2023

Securing web applications through vulnerability detection and runtime defenses

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

SECURING WEB APPLICATIONS THROUGH VULNERABILITY DETECTION AND RUNTIME DEFENSES

by

RASOUL JAHANSHAHI

B.S., Isfahan University of Technology, 2013
M.S., Amirkabir University of Technology, 2016

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2023

Approved by

First Reader

Manuel Egele, Ph.D.
Associate Professor of Electrical and Computer Engineering

Second Reader

Gianluca Stringhini, Ph.D.
Associate Professor of Electrical and Computer Engineering

Third Reader

David Starobinski, Ph.D.
Professor of Electrical and Computer Engineering
Professor of Systems Engineering
Professor of Computer Science

Fourth Reader

Nick Nikiforakis, Ph.D.
Associate Professor of Computer Science
Stony Brook University

Knowledge without conscience is only the ruin of the soul.

François Rabelais

Acknowledgments

First, I would like to express my appreciation and gratitude to my advisor, Prof. Manuel Egele, for his continuous support and guidance throughout my PhD. His guidance made me a more resilient researcher. I would like to thank Prof. Gianluca Stringhini and Prof. David Starobinski for their guidance during the earlier stages of my PhD research. Additionally, I would like to extend my gratitude to Prof. Nikiforakis for his precious time, generous support, and insightful feedback.

I would like to thank my friends for their unquestionable role in maintaining my sanity, to whom I owe a great deal of debt: Rachael Ivison, Onur Zungur, Annie Rabi Bernard, Pujan Paudel, and Assel Aliyeva. I am very grateful to all my collaborators and co-authors, especially Alexander Bulekov and Dr. Babak Amin Azad. I would also like to thank all my friends in the SeclaBU research group.

Finally, I want to thank my parents and my sister for their unconditional love and support and for being the source of inspiration and encouragement my whole life.

SECURING WEB APPLICATIONS THROUGH VULNERABILITY DETECTION AND RUNTIME DEFENSES

RASOUL JAHANSHAHI

Boston University, College of Engineering, 2023

Major Professor: Manuel Egele, PhD
Associate Professor of Electrical and Computer Engineering

ABSTRACT

Social networks, eCommerce, and online news attract billions of daily users. The PHP interpreter powers a host of web applications, including messaging, development environments, news, and video games. The abundance of personal, financial, and other sensitive information held by these applications makes them prime targets for cyber attacks. Considering the significance of safeguarding online platforms against cyber attacks, researchers investigated different approaches to protect web applications. However, regardless of the community's achievements in improving the security of web applications, new vulnerabilities and cyber attacks occur on a daily basis (CISA, 2021; Bekerman and Yerushalmi, 2020).

In general, cyber security threat mitigation techniques are divided into two categories: prevention and detection. In this thesis, I focus on tackling challenges in both prevention and detection scenarios and propose novel contributions to improve the security of PHP applications. Specifically, I propose methods for holistic analyses of both the web applications and the PHP interpreter to prevent cyber attacks and detect security vulnerabilities in PHP web applications. For prevention techniques, I propose three approaches called

Saphire, SQLBlock, and Minimalist. I first present Saphire, an integrated analysis of both the PHP interpreter and web applications to defend against remote code execution (RCE) attacks by creating a system call sandbox. The evaluation of Saphire shows that, unlike prior work, Saphire protects web applications against RCE attacks in our dataset. Next, I present SQLBlock, which generates SQL profiles for PHP web applications through a hybrid static-dynamic analysis to prevent SQL injection attacks. My third contribution is Minimalist, which removes unnecessary code from PHP web applications according to prior user interaction. My results demonstrate that, on average, Minimalist debloats 17.78% of the source-code in PHP web applications while removing up to 38% of security vulnerabilities. Finally, as a contribution to vulnerability detection, I present Argus, a hybrid static-dynamic analysis over the PHP interpreter, to identify a comprehensive set of PHP built-in functions that an attacker can use to inject malicious input to web applications (i.e., injection-sink APIs). I discovered more than 300 injection-sink APIs in PHP 7.2 using Argus, an order of magnitude more than the most exhaustive list used in prior work. Furthermore, I integrated Argus' results with existing program analysis tools, which identified 13 previously unknown XSS and insecure deserialization vulnerabilities in PHP web applications.

In summary, I improve the security of PHP web applications through a holistic analysis of both the PHP interpreter and the web applications. I further apply hybrid static-dynamic analysis techniques to the PHP interpreter as well as PHP web applications to provide prevention mechanisms against cyber attacks or detect previously unknown security vulnerabilities. These achievements are only possible due to the holistic analysis of the web stack put forth in my research.

Contents

List of Tables

List of Figures

List of Abbreviations

ACE	. . . . . .	Arbitrary Code Execution
API	. . . . . .	Application Programming Interface
AST	. . . . . .	Abstract Syntax Tree
CDG	. . . . . .	Class Dependency Graph
CMS	. . . . . .	Content Management System
CSA	. . . . . .	Custom Static Analysis
HTML	. . . . . .	HyperText Markup Language
LHS	. . . . . .	Left Hand Side
LSM	. . . . . .	Linux Security Module
MAC	. . . . . .	Mandatory Access Control
OS	. . . . . .	Operating System
PHP	. . . . . .	PHP: Hypertext Preprocessor
POI	. . . . . .	PHP Object Injection
POP	. . . . . .	Property Object Programming
RCE	. . . . . .	Remote Code Execution
RHS	. . . . . .	Right Hand Side
SQLi	. . . . . .	SQL Injection
VIF	. . . . . .	Vulnerability Indicator Function
XSS	. . . . . .	Cross-site Scripting

Chapter 1

Introduction

E-commerce, social media, and online news draw the attention of billions of users on a daily basis. Importantly, as of 2022, PHP powers 77.4% of all live web sites (Q-Success, 2023b). A further testament to the prevalence of the PHP programming language is the popularity of applications written in PHP. For instance, WordPress alone powers 43.1% of all websites (Q-Success, 2023a). At a high level, a set of software technologies has to work together to process and respond to a user's web requests. This set, which we refer to as the *web stack*, includes the operating system (OS), the web server (e.g., Nginx), the database (e.g., MySQL), and the backend scripting language (e.g., PHP). LAMP (Linux, Apache, MySQL, PHP/Perl/Python) (IBM, 2021) is a popular example of the web stack of open-source building blocks.

The growing number of online users on social networks and online stores makes online platforms an attractive target of sensitive information for attackers. Based on collected data from 550 organizations that encountered data breaches between March 2021 and March 2022, IBM reports that the average cost of a cyber attack for a company was $4.35 million, which is a 10% increase compared to the prior year (IBM, 2022). Exacerbating the situation, Checkpoint's cyber attack trend report shows that global cyber attacks increased by 28% in the third quarter of 2022 compared to the same period in 2021 (Che, 2022). Successful cyber attacks can have a significant impact on both the platform and its users by compromising the integrity of data held by the online platform as well as the operation of the running application.

The most common type of vulnerability leveraged by attackers are injection vulnerabilities (CISA, 2021). OWASP (OWASP, 2023) defines an injection vulnerability as *"a vulnerability which allows an attacker to relay malicious code through an application to another system. This can include compromising both backend systems as well as other clients connected to the vulnerable application."* The root cause of an injection vulnerability is passing insufficiently sanitized attacker-controlled user-input to *sensitive* functions in the vulnerable applications. OWASP's 2017 application security risk report classifies injection vulnerabilities as the most critical type of security risk to web applications (OWASP, 2017), since these attacks can often completely compromise a target application on the online platform.

In order to protect online platforms against attackers, security researchers have investigated a plethora of techniques to identify threats or neutralize the exploitation of security vulnerabilities. Existing approaches on improving the security of online platforms are classified into two groups: 1) detection mechanisms that identify security vulnerabilities, and 2) prevention techniques that provide the necessary tools to protect resources against cyber attacks. In this thesis, I propose novel contributions to improve the security of PHP web applications through both prevention and detection techniques.

1.1 Thesis Contributions

This thesis comprises two synergistic and complementary themes: runtime defenses and vulnerability detection. This thesis proposes to leverage program analysis techniques to address the shortcomings of prior solutions and improve the security of PHP web applications through protection and detection mechanisms. To this end, I introduce a series of novel runtime defenses against SQL injection (SQLi) and remote code execution (RCE) attacks, as well as a semi-automated debloating approach. Additionally, I propose the first systematic and principled approach to identify injection-sinks in the PHP interpreter, which improves existing vulnerability detection systems. My thesis is given in the following statement:

An integrated analysis of the web stack improves the security of web applications in both detection and prevention scenarios.

To substantiate this thesis statement, my Ph.D. research comprises four novel contributions, which include three prevention mechanisms and one detection method. I will first describe my contributions to improving the security of PHP applications through runtime defenses. Next, I elaborate on my contribution toward improving existing tools for detecting security vulnerabilities in PHP applications.

1.1.1 Prevention Mechanisms

Web applications and the interpreted languages that power them are at the core of security breaches that affect society at large. In this subsection, I elaborate on my contributions to runtime defenses against two types of injection attacks: RCE and SQLi. Furthermore, I describe a semi-automated debloating approach to improve the security of PHP web applications by removing vulnerable portions of the source code.

Saphire: Sandboxing PHP applications against RCE attacks

Remote code execution (RCE) vulnerabilities are the most dangerous class of vulnerabilities, where an attacker can completely control the compromised application. The issue that makes RCE so perilous is the fact that modern interpreted applications do not adhere to the principle of least privilege (PoLP). At-risk projects such as QEMU, Chrome, Firefox, and Tor have recognized this issue and reduced the run-time privileges of their software, which limits the potential impact of a vulnerability. However, this approach is not effective on interpreted applications since interpreters such as PHP introduce a layer of abstraction between the program and the underlying OS, which manages the OS resources through an API.

In my first contribution, I introduce an abstraction-aware technique for applying the PoLP to interpreted PHP applications called Saphire. The proposed system treats the capability to issue a system call as a privilege. Thus, the PoLP dictates that each PHP program should *only* be allowed to invoke the system calls that it needs to function correctly. Saphire consists of three essential steps that broadly apply to any interpreted language. In the first step, Saphire analyzes the available APIs for the PHP application and identifies the set of system calls that each API invokes to access the operating system's resources. Next, Saphire analyzes the PHP application to identify the set of APIs used by each PHP file. Combining the above information allows Saphire to generate a system call profile for each PHP file. In the last step, Saphire enforces the system call profile on the executed PHP files. The evaluation of Saphire shows that, on average, my approach reduces the available system calls of each PHP file by 80.5% for the web applications in our dataset. In addition, Saphire successfully defends against 21 previously known vulnerabilities in our dataset with a low 2% overhead on the response time of the web server.

SQLBlock: Mitigating SQL injection attacks against PHP applications

For my second contribution, I focused on SQL injection (SQLi) attacks. A SQLi vulnerability is a type of injection vulnerability where the attacker aims to execute a malicious SQL query on a database. According to a 2020 report from Imperva (Bekerman and Yerushalmi, 2020), 82% of detected injection vulnerabilities are SQLi vulnerabilities. There has been a great deal of research on designing runtime defenses to protect web applications against SQLi attacks (Bandhakavi et al., 2007; Buehrer et al., 2005; Liu et al., 2009; Medeiros et al., 2016; Medeiros et al., 2019; Sun and Beznosov, 2008; Boyd and Keromytis, 2004; Halfond and Orso, 2005; Naderi-Afooshteh et al., 2015). However, the shortcomings of existing systems, such as relying on an incomplete definition of SQLi attacks (Ray and Ligatti, 2012) and coarse-grained profiles of benign SQL queries (Medeiros et al., 2016; Medeiros et al., 2019), leave popular web applications such as WordPress, Joomla, and

Drupal vulnerable to SQLi attacks. Furthermore, overly strict approaches (Medeiros et al., 2019) result in false positives (i.e., the application rejects valid user interaction).

