. C. Olesen, and R. R. Hansen, "Thaps: automated vulnerability scanning of php applications," in *Secure IT Systems: 17th Nordic Conference, NordSec 2012, Karlskrona, Sweden, October 31–November 2, 2012. Proceedings 17*. Springer, 2012, pp. 31–46.

[38] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner," in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 523–538.

[39] Y. Zheng and X. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 652–661.

[40] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2015.

[41] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Chainsaw: Chained automated workflow-based exploit generation," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 641–652.

[42] I. Andrianto, M. I. Liem, and Y. D. W. Asnar, "Web application fuzz testing," in *2017 International Conference on Data and Software Engineering (ICoDSE)*. IEEE, 2017, pp. 1–6.

[43] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, A. Razmjoo-Qalaei, M.-R. Zamiri-Gourabi, and J. W. Davidson, "Malmax: Multi-aspect execution for automated dynamic web server malware analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1849–1866.

[44] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "{NAVEX}: Precise and scalable exploit generation for dynamic web applications," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 377–392.

[45] M. Steffens, C. Rossow, M. Johns, and B. Stock, "Don't trust the locals: Investigating the prevalence of persistent client-side cross-site scripting in the wild." 2019.

[46] A. S. Buyukkayhan, C. Gemicioglu, T. Lauinger, A. Oprea, W. Robertson, and E. Kirda, "What's in an exploit? an empirical analysis of reflected server xss exploitation techniques." in *RAID*, 2020, pp. 107–120.

[47] T. Lee, S. Wi, S. Lee, and S. Son, "Fuse: Finding file upload bugs via penetration testing." in *NDSS*, 2020.

[48] J. Huang, J. Zhang, J. Liu, C. Li, and R. Dai, "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*, 2021, pp. 78–90.

[49] P. Li and W. Meng, "Lchecker: Detecting loose comparison bugs in php," in *Proceedings of the Web Conference 2021*, 2021, pp. 2721–2732.

[50] P. Li, W. Meng, K. Lu, and C. Luo, "On the feasibility of automated built-in function modeling for php symbolic execution," in *Proceedings of the Web Conference 2021*, 2021, pp. 58–69.

[51] B. Eriksson, G. Pellegrino, and A. Sabelfeld, "Black widow: Blackbox data-driven web scanning," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1125–1142.

[52] S. Khodayari and G. Pellegrino, "Jaw: Studying client-side csrf with hybrid property graphs and declarative traversals," in *USENIX Security Symposium*, 2021.

[53] J. Rautenstrauch, G. Pellegrino, and B. Stock, "The leaky web: Automated discovery of cross-site information leaks in browsers and the web," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[54] N. Koutroumpouchos, G. Lavdanis, E. Veroni, C. Ntantogian, and C. Xenakis, "Objectmap: Detecting insecure object deserialization," in *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*, 2019, pp. 67–72.

[55] T. P. Group. (2023) PHP: Iterator - Manual . [Online]. Available: https://www.php.net/manual/en/class.iterator.php

[56] ——. (2023) PHP: arrayaccess - Manual . [Online]. Available: https://www.php.net/manual/en/class.arrayaccess.php

Appendix A. POP-related Magic Methods in PHP

We list in Table 7 the magic methods that could be invoked in gadget chains.

TABLE 7. POP-RELATED MAGIC METHODS IN PHP

Magic Method	Description
__construct()	Called when an object is instanced
__destruct()	Called when an object is destroyed, which is usually used as the entry of the gadget chain
__sleep()	Called when an object is serialized
__wakeup()	Called when an object is unserialized, which is usually used as the entry of the gadget chain or constructing a gadget chain patch
__serialize()	Called when an object is serialized
__unserialize()	Called when an object is unserialized, which is usually used as the entry of the gadget chain or constructing a gadget chain patch
__call()	Called when the inaccessible method is invoked in an object context
__callStatic()	Called when the inaccessible static method is invoked in a static context
__invoke()	Called when an object is called as a function
__get()	Called when values from non-existent properties are read
__set()	Called when writing values to non-existent properties
__isset()	Called when isset() or empty() is used on non-existent properties
__unset()	Called when unset() is used on non-existent properties
__toString()	Called when an object is treated like a string
__set_state()	Called when var_export() is used on an object
__clone()	Called when the cloning of an object is complete
__debugInfo()	Called when var_dump() is used on an object
Iterator interface [55]	When conducting the foreach statement on an object, the rewind(), valid(), current(), key(), and next() methods of the object are called
ArrayAccess interface [56]	When an object is treated as an array, the offsetExists(), the offsetUnset(), the offsetGet() and the offsetSet() methods are called

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper tackles PHP Object Injection (POI) vulnerabilities by automatically generating patches for gadget chains that might lead to exploitation of POI vulnerabilities. Unlike existing tools that focus on detecting gadget chains, PFORTIFIER goes further by simulating code execution to identify vulnerable paths and automatically generating patches based on heuristic rules.

B.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) This paper addresses the important and interesting problem of mitigating PHP Object Injection vulnerabilities.
2) This paper advances the state-of-the-art in PHP Object Injection exploit chain identification, and implements new gadget chains and publishes the code.
3) Experimental results demonstrate high coverage of gadget chains, higher rate of patching detected chains, and efficiency of analysis, compared to the SOTA.