As a novel contribution, I proposed SQLBlock, a hybrid static-dynamic approach that protects web applications against SQLi attacks. To this end, SQLBlock first statically analyzes the PHP web application to identify the database APIs that are used across the web application to issue queries to the database. This step allows SQLBlock to determine the database API and subsequently identify the PHP functions that use this API to communicate with the database. Second, SQLBlock records the set of issued queries during benign browsing of a given web application. In the next step, SQLBlock generates a profile, which is a mapping between the function that issued the query and a query descriptor that characterizes the benign functionality of the SQL query. Finally, SQLBlock enforces the profile inside the database and rejects SQL queries that do not match the profile. The evaluation of SQLBlock confirms that it protects against real-world SQLi attacks in our dataset.

Minimalist: Debloating PHP applications through static analysis

The growth in features and capabilities of web applications involves the constant introduction of new code. This ever-increasing codebase can be partially explained by the increasing reliance on third-party libraries and frameworks that facilitate the development process. Developers include an entire library in the web application while only using a small portion of the code from the library. Concurrently, users do not always use the entirety of introduced features, leading to another source of unnecessary code. In this thesis, we refer to unnecessary code, independent of its source (e.g., libraries or user behavior), as *bloat*.

Debloating is the process of determining the functionality that a user or system requires to fulfill its purpose and subsequently preventing the execution of all other code in the system (e.g., a web application). An essential aspect of debloating is determining what code to remove vs. what code to retain, which can be done dynamically (e.g., through reliance on dynamic traces (Abubakar et al., 2021; Azad et al., 2019; Bulekov et al., 2021; Jahanshahi et al., 2020)) or statically (e.g., through static call-graph construction (Agadakos et al., 2019; Ghavamnia et al., 2020a; Qian et al., 2020) or dependency graphs (Koishybayev and Kapravelos, 2020)). Dynamic debloating techniques require a training phase that records execution traces to determine the used portion of code in a web application, which then allows dynamic approaches to reject all behaviors that were not observed during training. However, profiling user interaction is a resource-intensive task, which can hinder the server's performance in terms of response time. For instance, my evaluation shows that Less is More (LIM), a state-of-the-art dynamic debloating technique, slows down the server's response time by 17X in web applications like WordPress. Furthermore, a key goal for both static and dynamic approaches is minimizing the false positives in a debloated web application (i.e., the incorrect removal of required functionality). Achieving a low number of false positives for a debloating mechanism is an important yet challenging task since false positives make a debloated application unresponsive for legitimate users. My evaluations demonstrate that by adding small variations in already exercised features in a web application (e.g., changing an option in a drop-down list on a submitted form), users observe a breakage in 33% of their actions for a web application debloated by LIM. The drawbacks of existing systems point out the necessity of an analysis that avoids false positives from debloated applications by not relying on dynamic code coverage.

To address the shortcomings of prior work in debloating PHP web applications, I propose a semi-automated static debloating method called Minimalist for web applications written in PHP. Minimalist generates a call-graph for a given web application, which is then used for the debloating process. At a high level, my debloating scheme consists of three major steps. First, Minimalist statically analyzes the given PHP web application to generate a call-graph. Second, Minimalist prunes the call-graph by removing the functions that the web application does not require to respond to users' requests. Finally, Minimalist performs a function-level debloating to remove the unused functions. The evaluations show that Minimalist debloats 17.78% of code in 12 web applications and removes 38% of high-severity vulnerabilities in our dataset.

1.1.2 Detection Mechanism

As mentioned before, injection vulnerabilities (e.g., SQLi, insecure deserialization, or cross-site scripting) are the most common category of application vulnerabilities in web applications. To exploit such injection vulnerabilities, attackers provide malicious input to the web application, compromising both the backend systems as well as the clients. In this novel contribution, we focus on two types of injection vulnerabilities: cross-site scripting (XSS) and insecure deserialization. An XSS vulnerability allows an attacker to execute malicious JavaScript on the client's browser and compromise the interaction between the client and the vulnerable website. In addition, an insecure deserialization vulnerability arises when an attacker can inject malicious serialized data to a vulnerable application, which allows the attacker to reuse existing application code in harmful ways. The root cause of an injection vulnerability is passing insufficiently sanitized attacker-controlled user-input to *sensitive* APIs. Depending on the injection vulnerability type, the sensitive APIs differ. For instance, PHP's `echo` is recognized as a sensitive API for XSS attacks by the community, while `unserialize` plays the same role for insecure deserialization. Despite the differences in their details, injection vulnerabilities are data-flow problems where untrusted user-input is propagated throughout the web application's execution and finally reaches sensitive sinks (e.g., `echo` or `unserialize`). Despite the effectiveness of prior work in detecting security vulnerabilities, all existing systems share one common flaw: *relying on a manually curated or historically defined list of sensitive APIs to detect injection vulnerabilities.* The accuracy of a manually curated list of sensitive APIs depends on the documentation for the programming language and the expertise of the analyst who identifies the sensitive APIs. As is often the case with human involvement, the listings are not comprehensive, which leads to undetected injection vulnerabilities.

To address this shortcoming, I propose a systematic and principled approach to infer a comprehensive set of APIs that can lead to XSS and insecure deserialization. To this end, I designed, implemented, and evaluated an automatic approach called Argus to identify the set of APIs that deserialize user-input or write to the output buffer (e.g., the HTML response for the user). A complete set of sinks offers significant security benefits. I demonstrate that by incorporating Argus' resulting sink list into existing systems, those systems produce significantly higher-quality results. My key observation is that there is precisely *one* function inside the PHP interpreter that deserializes data – `php_var_unserialize`. A similar observation shows that there is one function used for writing to the output buffer called `php_output_write`. Argus detects the set of injection-sinks through an automated hybrid static-dynamic analysis of the PHP interpreter. Specifically, Argus first generates the call-graph for the PHP interpreter. Next, Argus performs a reachability analysis to identify the set of PHP APIs whose invocation can trigger deserialization or writing to the output buffer. Using this approach, Argus detects 284 deserialization APIs, an order of magnitude more than the most exhaustive manually curated list used in related works. Furthermore, Argus also detected 22 output APIs in PHP 7.2, which is twice the number of sensitive APIs used in existing detection systems. To demonstrate the benefits of this approach, I extended two existing vulnerability detection systems (i.e., Psalm and FUGIO) to identify and exploit injection vulnerabilities such as XSS and insecure deserialization. The extension of existing program analysis tools led to the identification of 13 previously unknown vulnerabilities.

In summary, I introduced a series of novel protection and detection methods to improve the security of PHP web applications. In protection scenarios, I designed and developed three approaches to defend against RCE (Saphire) and SQLi attacks (SQLBlock), as well as a semi-automated static debloating approach (Minimalist). Finally, as a detection mechanism, I propose a systematic and principled analysis of the PHP interpreter (Argus), which identifies the set of PHP APIs that can lead to XSS and insecure deserialization vulnerabilities. Overall, these contributions demonstrate how holistic analysis of the web stack provides more protection against various attacks against PHP web applications as well as the detection of previously unknown security vulnerabilities.

Chapter 2

Background

In this section, I provide background knowledge on the resource management of the operating system (OS) through system-call, the PHP language, and the threat of vulnerabilities in PHP web applications. I look into the PHP interpreter and review the language constructs that complicate the analysis of PHP web applications in order to generate a call-graph. Furthermore, I elaborate on the threat of SQL injection (SQLi), remote code execution (RCE), cross-site scripting (XSS), and insecure deserialization. These factors motivate the design of my runtime defenses as well as program analysis techniques, which I elaborate in the next chapters.

2.1 System Calls

An operating system manages the resources on a computer and provides user-space processes with mediated access to these resources via the system-call API. The available system-calls and how they are invoked depend on the OS and the processor's instruction set architecture. Programs rely on system-calls as they are the only means of communicating with the rest of the system and with the outside world. Programs and payload code alike can only communicate with the process' environment through the system-call API.

Recognizing that the system-call API is a key interface that is used by both benign and compromised processes, operating systems provide methods for limiting the system-calls accessible to an attacker who has exploited a process. For example, with Linux' `seccomp`, a process can provide the kernel with a filter, which the kernel uses to decide which system-calls to allow from the process in the future. Once the filter is installed, it is enabled for the lifetime of the process, and it is not possible to remove restrictions. If a process calls a filtered system-call, the kernel kills the process. `seccomp` is used for sandboxing major client and server software, including Chrome, Firefox, Tor, QEMU, and OpenSSH. Since the kernel performs the filtering, the overhead is negligible.

Linux also supports security modules (LSM), such as AppArmor and SELinux, which add support for access control policies, including mandatory access control (MAC). MAC rule-sets can be used to explicitly limit the "capabilities" of a program, such as access to a network or specific files. Security modules allow an administrator to manually secure a process if its interactions with the OS are well-defined. However, LSMs cannot distinguish between individual scripts executed by an interpreter. Hence, it is difficult to build a MAC rule-set for an interpreter while enforcing the PoLP for individual interpreted programs.

2.2 Interpreters

Interpreted programs rely on a separate application - the interpreter, for execution. Separating the binary code in the interpreter from the actual program makes code portable and allows for straightforward implementation of advanced language features, such as reflection and dynamic-scoping. Essentially, the interpreter is a layer of abstraction, separating the program code from the low-level details of the underlying operating system. Since interpreted programs still need access to system resources (e.g., files or network sockets), they must have a means of communicating with the kernel. To bridge the gap created by abstraction and provide programs with access to OS-managed resources, interpreters provide an API to the programs they execute.

For example, the PHP interpreter's built-in API provides access to the file-system, network, and databases, as well as unprivileged resources such as built-in data structures and string-operations. When an API feature requires access to OS-managed resources, such as network sockets or file descriptors, the interpreter issues a system-call, which is handled by the kernel. Interpreted programs can define functions in their code, but unlike the interpreter API, these functions can never directly issue system-calls, as this would break the interpreter abstraction.

2.3 PHP

PHP is an open-source server-side scripting language. According to W3Techs (Q-Success, 2023b), 79.1% of all websites use PHP as their server-side language. The PHP interpreter provides various features to develop PHP web applications. In this section, we go over the common source of dynamicity in the control flow of PHP web applications.

2.3.1 Object Oriented Programming

PHP supports the Object-Oriented Programming model, which introduces three new concepts for developing PHP web applications: inheritance, polymorphism, and encapsulation. Inheritance and polymorphism let developers extend the functionality of classes or implement an interface in more than one way. Encapsulation bundles data and methods into a single unit. Hence, OOP allows developers to create modular programs and extend the functionality of PHP database extensions. Additionally, PHP provides dynamic features, such as creating objects from dynamic strings. `new` is the keyword for creating objects from a class in PHP. The argument for the `new` keyword can be a class name or a string that represents the name of the class. An example is shown in Figure 2.1, line 22, where the value of `type` defines the class that should be instantiated.

```
1  ## Class.php
2  class ParentClass {
3      public $feature = 0;
4      public function __construct() {
5          $this->feature = 1;
6      }
7      public function Cprint(){
8          echo $this->feature."\n";
9      }
10 }
11 class ChildClass extends ParentClass {
12     public function call() {
13         call_user_func(array($this, 'Cprint'));
14         }  └──→ Invoke Cprint in ParentClass
15 }
16
17 ## test.php
18 define('classpath', __DIR__ );
19 $included = classpath."/Class";
20 include_once $included.'.php';
21 $type = "ChildClass";      └→ Variable file inclusion
22 $obj = new $type;  ↘
23 $method = "call";     Invoke the parent constructor
24 $obj->$method(); ──→ Variable invocation
```

Listing 2.1: Usage of dynamic PHP language constructs in file inclusion, class instantiation and function calls.

2.3.2 File Inclusion Schemes

Program dependencies are a prominent feature in many interpreters. Dependencies allow developers to organize and reuse code and encourage good software engineering practices. PHP provides two mechanisms for file inclusion: 1) Direct inclusion using `include` and `require` statements, and 2)`autoloaders`.

Direct file inclusion enables developers to load PHP files corresponding to different classes and modules at runtime. Lines 18 and 19 in Listing 2.1 incorporate a constant variable definition based on the path of the current directory (using the `__DIR__` built-in constant) to generate the file path that is then used in the include statement on line 20. This file inclusion scheme is commonly used in applications such as WordPress and phpMyAdmin. In order to statically resolve such file paths, an application analysis needs to properly models the data flow (e.g., direct variable assignments, use of arrays, constants, and global variables) for the target variables.

Autoloaders allow developers to dynamically resolve and load undeclared classes without explicit calls to `include` or `require`. A PHP application can introduce autoloading rules to the PHP interpreter using the `spl_autoload_register`. This way, the PHP interpreter can automatically use the defined rules to load undeclared classes. In Listing 2.1, autoloaders could be used instead of direct file inclusion on line 20. This way, the PHP interpreter would automatically include the `Class.php` file inside `test.php` on line 22 when the class instantiation occurs with an undefined class name. Regardless of the file inclusion mechanism, the PHP engine executes all the code in the main body (i.e., not part of a function or a class) of the included PHP script upon inclusion.

2.3.3 Function Invocation

PHP provides various language APIs to invoke functions (e.g., direct invocation, variable function names, and callbacks). A dynamic function call refers to a function invocation where the function to be invoked is determined at runtime. Next, we go over the PHP language APIs and constructs that can result in dynamic function calls:

Reflection in the PHP interpreter allows programs to examine classes, interfaces, methods, and extensions. PHP applications can use reflection to dynamically instantiate a class, list available methods and properties, and invoke class methods.

Variable functions in PHP are an implicit way of calling functions using reflection. In this scheme, the target function name is stored in a variable, which is then used in a function invocation. Lines 23 and 24 of Listing 2.1 demonstrate this use case, where the variable `$method` is assigned with the function name `call`. Likewise, the class name that implements this method is defined on line 21 by a variable named `$type`.

Callback functions are another means of invoking functions. This scheme is commonly used to delay the invocation of a function. Certain PHP built-in functions, such as *array_filter(), call_user_func(), and set_error_handler()*, accept a function name to be called after a certain event. Line 13 in Listing 2.1 showcases a function named "Cprint" defined under $this scope, which refers to *ChildClass* being invoked using a callback scheme via *call_user_func()*.

2.3.4 Deserialization

As part of the PHP API, the PHP interpreter provides a native implementation of serialization through two PHP API functions – `serialize` and `unserialize`. Serialization provides an easy mechanism to persist data in memory or storage. Serialization also allows applications to communicate and transfer data to each other by converting the data to a stream of bytes. The receiver of serialized data must follow a process called deserialization, which converts the byte-stream back into PHP objects. Importantly, the serialized byte-stream can only contain data; code, which defines the functionality and behavior of objects, is not included. To deserialize a byte-stream, PHP provides the `unserialize` API function. The PHP interpreter takes the following steps to deserialize an object:

- The PHP interpreter instantiates an object of the specified class and assigns its properties based on the byte-stream contents. Using this approach, the PHP interpreter creates a copy of the originally serialized object. Note that in order to instantiate objects via deserialization, their corresponding class structure and method implementation must already be defined in the program.
- Once the object is created, the PHP interpreter invokes the `__wakeup` function if it is implemented by the corresponding class. This function recreates any resources that the object contains. This functionality can be used to restore database connections,

file handlers, etc.

While executing a PHP script, the PHP interpreter destroys the object when there are no references to the object by invoking the __destruct function. Deserializing an input is safe as long as the input is properly sanitized. Passing malicious data to unserialize in PHP can lead to a diverse set of attacks, such as denial of service and arbitrary code execution.

2.3.5 Stream Wrappers

A stream in the PHP interpreter is a generalization of a data source which implements a set of common file operation functions such as fopen and copy. PHP Stream wrappers allow developers to use consistently-named file-related functions such as fopen for different types of file resources. The types of resources are identified analogous to URL schemes and can vary from classic local files (e.g., /etc/passwd), network reachable resources (e.g., https://example.com/text), to PHAR archive types (e.g., phar://usr/share/app.phar). Importantly, once the resource's type is identified, the PHP interpreter, maps each type to a corresponding stream wrapper which allows the application developer to transparently perform (supported) file operations on the resource (e.g., read, seek, etc.).

PHP Archives (phar) enable developers to compress an entire PHP application in a single file. To interact with phar files, PHP provides a built-in stream wrapper. Each phar file contains the following sections:

- **Stub** is a simple PHP file that instructs the PHP interpreter how to load the application.
- **Manifest** determines the number of files in the phar as well as the file permissions, type of compression, and serialized metadata. The metadata includes a description of the existing files in the archive in a serialized format.

- **Contents** includes the actual contents of all of the files in the phar archive.
- **Signature** is an optional approach to verify the file's integrity.

2.4 PHP Web App Vulnerabilities

In this section, we discuss four vulnerabilities that arise in PHP web applications: 1) SQL injection, 2) php object injection, 3) remote code execution, and 4) cross-site scripting.

2.4.1 SQL Injection

SQL injection (SQLi) is a code injection attack in which an attacker is able to control a SQL query to execute malicious SQL statements to manipulate the database. SQLi attacks are classified into eight categories (Halfond et al., 2006; Dahse and Holz, 2014b):

1. ***Tautologies:*** The attacker injects a piece of code into the conditional clause (i.e., where clause) in a SQL query such that the SQL query always evaluates to true (Halfond et al., 2006). The goal of this attack varies from bypassing authentication to extracting data, depending on how the returned data is used in the application.
2. ***Illegal/Logically incorrect Queries:*** By leveraging this vulnerability, an attacker can modify the SQL query to cause syntax, type conversion, or logical errors (Halfond et al., 2006). If the web application's error page shows the database error, the attacker can learn information about the back-end database. This vulnerability can be a stepping stone for further attacks by revealing the injectable parameters to the attacker.
3. ***Union Query:*** In union query attacks, the attacker tricks the application to append data from the tables in the database for a given query (Halfond et al., 2006). An attacker adds one or more additional *SELECT* clauses, which start with the keyword

UNION, that lead to merging results from other tables in the database with the result of the original SQL query. The goal of such an attack is to extract data from additional tables in the database.

4. ***Piggy-backed Query:*** A piggy-backed query enables attackers to append at least one additional query to the original query. Therefore, the database receives multiple queries in one string for execution (Halfond et al., 2006). The attacker does not intend to modify the original query but to add additional queries. Using the piggy-backed query, an attacker can insert, extract, or modify data, as well as execute remote commands and extract data from the database. The success of the attack depends on whether the database allows the execution of multiple queries from a single string.

5. ***Stored procedures:*** Stored procedures are a group of SQL queries that encapsulate a repetitive task. Stored procedures also allow interaction with the operating system (Halfond et al., 2006), which can be invoked by another application, a command line, or another stored procedure. While a database has a set of default stored procedures, the SQL queries in a stored procedure can be vulnerable, similar to SQL queries outside the stored procedure.

6. ***Inference:*** In this type of attack, the application and the database are prevented from returning feedback and error messages; therefore, the attacker cannot verify whether the injection was successful or not (Halfond et al., 2006). In inference attacks, the attacker tries to extract data based on answers to true/false questions about the data already stored in the database.

7. ***Alternate Encoding:*** In order to evade detection, the attackers use different encoding methods to send their payload to the database. Each layer of the application deploys various approaches for handling encodings (Halfond et al., 2006). The difference between handling escape characters can help an attacker evade the application layer

and execute an alternate encoded string on the database layer.

8. ***Second order injections:*** One common misconception is that the data already stored in the database is safe to extract (Dahse and Holz, 2014b). In a second order attack, an attacker sends his crafted SQL query to the database to store his attack payload in the database. The malicious payload stays dormant in the database until the database returns it as a result of another query, and the malicious payload is insecurely used to create another SQL query.

2.4.2 Remote Code Execution

Remote code execution (RCE) occurs when a network attacker gains the ability to execute arbitrary code (ACE) on a target system. RCE exploits against interpreted programs generally rely on improper usage of language features. Notably, RCE exploits commonly rely on code injection, insecure deserialization, or unrestricted file uploads.

Once the attacker exploits an RCE vulnerability, they leverage the exploited process to run a payload to, generally, gain access to additional resources. The operating system provides a system-call interface that processes must use to access privileged resources, such as the network, file system, and process-management. Therefore, in order for the attack to be fruitful, the payload must invoke system-calls to access resources managed by the OS. For example, a simple payload may try to expose a shell that the attacker can connect to remotely. Such a payload requires, at minimum, access to the network and process management to spawn a shell process.

2.4.3 PHP Object Injection

PHP object injection (POI) is a security vulnerability that leverages insecure deserialization in PHP applications. To exploit such a vulnerability, an adversary must control the properties of an insecurely deserialized object. By exploiting a POI vulnerability, an attacker can potentially hijack the program's execution by controlling the properties used in automatic calls to the `__wakeup` and `__destruct` methods.

The snippet in Listing 2.2 presents a PHP script that contains a deserialization vulnerability. We observe that at Line 25, user-input is passed to the `unserialize` function without sanitization. In order to exploit this vulnerability, the attacker needs to satisfy two conditions.

- There needs to be at least one class which implements the class methods `__wakeup` or `__destruct` to carry out the attack.
- All of the classes used in the exploit need to be defined (or the application must support automatic loading of classes) when the `unserialize` function is called on Line 25.

```
class Example
{
 protected $obj;
 function __construct()
 {
  // some PHP code ...
 }
 function __destruct()
 {
  if (isset($this->obj))
  {
   return $this->obj->getValue();
  }
 }
}
class Exec
{
 private $_cmd;
 function getValue()
 {
  system($this->_cmd);
 }
}
// unsantizied user-input passed to unserialize
$user_data = unserialize($_POST['data']);

// unsanitized file operation
file_exists($_POST['file']);
```

Listing 2.2: A deserialization vulnerability leading to arbitrary code execution. An adversary can execute any command by crafting a PHP object which modifies the value of `_cmd` property.

Exploiting a POI vulnerability is inherently a code-reuse attack, where an attacker simply recombines the already existing code to achieve malicious outcome by introducing a malicious object. To exploit a POI vulnerability the attacker needs to identify the user-defined functions and methods (i.e., gadgets) in the PHP app that can be used to achieve his goals (Park et al., 2022). As an example, we describe how an attacker can choose the gadgets to link and perform a remote code execution attack in Listing 2.2. Looking at Listing 2.2, we observe that there are two defined classes prior to the deserialization at Line 25: `Example` and `Exec`. The destructor of class `Example` calls a function named `getValue` from the variable `obj`. If an attacker sets the variable `obj` to an object of class `Exec`, then the destructor will call the class method `getValue` at 12. Looking at the implementation of the class method `getValue`, we see the invocation of the function `system` on the property of `_cmd`. An attacker can run an arbitrary command by setting the value of the `_cmd` property. The first part of Listing 2.3 contains the exploit written for the vulnerability in Listing 2.2.

2.4.4 Exploiting phar wrappers.

Thomas in (Thomas, 2018) demonstrated how an attacker can exploit an invocation of a file operation API and perform a PHP object injection. He showed that the PHP interpreter deserializes the metadata upon any file operation on a phar file. Considering the aforementioned information, an adversary can achieve arbitrary code execution by leading the PHP interpreter to perform file operations (e.g., `file_exists`) on a phar file with a malicious metadata field.

The second part of Listing 2.3 shows how an attacker can generate a phar file with malicious metadata (set on Line 20). Looking at the snippet in Listing 2.2, we observe that the PHP script checks the existence of a file by passing an unsanitized user-input at line 28. In order to exploit the vulnerability at Line 28 of Listing 2.2, the attacker can set the post variable `file` to `phar://path-to-malicious-phar-file`.

```
// PART ONE: modify properties
class Exec
{
   private $_cmd = "cat secret";
}
class Example
{
   protected $obj;
   function __construct()
   {
      $this->obj = new Exec;
   }
}

print urlencode( serialized( new Example ) );

// PART TWO: create Phar file
$phar = new Phar('exploit.phar');
$phar->startBuffering();
$phar->setMetadata(new Example());
$phar->stopBuffering();
```

Listing 2.3: Adversary can exploit file operations by generating a malicious phar file.

2.4.5 Cross-site Scripting

A cross-site scripting (XSS) vulnerability is an injection vulnerability that allows an attacker to compromise the interactions of the victim with a vulnerable application. This vulnerability allows the attacker to execute malicious scripts in the victim's web browser by including malicious code in a legitimate web application. In order to understand the cause of an XSS vulnerability, we first explain the life-cycle of a request sent by a user and then discuss the role of the PHP interpreter in this life-cycle.

In the life-cycle of a request sent to a web-server such as Apache or Nginx, the PHP interpreter plays an important role in providing the output shown to the user. When a web-server receives a request for a PHP script, the web-server invokes the PHP interpreter to determine the output. The PHP interpreter executes the PHP script, which can include interaction with a database, the file system, or the underlying operating system. After the execution of the script, the PHP interpreter provides the output in the form of HTML to the web-server, which is then sent back to the user as the response.

As mentioned above, the PHP interpreter determines the response that the web-server sends back to users. In order for a PHP script to modify the response, the PHP interpreter provides a set of built-in functions (i.e., PHP API), which PHP scripts can use. One of the APIs that is used to modify the response of a web-server is `echo`. This function accepts one or more strings, which are then sent to the output buffer, which in this case is the response of a web-server. However, the set of APIs that are able to modify the response of a web-server is not limited to only one API. According to prior work such as RIPS (Dahse and Holz, 2014a), there are 12 functions in the PHP interpreter that can modify the output buffer.

In a cross-site scripting attack, the attacker is able to modify the response that is sent back to the user's browser. If an attacker has control over the arguments passed to an API such as `echo`, an XSS attack is a certainty. This capability of the `echo` API is provided by an internal function of the PHP interpreter, which allows APIs to write into the output buffer (i.e., the HTML response). An analysis on the source-code of the PHP interpreter reveals that the function called `php_output_write` performs the write operation to the output buffer. As a result, any PHP API that invokes `php_output_write` can modify the response sent back to the web-server, which is the superset of all the APIs identified in prior work.

Chapter 3

Related Work

There are a significant number of studies on the security of PHP web applications. In this chapter, I discuss prior work on PHP web application security. I categorized the related work into prevention and detection techniques. In particular, I elaborate on debloating techniques as well as runtime defenses against RCE and SQLi attacks. Next, I detail the work on detection of vulnerabilities in PHP web applications, including deserialization vulnerabilities.

3.1 Prevention Techniques

Prevention approaches provide the means to defend PHP web applications against different types of attacks. Prior work in this area focused on preventing attacks based on vulnerabilities in PHP web applications using runtime defenses and debloating techniques. In this section, I discuss existing runtime defenses against SQLi and RCE attacks. Next, I elaborate on debloating techniques and how they protect web applications against malicious attacks.

3.1.1 Runtime Defenses Against SQLi Attacks

In this subsection, I review the relevant literature on defending web applications against SQLi attacks. I also compare SQLBlock with five existing approaches and explain why prior systems are not sufficient for PHP web applications that utilize OOP to communicate with databases.

Tool	Taut.	Illegal/Incorrect	Union	Piggy-back	Stored proc.	Infer.	Alt. encoding	Second order inj.
SQLrand (Boyd and Keromytis, 2004)	●	○	●	●	○	○	○	○
SQLCheck (Su and Wassermann, 2006)	◑	◑	◑	◑	○	◑	◑	○
Merlo et. al. (Merlo et al., 2007)	◑	○	◑	◑	◑	◑	◑	○
SEPTIC (Medeiros et al., 2019)	◑	○	◑	◑	◑	◑	◑	◑
DIGLOSSIA (Son et al., 2013)	●	●	●	●	○	●	○	○
SQLBlock	●	○	●	●	●	●	●	●

Table 3.1: Comparison of SQLBlock with other techniques with respect to SQLi attack type. SQLBlock provides the most effective protection.

The comparison between prior work and SQLBlock based on the SQLi attack type is presented in Table 3.1. For each SQLi attack type in Table 3.1, ● means the tool can defend against the type of attack, ○ means the tool is ineffective, and ◑ means that the tool can partially defend the web application against a SQLi attack. Partially defending means that either the tool can only defend web applications that do not use OOP for implementing communication with the database or the definition of SQLi attacks in the tool is incomplete.

In general, runtime defenses track user-inputs (Sun and Beznosov, 2008; Buehrer et al., 2005; Su and Wassermann, 2006; Bandhakavi et al., 2007) or build a profile of benign SQL queries (Medeiros et al., 2019; Son et al., 2013; Medeiros et al., 2016; Merlo et al., 2007) to prevent SQLi attacks on web applications. SQLPrevent (Sun and Beznosov, 2008) analyzes generated queries for the existence of HTTP request parameters and raises an alert when an HTTP request parameter modifies the syntax structure of a query. SQLGuard (Buehrer et al., 2005) proposed a dynamic approach for comparing the parse tree of issued queries at runtime before and after the inclusion of user inputs. SQLGuard needs to modify the source code of the web application. WASP (Halfond et al., 2008) proposes a taint analysis to detect SQLi attacks on web applications. CANDID (Bandhakavi et al., 2007) records a set of benign SQL queries that the web application can issue by instrumenting the web application's source code and dynamically executing the SQL statements with benign inputs. CANDID, SQLGaurd, WASP, and SQLPrevent assume that if the input does not change the syntax structure of a SQL query, then a SQLi attack has not occurred. Such an assumption can leave the web application vulnerable to SQLi attacks and also block benignly generated queries (Ray and Ligatti, 2012). Unlike CANDID, SQLGaurd, WASP, and SQLPrevent, SQLBlock does not detect SQLi attacks based on the modification to the syntax structure of the SQL query. SQLBlock generates a set of query descriptors for benign queries that each PHP function issues to the database. Furthermore, SQLBlock allows functions in the web application to issue queries, as long as the query matches its query descriptors. Beyond this, SQLBlock does not need to modify the source code of the web application for its operation.

SQLCheck (Su and Wassermann, 2006) tracks user-inputs to SQL queries and flags a SQL query as an attack if user-input modifies the syntactic structure of the SQL query. This incomplete definition of SQLi attacks prevents SQLCheck from defending against tautology, inference, stored-procedure, and alternate encoding attacks. These four attacks do not necessarily modify the syntax structure of a SQL query. Considering these weaknesses, SQLCheck cannot protect web applications against any of the vulnerabilities in the evaluation dataset mentioned in Table 3.1.

Diglossia (Son et al., 2013) proposed a dual parser as an extension to the PHP interpreter. Diglossia maps the query without user-input to a shadow query, and then it checks whether the parse tree of the actual query and the shadow query are isomorphic or not. If both parse trees are isomorphic and the code in the shadow query is not tainted by user-inputs, Diglossia passes the query to the back-end database. Diglossia is unable to defend against second-order injection since Diglossia only checks queries with user-inputs. Moreover, Diglossia cannot detect alternate encoding and stored-procedure attacks since these attacks do not modify the parse tree of the SQL query (Medeiros et al., 2019). As shown in

Table 3.1, SQLBlock defends web applications against more variants of SQLi attacks than Diglossia.

SEPTIC (Medeiros et al., 2019) creates a profile for each issued query during the training phase and enforces this profile to protect web applications against SQLi attacks. During the training, SEPTIC creates a query model that includes all the nodes in the parse tree of a SQL query. The profile in SEPTIC is a mapping between the query model and an ID. The ID is the sequence of functions that pass the query as an argument. During the enforcement, SEPTIC uses this sequence of functions as an identifier and finds the appropriate query model in the profile. If the issued query matches the query model in the profile, SEPTIC allows the database to execute the query. Enforcing a profile based on the exact model of the generated queries that includes the name of table columns and number of SQL functions prevents web applications from generating dynamic yet benign SQL queries, which causes false positives in SEPTIC. For instance, assume there is a webpage for searching for published music albums, and users can search based on the name of an album, an artist's name, or the release year. If SEPTIC is trained with SQL queries that only include the album's name or the release year, it rejects any SQL queries from a user that search using the artist's name. SQLBlock solves this problem by creating query descriptors for SQL queries. Query descriptors generalize the benign SQL queries, which allow the web application to produce a range of dynamic queries.

Furthermore, to create an identifier for each issued query in the profile, SEPTIC uses the information in the PHP call-stack that issued the call to methods from *mysql* or *mysqli*. SEPTIC checks the sequence of functions in the PHP call-stack for the presence of a SQL query in the function's arguments. Since OOP web applications do not pass the SQL query as an argument, SEPTIC cannot generate a correct identifier for SQL queries. Instead, it creates the same identifier for all the issued queries in the OOP web application. Consequently, an attacker can use a vulnerable function in the web application to issue any query from the profile. Considering the coarse-grained mapping that SEPTIC builds for the web applications that use OOP, SEPTIC can defend against only four variants of SQLi attacks in our dataset. All four SQLi attacks that SEPTIC can defend against reside in WordPress. Wordpress does not use the encapsulation concept in its database API, and its modules provide SQL queries as function arguments; consequently, SEPTIC can correctly create its mapping. SQLBlock overcomes this problem by utilizing a static analysis that identifies the database API in the web application, which helps SQLBlock correctly determine the function that interacts with the database.

Merlo et al. (Merlo et al., 2007) proposed a two-step approach. First, it intercepts every function call to `mysql_query` and records a profile for benignly issued SQL queries. The profile is a mapping between the issued SQL query and the PHP function that calls `mysql_query`. During enforcement, the proposed approach in (Merlo et al., 2007) looks for the received SQL query in its mapping profile, and if the query does not syntactically match with any recorded query for the PHP function, the approach blocks the query. This approach maps all the SQL queries to the internal functions in the database API instead of the appropriate function that uses the database API for communicating with the database. Besides that, enforcing a strict comparison of the parse tree limits the functionality of the web application for generating dynamic SQL queries. SQLRand (Boyd and Keromytis, 2004) proposed a randomization technique for randomizing queries in web applications. SQLRand randomizes the SQL queries in the web application and uses an intermediary proxy for de-randomizing before sending the queries to the database. Since web applications generate SQL queries dynamically, randomizing the queries using SQLRand is a challenging task. Using an intermediate proxy introduces an overwhelming overhead to web application performance (Halfond et al., 2006; Su and Wassermann, 2006). Besides that, since there is one static key that modifies SQL keywords, knowledge of new SQL keywords can compromise the security of SQLRand (Buehrer et al., 2005).

3.1.2 Mitigation techniques against RCE

As system-calls guard access to sensitive OS-managed resources, there is abundant research related to system-call based sandboxing, focused on restricting resources available to an application (Kim and Zeldovich, 2013; Goldberg et al., 1996; Provos, 2003; Wagner, 1999; Jain and Sekar, 2000; Forrest et al., 1996; Garfinkel et al., 2004; Seaborn, 2019), intrusion detection systems (Hofmeyr et al., 1998; Wagner and Dean, 2001; Somayaji and Forrest, 2000; Sekar et al., 2001; Mutz et al., 2006; Zheng and Huang, 2018), and confining Linux containers (Wan et al., 2017; Lei et al., 2017; Jamrozik et al., 2016).

Janus (Wagner, 1999) relies on system-call interposition with *ptrace* to intercept and filter dangerous system-calls, according to defined policies. Plash (Seaborn, 2019) restricts a process by executing it in a chroot environment with a set of instrumented system-calls, relying on an RPC server. Systrace (Provos, 2003) generates system-call policies interactively, with input from the user. Systrace requires the user to manually modify the policies for applications that pass non-deterministic arguments to system-calls (Provos, 2003). Ostia (Garfinkel et al., 2004) and REMUS (Bernaschi et al., 2002) rely on user-specified rules to filter system-calls.

Unlike Saphire, which is completely automatic, existing systems (Wagner, 1999; Provos, 2003; Garfinkel et al., 2004; Seaborn, 2019; Bernaschi et al., 2002) require user involvement in profile generation. *N*-gram-based allowlists (Forrest et al., 1996; Sekar et al., 2001; Wagner and Dean, 2001) make decisions based on whether sequences of system-calls were observed during benign execution, but rely on representative sets of benign executions. Janus, Systrace, Ostia, and the *N*-gram approaches incur significant overhead, which makes them impractical (Linn et al., 2005). Unlike Saphire, prior filtering approaches do not tailor system-call profiles to individual interpreted programs. Since interpreted programs are prime targets for attackers today, this is a major limitation.

SELinux (Loscocco and Smalley, 2001) leverages role-based access control and multi-level security to implement mandatory access control and enforce restrictions on data for user roles. AppArmor (Cowan et al., 2000) restricts a program's access to files and capabilities according to a profile. Both SELinux and AppArmor require an administrator to manually specify, or dynamically collect a security profile, which is non-trivial. AppArmor and SELinux's protection cannot distinguish between the execution of different programs by an interpreter. FMAC (Prevelakis and Spinellis, 2001) creates an access profile based on benign inputs to a program and uses it to deny or restrict access to files. MAPbox (Acharya and Raje, 2000) allows a user to manually specify a list of acceptable application behaviors, each of which corresponds to a sandbox configuration. Boxmate (Jamrozik et al., 2016) confines an Android app to the set of resources it accessed during a training stage. Boxmate blocks any access to resources that were not accessed during training. Wan et. al (Wan et al., 2017) extend Boxmate to Linux containers by recording a list of the accessed system calls during automatic testing and using this as an allowlist for filtering system-calls in Linux containers. Boxmate (Jamrozik et al., 2016) and (Wan et al., 2017) confine processes and are not fine-grained enough to identify the execution of different scripts by an interpreter.

3.1.3 Debloating Techniques

The application of software debloating to vulnerability reduction has recently received a great deal of attention. Prior work has applied debloating techniques to a wide spectrum of software applications, ranging from low-level platforms such as kernels and containers (Abubakar et al., 2021; Ghavamnia et al., 2020a) to higher level binaries (Qian et al., 2020; Ghavamnia et al., 2020b; Snyder et al., 2017; Redini et al., 2019; Heo et al., 2018; Quach et al., 2018; Mishra and Polychronakis, 2020; Mishra and Polychronakis, 2018; Mishra and Polychronakis, 2021; Koo et al., 2019) and web applications (Azad et al., 2019; Koishybayev and Kapravelos, 2020; Bulekov et al., 2021).

Static and dynamic analysis are commonly used to identify unnecessary parts of applications that are candidates for debloating. My work also overlaps with techniques that are not geared towards debloating but that statically extract the call graph of web applications, typically to identify vulnerabilities.

Debloating web applications. Prior work focused on debloating different parts of web apps. SQLBlock protects legacy web apps against SQL injection attacks by only allowing a limited set of SQL APIs in each function of a web application (Jahanshahi et al., 2020). Orthogonally, Saphire protects web applications by limiting the list of system calls available to each PHP script extracted by static analysis (Bulekov et al., 2021). Mininode focuses on third-party dependencies in Node.js applications and their code bloat (Koishybayev and Kapravelos, 2020).

Both Saphire and Mininode rely on static analysis to resolve dynamic language constructs such as dynamic file inclusions and class instantiations. Specifically for Mininode, an incomplete static call graph generation due to dynamic imports translates to failure to analyze 12% of packages in their dataset. While this is an acceptable failure ratio for a large scale analysis, it quickly becomes a challenge for application-wide, static debloating techniques.

Less is More (Azad et al., 2019) demonstrates that debloating web apps can lead to the removal of high-severity vulnerabilities and the reduction of up to 60% of their source code. The authors synthetically generate a set of baseline usage profiles for their target applications and dynamically record the files and lines covered while running their tests.

Debloating browsers and other platforms. Hoe et al. explored the idea of reinforcement learning for source code removal in software debloating (Heo et al., 2018). Abubakar et al. apply the idea of debloating to kernels (Abubakar et al., 2021). Orthogonally, Cofine aims to build restrictive system call policies for container environments (Ghavamnia et al., 2020a). Another line of work focuses on the identification and removal of unreachable code in binaries that can be used in code-reuse attacks (Quach et al., 2018; Redini et al., 2019).

Finally, Koo et al. debloat up to 77% of NGINX, VSFTPD, and OpenSSH by the analyzing specific configurations of each instance of these applications and removing code that is not exercised with each configuration profile (Koo et al., 2019). Minimalist is similar in that I use an abstraction of the outside environment to identify the set of features that will not be used within that abstraction (e.g., configuration files or web server logs). That said, the intricacies of binaries and web applications, as well as the differences in configuration files and application entry points, lead to different sets of significant challenges for each project.

3.2 Vulnerability Detection Systems

The second thrust in improving the security of web applications is to detect the security vulnerabilities and patch them before an attacker can exploit. In this section, we discuss existing systems for detecting security vulnerabilities in PHP web applications.

3.2.1 Deserialization Vulnerability Detection

There are a significant number of studies on the security of PHP web applications. In this section, we review the related literature on detecting security vulnerabilities and defending PHP applications against malicious behavior.

Deserialization in PHP Application: In 2009, Esser introduced an approach to exploit deserialization vulnerabilities in PHP applications (Esser, 2010). In light of new attack scenarios, new research has emerged on detecting deserialization vulnerabilities and detecting such attacks on PHP applications. RIPS (Dahse and Holz, 2014a) performs an intra-procedural data flow analysis to detect injection vulnerabilities, including POI. Dahse (Dahse et al., 2014) also proposed an automatic approach for identifying gadget chains that can be used to exploit POI vulnerabilities. Furthermore, FUGIO (Park et al., 2022) introduced an automatic exploit generation tool to create exploit objects for POI vulnerabilities. In an orthogonal direction, my work is to detect the set of PHP API functions that can deserialize user-input. Furthermore, prior works solely relied on manual analysis to specify the set of sinks for the taint analysis or exploit generation tools. Unlike prior work, Argus performs an automatic hybrid analysis to identify the set of PHP API functions that deserialize user-input. In the evaluation of Argus, I showed how the results can improve prior work in detecting injection vulnerabilities.

Deserialization in Other Platforms: Deserialization vulnerabilities threaten a variety of platforms including Java, PHP, Python, and .NET. The research in this area focuses on detecting such vulnerabilities or defending against deserialization attacks. SerialDetector (Shcherbakov and Balliu, 2021) leverages call-graph analysis to identify new patterns of object injection in .NET libraries. Then it uses the discovered patterns to identify new injection vulnerabilities in .net applications. The key difference between SerialDetector and Minimalist is that we aim to detect functions at the PHP interpreter level that deserialize user-input, whereas SerialDetector finds new object injection patterns at the library level. Tanaka presents patterns for attacking deserialization vulnerabilities in the `Pickle` library in Python that lead to denial of service (DoS) attacks (Tanaka and Saito, 2018). Look-ahead object input stream (LAOIS) is a defense mechanism against Java deserialziation vulnerabilities, allowing the type check of the serialized stream before deserialization. There are multiple implementations of this mechanism in SerialKilller (Carettoni, 2021), Contrat-RO0 (Security, 2021), Apache's Common IO library (Commons, 2021), and Java Serialization Filtering (OpenJDK, 2021). The LAOIS approach shows promising results in defending against insecure deserialization, but it is currently deficient in its current implementations against DoS attacks (Koutroumpouchos et al., 2019).

3.2.2 Detection of Other Vulnerabilities

There are multiple studies on detecting vulnerabilities in PHP applications using static, dynamic, or hybrid analysis. Several approaches rely on taint analysis to track unsanitized data and detect injection vulnerabilities (Backes et al., 2017; Dahse and Holz, 2014a; Dahse and Holz, 2014b; Jovanovic et al., 2006; Son and Shmatikov, 2011; Zheng and Zhang, 2013; Wassermann and Su, 2007). Pixy (Jovanovic et al., 2006) and Dahse et al. (Dahse and Holz, 2014b), track data-flows to detect injection vulnerabilities. RIPS (Dahse and Holz, 2014a) conducts taint analysis by building a control flow graph for PHP files in web applications to find injection vulnerabilities. SaferPHP (Son and Shmatikov, 2011) relies on static taint analysis to identify semantic vulnerabilities, including DoS, infinite loops, and missing authorization checks. Zheng et al. (Zheng and Zhang, 2013) focused on identifying remote code execution vulnerabilities by leveraging SMT solvers and finding a path between user-input and critical PHP functions. Backes (Backes et al., 2017) utilizes the code property graph to identify different types of security vulnerabilities, such as SQL injection (SQLi), code injection, and arbitrary file read/write. Multiple works focus on detecting specific type of vulnerabilities such a detecting SQLi (Wassermann and Su, 2007), DoS (Son and Shmatikov, 2011), and cross-site scripting (Jovanovic et al., 2006). Dynamic analysis and hybrid techniques also play an important role in the detection of vulnerabilities (Son et al., 2013; Saxena et al., 2011; Nguyen-Tuong et al., 2005). While earlier work in this area, require manual analysis toward identifying critical functions as taint sinks, Argus uses a systematic and automatic approach to detect injection-sink. In addition, my contribution relies on the analysis of the PHP interpreter itself rather than the PHP application.

Chapter 4

Sandboxing Interpreted Applications with System Call Allowlists

In this chapter, I describe my first contribution to improving the security of PHP web applications. I elaborate on the design of the system in Section 4.1, and then detail Saphire – my prototype implementation of this approach for PHP web applications, in Section 4.3.

4.1 Overview

My method of protecting interpreted applications involves collecting information about the system-calls invoked through the interpreter API, finding the interpreter API functions (e.g. `fopen` in Fig. 4·1) used by interpreted applications, and combining the results to enforce a tailored system-call allowlist. To explain the process in more detail, I first describe the interpreters and programs to which my approach applies. I then explain why generating meaningful system-call allowlists requires consideration of *both* the interpreter and each interpreted program. Finally, I describe the purpose of each of the three stages and explain how their functionality can be combined to secure programs.

4.1.1 Interpreters

I define *interpreted programs* as programs that require an ancillary application (i.e., an interpreter) to execute on a computer. The interpreter is, generally, an application native to the computer system – i.e., it can be directly executed within an operating system by the hardware. Hence, interpreted applications can be portable across systems for which

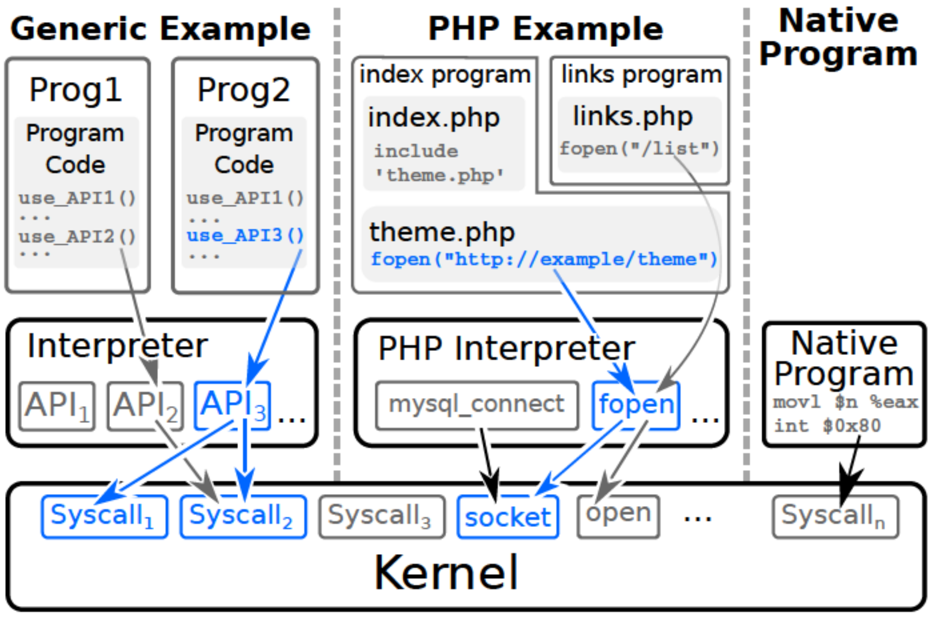

Figure 4·1: Programs rely on system-calls to access resources of the operating system.

compatible interpreters exist. In addition to parsing and executing programs, interpreters expose an API that allows programs to rely on the interpreter for built-in functionality. The API is composed of *functions*, which can be invoked by the interpreted program, each of which can be implemented natively as an interpreter API *handler*. In Figure 4·1, $\texttt{API}_{1,2,3}$, `mysql_connect`, and `fopen` are API functions. The interpreter forms an abstraction layer between the program and the system, with a natively-implemented API bridging the gap. Figure 4·1, shows how interpreted programs access resources guarded by system-calls through the built-in API. Thus, I make a key observation about interpreters and interpreted applications compatible with my approach: *Only the interpreter's code issues system-calls, commonly in response to an interpreted program invoking the API.*[1] Note that some interpreters implement just-in-time (JIT) compilation, translating the interpreted program into native machine code at runtime. For interpreters with JIT-support, such as Java Hotspot

[1]Some interpreters provide the means for programs to execute native code within the interpreter's process(e.g., JNI for Java). Such native additions can be treated as extensions to the interpreter's API. None of the applications in my evaluation rely on this feature, so I do not implement it in my prototype.

and .NET CLR, the translated code still calls into a native API to invoke syscalls, so in the context of Saphire, this optimization is simply an interpreter implementation detail.

4.1.2 An API for all interpreted programs

The functions in the interpreter API must be generic so that they are useful to a wide range of interpreted programs. As a result, the interpreter provides API functions that collectively invoke a diverse set of system-calls. Therefore, I cannot create a meaningful system-call filter by simply enumerating the system-calls invoked anywhere in the interpreter.

Fortunately, individual interpreted programs depend on a small subset of all API functions provided by the interpreter, and in extension only require a small set of system-calls to execute correctly. For example, the generic `Prog1` in Figure 4·1 does not rely on $\texttt{API}_3$ and hence does not need $\texttt{Syscall}_1$.

Thus, during the execution of `Prog1`, it is safe to filter access to $\texttt{Syscall}_1$, even though it occurs within the interpreter binary. To enforce the PoLP, one must analyze the *joint behavior* of the interpreter and the program. Based on these insights, I present a three-stage process for creating tailored system-call filters for interpreted programs.

4.1.3 Securing Interpreted Programs

Stage ① maps the API exposed by the interpreter to a set of system-calls invoked by each API function. In stage ②, the interpreted program is analyzed to identify the APIs it invokes. After composing this information with the map from stage ①, the output of stage ② is the list of system-calls required by the interpreted program (i.e., the system-call filter). In the final stage, the program is executed, and the system-call filter is applied to the interpreter process, protecting the program.

4.1.4 Mapping the interpreter API to syscalls

The goal of stage ① is to identify the system-calls invoked by each API function. As mentioned in Section 4.1.1, interpreters provide programs with access to a generic API, parts of which perform system-calls to expose system-managed resources. Generally, interpreter APIs that depend on system-calls are implemented natively, conforming to the OS-specific system-call interface (see Fig. 4·1).

Both static and dynamic analysis techniques can be used to map API functions to system-calls. For example, APIs can be mapped to system-calls through a static control-flow analysis of the interpreter. The analysis involves labeling the API function handlers as sources, the system-call invocations as sinks, and calculating the reachability between the two sets in the interpreter's call-graph. A dynamic analysis can be used to refine this statically-obtained mapping.

The result of stage ① is a mapping of interpreter API functions to required system-calls. This mapping is generated once, for each version of the interpreter.

4.1.5 Identifying API calls within an interpreted program

In stage ②, I identify the API functions invoked by an interpreted program. Incorporating the mapping from stage ①, this stage determines the system-calls needed by the program. I define a program as the body of interpreted code that can be executed by an interpreter process from an entry-point. For example, in the PHP example in Fig. 4·1, the index program includes the code defined both in `index.php` and in the included `theme.php`. Note that a single script can be included in multiple places, and therefore belong to multiple programs. There are two steps to identify API calls by a program:

1. Identify *all* the code comprising the interpreted program.
2. Analyze the program's code to determine the interpreter API calls it can perform.

Identifying the program's code requires consideration of the interpreted language features that create code-dependencies. In addition to "includes", dependencies can arise from implicit sources, such as customizable auto-loading rules. Once the dependency analysis is complete, I scan the code in the program for API function calls (step 2).

As in stage ①, both static and dynamic techniques can be applied to the program. The result of stage ② is a set of API functions referenced by the interpreted program, that together with the mapping of API calls to system-calls produces the final mapping of programs to system-calls, which is used as an allowlist in the final stage.

4.1.6 Protecting the Program

In stage ③, to protect the program, the interpreter (or program) is modified to load the corresponding allowlist, prior to execution. This dynamic protection can be facilitated by built-in low-overhead support for filtering system-calls, which is present in operating systems such as Linux, FreeBSD, and Windows. The implementation of the protection depends on the execution model of the interpreter. For example, the way protections are applied may differ for programs invoked on the command-line and those executed by a web-server. In Section 4.3, I describe the implementation of system-call filtering for the PHP interpreter on Linux. My filtering mechanism works with both the Apache and nginx webservers, as well as the standalone php-cli interface.

4.2 Threat Model

The threat-model for Saphire assumes that an interpreted application running atop an uncompromised OS contains an ACE vulnerability for which the attacker has an exploit. The goal of this work is to enforce the PoLP on interpreted programs and thereby restrict the capabilities (i.e., the set of available system-calls) the attacker's payload can use. Saphire is designed to restrict an attacker from exploiting an ACE vulnerability in an interpreted program. As such, this work does not focus on attacks that leverage a compromised interpreter to obtain arbitrary-code-execution in a separate service (for example, by triggering a buffer overflow in a database daemon over a network socket). As the evaluation (§4.4) shows, the vast majority of programs comprising real-world web applications can be confined such that existing exploits are mitigated.

4.3 Implementation

I implemented the three steps outlined in the previous section for the PHP language and interpreter in my prototype – Saphire. I choose PHP due to its dominance among web applications, which are major targets of RCE attacks, and because it represents an interpreted language with advanced dynamic features. PHP is dynamically typed, with dynamic binding of function and class names, dynamic name resolution, dynamic symbol inspection, reflection, and dynamic code evaluation support. I explain how Saphire combines static and dynamic analysis techniques in stage ①. I describe the static web application analysis performed in stage ②. Finally, I detail how Saphire uses `seccomp` to sandbox the PHP interpreter on a live web app in stage ③. Figure 5·1 details Saphire's implementation of the three stages introduced in Section 4.1.

4.3.1 Mapping built-in PHP functions to system-calls

PHP refers to API functions as built-in PHP functions. Hence Saphire's stage ① maps built-in PHP functions to system-calls. To this end, Saphire generates an initial mapping, by performing a static call-graph analysis over the PHP interpreter. To refine the statically-collected mapping, I use Linux `ptrace` , which allows Saphire to inspect the system-calls invoked by a running PHP process. Note that `ptrace` is only used, offline, for ① and is not used for any active defense. Moreover, Saphire blocks `ptrace` for all scripts in the web applications I evaluated, by default, as I found no built-in PHP functions that invoke the

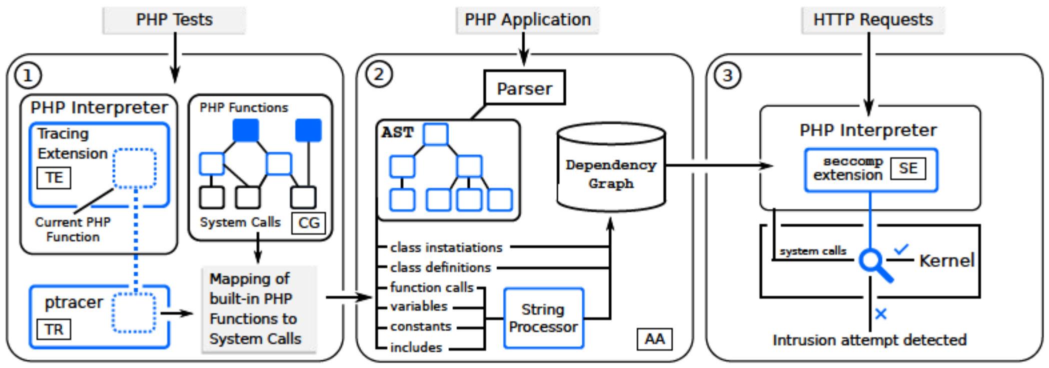

Figure 4·2: Saphire builds a mapping of built-in PHP functions to system-calls, acquires a list of built-in built-in PHP function calls in application scripts, and uses this information to protect the web application using `seccomp` system-call filters.

`ptrace` system-call.

Static analysis over the PHP Interpreter

The PHP 7.1 build I use in the evaluation relies on 55 pre-compiled, dynamic shared libraries. Since PHP generally invokes system-calls through libraries (e.g., libpthread and libc), Saphire builds a static call-graph over the interpreter and all included libraries. Note that the debugging symbols for the interpreter and the 55 libraries are readily available in the Debian repositories. Saphire uses these symbols to facilitate the analysis in stage ① and the production-binary used in stage ③ is stripped. Using the symbols, Saphire builds a call-graph, where each node is a function and each edge is a direct function call. Saphire annotates the function nodes with the system-calls they invoke. To identify the nodes corresponding to the built-in PHP function handlers, I augment `get_defined_functions()` (which lists currently defined functions) to output the address of the handler for each built-in PHP function. Similar techniques are applicable to any interpreter with a symbol table, such as Python (where functions can be enumerated with `dir()` and `global()`). Saphire performs a reachability analysis over the call-graph, where the built-in PHP function handler nodes are the sources, and nodes annotated with system-calls are sinks.

Saphire's rudimentary call-graph analysis is purpose-built to cover libraries and identify system-calls. While this step can be implemented as a static source-code analysis and might yield a more precise call-graph, the analysis needs to operate over dozens of code-bases (for the interpreter, and 55 libraries) using different languages and build-processes.

Refining the mapping through dynamic analysis

The reachability analysis performs an exhaustive search over the code within a PHP process but does not handle indirect calls, which can occur in built-in PHP functions. For example, PHP's `fopen()`, can access remote files over HTTP (see Figure 4·1). Based on the URI, fopen sets a function pointer that specifies whether to use an encrypted HTTPS, or unencrypted connection handler. The static call-graph does not contain edges to either of these functions, which leads to an incomplete mapping of built-in PHP functions to system-calls. Furthermore, some built-in PHP functions execute external programs. `mail()` executes the `sendmail` binary. In order to apply PoLP to the `mail`, the mapping of system-calls should contain the system-calls performed by sendmail. The static analysis over $\boxed{\texttt{CG}}$ does not reason about the system-calls that occur in external processes.

To address these issues, I extend the statically-built profile, by tracing the system-calls performed by the PHP interpreter, while executing its test-suite. The test-suite is packaged with PHP's source. I rely on a PHP extension $\boxed{\texttt{TE}}$, which exposes the name of the currently running built-in PHP function through shared memory. A companion tracer, $\boxed{\texttt{TR}}$ uses Linux's `ptrace` functionality to intercept system-calls. While the interpreter is executing the test-suite, $\boxed{\texttt{TR}}$ intercepts each system-call and examines the current PHP function, exposed by $\boxed{\texttt{TE}}$. This allows Saphire to easily detect whether the currently running built-in PHP function relies on any system-calls missing from the statically-generated mapping. $\boxed{\texttt{TE}}$ also traces system-calls in external programs called by PHP, to account for built-in PHP functions, such as `mail()` which rely on external programs. The test-suite achieves 73.4% line coverage over the PHP interpreter, allowing Saphire to discover additional system-calls used by 137 out of 4,655 built-in PHP functions.

4.3.2 Creating system-call filters for web applications

In Stage ②, Saphire identifies each script's dependencies and determines the built-in PHP functions the interpreter can invoke while running the script. Composing this information with the mapping from ①, Stage ② outputs a set of possible system-calls invoked for each script in the web application.

To achieve this outcome, I built $\boxed{\texttt{AA}}$ to perform a lightweight, flow-insensitive analysis as a limited form of constant folding over strings that compose include statements. $\boxed{\texttt{AA}}$ iterates over all of the PHP files in web applications. I use php-parser (Slizov, 2019) to parse each PHP script into its abstract syntax tree (AST). $\boxed{\texttt{AA}}$ scans the AST for function or method calls to identify possible built-in PHP function calls. If a function call's name matches a built-in PHP function, $\boxed{\texttt{AA}}$ infers that the script contains a call to the built-in. In the case of method calls, $\boxed{\texttt{AA}}$ looks for all assignments of the object within the current scope to identify the class type and checks whether the type and method combination corresponds to a built-in PHP function. To infer script-dependencies $\boxed{\texttt{AA}}$ identifies AST nodes representing: (1) constant definitions, which frequently occur within include paths (2) class definitions/instantiations, which are essential for creating edges for auto-loaded classes, and (3) includes via the `include/require` operations. $\boxed{\texttt{AA}}$ also identifies strings in all variable assignments, as these variables are often referenced in include statements. For each include, $\boxed{\texttt{AA}}$ assembles an internal representation for each of these nodes, optimized for static and string content.

String representation

PHP strings can be composed of literals, and references to constants, variables, and function return values. When $\boxed{\texttt{AA}}$ locates a node representing such a component, it notes its location. Once $\boxed{\texttt{AA}}$ finds all nodes that compose strings, it iterates over the includes in a script.

Saphire handles arguments to an includes, differently, depending on the node's type:

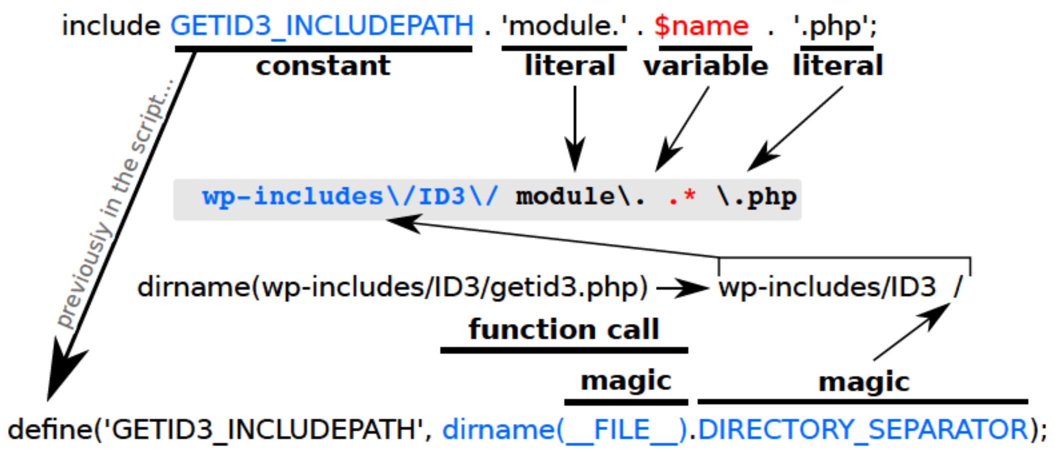
include GETID3_INCLUDEPATH . 'module.' . $name . '.php';
constant
literal
variable
literal
wp-includes\/ID3\/ module\. .* \.php
previously in the script...
dirname(wp-includes/ID3/getid3.php)
wp-includes/ID3 /
function call
magic
magic
define('GETID3_INCLUDEPATH', dirname(__FILE__).DIRECTORY_SEPARATOR);

Figure 4·3: Saphire inspects a WordPress include. The include references a constant defined in the script. Saphire reasons about the `dirname()` built-in API function, so it resolves the value of the constant. The variable `$name` is an argument to the function where the include occurs – Saphire cannot the possible contents, so it translates it to regex as a wildcard `.*`

Literals: For literal strings, nothing needs to be done.

Constant Reference: Since Saphire keeps a record of all constants in the web application, it replaces the reference with the nodes the constant was defined with, and recurses over them.

Magic Constants: The interpreter automatically defines special constants, such as `__DIR__` and `__FILE__`, which describe the location of the current script. $\boxed{\texttt{AA}}$ derives the script locations and filenames from the file-system hierarchy and translates them to literal values.

Variables: If the node is a reference to a variable, $\boxed{\texttt{AA}}$ checks whether the variable is defined in the current scope. As with constants, $\boxed{\texttt{AA}}$ replaces the reference with the nodes used in the assignment. If the variable was assigned multiple times, $\boxed{\texttt{AA}}$ explores each possibility.

Function Calls: If the function is a known common API, such as `dirname`, `realpath`, or `strtoupper`, $\boxed{\texttt{AA}}$ reproduces the functionality over the argument. Otherwise, $\boxed{\texttt{AA}}$ marks the result as unknown.

For each include, $\boxed{\texttt{AA}}$ applies this procedure, recursively, until the include is composed of only literals and unknowns, and the PHP string concatenation operators (`.` and `.=`). Then $\boxed{\texttt{AA}}$ translates the sequence of nodes into a regular expression, substituting the unknowns with regex wildcards (`.*`). Figure 4·3 demonstrates how $\boxed{\texttt{AA}}$ handles includes built with multiple components. If the include refers to variables with multiple assignments, $\boxed{\texttt{AA}}$ joins the regular expressions for each possibility with the "|" operator. $\boxed{\texttt{AA}}$ handles relative path elements, such as `../`, by removing the preceding portions of the expression. If the immediately preceding expression is dynamic (i.e., `.*`), $\boxed{\texttt{AA}}$ replaces all content before the relative path element with a wildcard. Once each include is represented as a regular expression, $\boxed{\texttt{AA}}$ resolves includes by evaluating the regular expression against the paths of the PHP scripts in the web application. For each match, $\boxed{\texttt{AA}}$ stores an edge in a dependency graph, where the nodes are PHP scripts.

Saphire handles auto-loaded classes in scripts by checking if a class with a matching name is declared in the resolved set of dependencies. If not, Saphire searches for matching class declarations in the rest of the web application and creates dependency edges to the corresponding scripts.

Unresolved Includes

In practice, $\boxed{\texttt{AA}}$ statically resolves 74% of includes to a single file. Additional includes can be "fuzzy-resolved" – i.e., resolved to a subset of the files in the web application, such as all files in a subdirectory. Some include statements do not contain any information amenable to static analysis. In these cases, Saphire cannot determine a subset of PHP scripts that can satisfy an include statement. To address this, Saphire provides an option (Conservative Includes, or `CI`) to resolve such includes to all scripts in the application. Enabling `CI` decreases the probability of false-positives due to missing edges in the dependency graph, but increases the number of allowlisted system-calls in scripts containing unresolved includes.

I examine the effects of this option on false-positives and false-negatives in Section 4.4.

Building system-call profiles for Scripts

After identifying the built-in PHP function calls in the script files and building the dependency graph, $\boxed{\texttt{AA}}$ calculates the transitive closure of dependencies for each script, to obtain the list of built-in PHP functions called by the script or any of its dependencies. $\boxed{\texttt{AA}}$ builds the system-call profiles by replacing each of the built-in PHP functions in the list with the set of corresponding system-calls obtained in Stage ①. The output of $\boxed{\texttt{AA}}$ and Stage ② is a system-call profile (i.e., an allowlist) for each script, representing the system-calls for the built-in PHP functions used within the script and all its dependencies. $\boxed{\texttt{AA}}$ marks each script path with its profile. The paths are relative to the root of the web application, so the output of ② is independent of the server and location of the application on the filesystem.

4.3.3 Sandboxing the Interpreter and web application

The goal of stage ③(Sec. 4.1.6) is to sandbox an interpreted program when it executes. My implementation, Saphire, applies the allowlists from Stage ② to a live web app using `seccomp`. Specifically, Saphire deprivileges the PHP interpreter process, before it executes a web application scripts. Internally, Saphire relies on a PHP extension (labeled $\boxed{\texttt{SE}}$ in Figure 5·1) that invokes Linux' `seccomp` facility.

To use `seccomp`, a process provides the kernel with a filter to enforce over process's future system-calls. Upon startup, the PHP interpreter loads the $\boxed{\texttt{SE}}$ extension into the process. $\boxed{\texttt{SE}}$ determines which script the interpreter is about to execute, and provides the kernel with a system-call allowlist – a set of allowed system-calls. After providing the kernel with the filter, $\boxed{\texttt{SE}}$'s task is complete, since the kernel is responsible for enforcing the `seccomp` allowlist.

$\boxed{\texttt{SE}}$ is activated twice during the lifetime of the interpreter. When the PHP process is starting, it loads the $\boxed{\texttt{SE}}$ extension. $\boxed{\texttt{SE}}$ uses this opportunity to load the system-call allowlists from the disk into memory. Once the interpreter receives a request, it hands control to $\boxed{\texttt{SE}}$. $\boxed{\texttt{SE}}$ loads the allowlist for the requested script from memory, and provides it to the kernel, as a filter program. Internally, $\boxed{\texttt{SE}}$ uses `libseccomp`'s bindings to convert a set of system-calls into an allowlist (Moore, 2018). PHP usually accepts web requests from a separate program - the web-server. Web-servers such as `nginx` and `Apache` implement advanced features such as reverse proxying, static resource caching, and load-balancing. When a web-server receives a request that must be handled dynamically, it communicates with a PHP interpreter using an API, such as FastCGI. With a common `nginx` web-server using FastCGI to invoke PHP, the extension and allowlist are only loaded once, by a master process that forks workers to process requests. In the evaluation, I installed Saphire's $\boxed{\texttt{SE}}$ plugin for a PHP interpreter accessible behind both major web-servers on Linux: nginx and Apache. I also deployed the same plugin for PHP's cli API, which allows executing PHP scripts from the command line (similar to Python or Perl). I did not evaluate this configuration, as the vast majority of PHP apps (and exploit targets) are web applications.

If a PHP script does not trigger `seccomp` violations, the interpreter process terminates once script execution concludes. Usually, the process cannot be reused to process other scripts, since different scripts have different system-call privileges, and `seccomp` does not allow a process to replace its system-call filters. This is a problem for interpreters that handle many short requests, since APIs such as `php-fpm` reuse the interpreter for many requests. There are two options to deal with this: (1) Configure the PHP API to only use a PHP interpreter process for a single request. While functional, this approach results in request latency, when the server is under high load. (2) Allow PHP workers to handle many requests, but ensure that each worker only handles requests for the same script. The worker loads a `seccomp` profile for the first request it receives, and this allowlist applies to all subsequent requests to the same script. For an application with many scripts, such as a CMS, dedicated workers handle scripts in high demand, and general workers handle the uncommon requests (restarting after each one). I evaluate both of these options in Section 4.4.4. This issue is specific to interpreters that handle many short-lived requests, such as PHP. For longer lived executions, the one-time overhead of applying the system-call profile is negligible, but if reusing the interpreter is beneficial, routing requests to minimize Saphire overhead is a generic and effective (see Sec. 4.4.4) solution.

4.4 Evaluation

I evaluate Saphire's ability to mitigate remote code execution attacks on a set of popular PHP web applications and plugins. Additionally, I assess Saphire's stages, individually. Specifically, I examine the capabilities of Saphire's include-resolution, the reduction of system-call privileges due to the analysis in stage ②, and the performance of stage ③. My experiments provide answers to three research questions:

RQ1 How precise is Saphire's dependency resolution (§4.4.2)?

RQ2 For each PHP script in a web application, what is the reduction in privilege/available system-calls with Saphire. How does the setting for `CI` affect the reduction (§4.4.3)?

RQ3 Does the retrofitted PoLP protect from known exploits without causing false positives? How does the setting for `CI` impact the accuracy of the system (§4.4.4)?

4.4.1 Evaluation Dataset

I evaluate Saphire on six of the most popular PHP web applications. The set includes the four most popular open-source content management systems: Wordpress, Joomla, Drupal, and Magento. According to W3Techs, these systems comprise 70.5% of the market share among CMS systems, and 38.4% of the market for all websites (Q-Success, 2023a). Additionally, I include one of the most popular administration tools: phpMyAdmin (PCWorld Staff, 2011), and Moodle, a popular course-management system.

In practice, administrators customize CMS deployments by installing plugins. To reflect this, I install nine vulnerable WordPress plugins: NMedia contact form, Wysija newsletter, Foxy Press, Photo Gallery, WP-Property, Reflex Gallery, Slideshow Gallery, WP Symposium, and WPtouch. For this dataset, I selected plugins and web application versions with the most recently published RCE vulnerabilities and readily available proof-of-concept exploits. Additionally, to evaluate false-positives for plugins, I installed 9 of the most popular freely available plugins. In total, the evaluation was conducted over 12 vulnerable versions of web applications, 9 vulnerable and 9 popular freely-available plugins.

Application	Includes						Classes		
	Total	Literal	Dynamic	Resolved	Fuzzy-resolved	Unresolved	Total	Resolved	Unresolved
Drupal 7.0	263	9	254	175	57	31 (11.7%)	40	30	10 (25%)
Drupal 7.5	265	9	256	174	60	31 (11.6%)	48	39	9 (18.75%)
Drupal 7.26	214	1	213	171	42	1 (0.5%)	44	34	10 (22%)
Drupal 7.57	217	1	216	172	43	2 (0.9%)	24	28	6 (25%)
Drupal 7.58	218	1	217	173	43	2 (0.9%)	35	29	6 (17.1%)
Joomla 2.5.25	348	2	346	179	149	20 (5.7%)	252	229	23 (9.1%)
Joomla 3.7	265	5	260	102	152	11 (4.2%)	481	441	40 (8.3%)
Magento	1,190	271	918	971	175	42 (3.5 %)	3,339	3,120	219 (6.6%)
Moodle	7,548	877	6,671	5,605	1,876	67 (0.9%)	2,241	2,149	92 (4.1%)
phpMyAdmin 3.3.10	753	677	76	704	33	16 (2.1%)	49	48	1 (2.0%)
phpMyAdmin 4.8.1	292	222	70	254	32	6 (2.1%)	438	402	36 (8.2%)
WordPress	1,892	517	1,375	1,747	109	36 (1.90%)	215	193	22 (10.2%)

Table 4.1: We break-down the static and dynamic includes for each web app and the number of include Saphire resolves precisely, and approximately. We also present similar data for class references.

4.4.2 Dependency Resolution (*RQ1*)

In stage ②, Saphire scans a PHP web application to determine the built-in PHP functions that might be invoked within each script. The accuracy of the system-call profile depends on the results of this stage. One of the main challenges for Saphire is resolving the dependencies between scripts. To address this challenge, Saphire uses include and class resolution to discover the dependencies.

Table 4.1 presents the include resolution statistics for the web applications in the dataset, collected after Saphire's static analysis. The *literal* column shows the number of include statements with a string literal argument. The *dynamic* column shows the number of include statements with arguments that are *not* string literals. The *resolved, fuzzy-resolved, and unresolved* provide a breakdown of how the static analysis in ② resolved these includes. Namely, the *resolved* column contains the number of includes resolved to a single script within the web application. The *fuzzy-resolved* column specifies the number of includes that are resolved to a subset of all web application scripts. That is, the regular expression generated by $\boxed{\texttt{SE}}$ matched to multiple scripts. On average Saphire resolves 74% includes and fuzzy-resolves 22%.

In the same table, I show Saphire's class resolution statistics. On average, 85% of classes are resolved. Unresolved classes can occur when web applications define classes dynamically. For example, for historical reasons, Joomla dynamically creates an alias for each defined class by prefixing the class name with "J" (e.g., the original `Http` class will trigger the creation of `JHttp` as an alias)[2]. As the Joomla code-base uses both notations interchangeably, Saphire detects dependencies only if the original notation is used and does not detect dependencies if classes are referred to via their aliases. While this behavior could be easily emulated in the analysis by duplicating the alias-generating logic in Saphire, I chose to elide any program-dependent modifications to the system. If the application relies on an autoloader, the effect of unresolved classes is the possibility of false-positives due to missing edges in the dependency graph. As we will see, the only false positives Saphire encountered during the evaluation were caused by the Joomla idiosyncrasies described above.

4.4.3 System-Call Profile Size (*RQ2*)

The security benefits provided by Saphire hinge on its ability to restrict access to system-calls – specifically those that are likely to be used by attackers. In this section, I examine the reduction in attack-surface in terms of the number of system-calls in the allowlists. For

[2]This is implemented in Joomla's class auto-loader. If the script instantiates a class with the name `JHttp` but the auto-loader cannot find it, the loader trims the "J" prefix and looks for a class with the name `Http` instead.

qualitative measure, Section 4.4.4 further examines the dangerous system-calls (as defined by Bernaschi et al. (Bernaschi et al., 2000)) that exploits can still use. I collected the data presented here by running stages ① and ② on a system running Linux Kernel 4.17 which provides 333 system-calls. Figure 4·4 shows the number of system-calls allowed for each script in WordPress 4.6, phpMyAdmin 4.8.1, Joomla 3.7, and Drupal 7.58. The colored regions represent profile sizes with the CI option *enabled*. The bottom-most region, *Available Dangerous*, represents the dangerous system-calls available to each script. The *Available* region represents additional system-calls available to each script that are not considered dangerous. Hence, the allowlist for a given file consists of the system-calls contained in these two regions. The upper two regions represent blocklisted system-calls and dangerous system-calls, respectively. The black line represents the system-call profile sizes when the CI option is *disabled* (no dependency edges for unresolved includes).

As the graphs illustrate, Saphire generates system-call allowlists that significantly reduce the attack surface. The overall reduction of the attack surface in the number of system-calls is 80.5% on average, with the most permissive profile (i.e., the left-most script in Joomla) still removing 72% of system-calls from the allowlist. More important than the bare number of system-calls, Saphire reduces the number of available *dangerous* system calls by 80% on average.

I note that "shelves" of system-calls occur in most of the graphs, indicating that many files require the same number of system-calls. This phenomenon is due to the fact that sets of scripts share the same dependencies. For example, Saphire finds that WordPress' `wp-includes/option.php` is included in 383 (28% of scripts). This leads to many files sharing similar system-call profiles.

When `CI` is enabled, scripts with unresolved includes include all other scripts in the web application. This results in "shelves" at the maximum profile-size, indicating that the scripts can invoke any system-call used in the entire web application. This reduces the possibility of false-positives due to missing dependency edges but increases the potential attack surface.

4.4.4 Defense Capabilities (*RQ3*)

I evaluate Saphire's protection against 21 remote code execution exploits. To this end, I created 12 Docker containers running vulnerable versions of the web applications in the evaluation dataset. As mentioned in Section 4.4.1, the WordPress installation contains 9 vulnerable plugins, for a total of 11 WordPress vulnerabilities. I attack the web applications

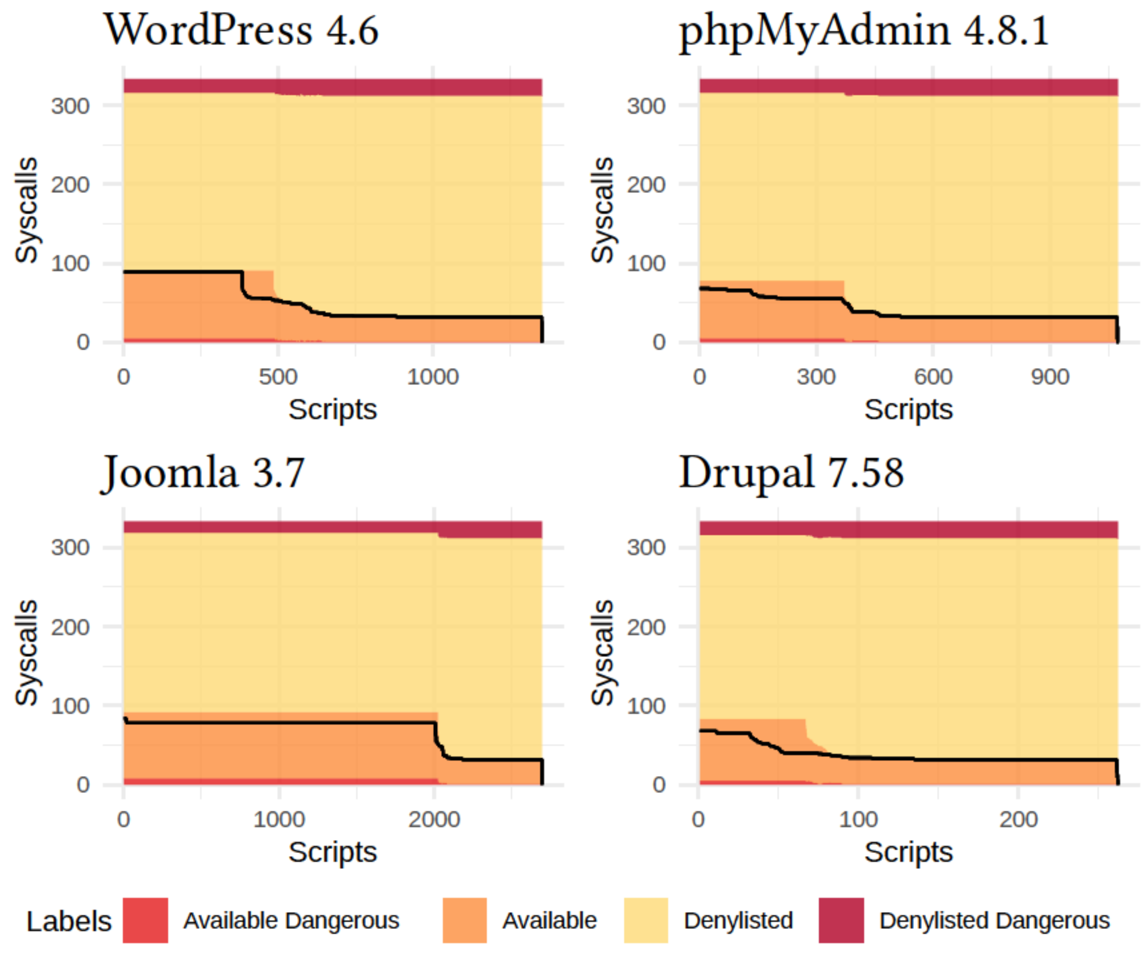

Figure 4·4: The sizes of the system-call allowlists for web apps in our test-set. The shaded areas represent allowlist sizes, when `CI` is *enabled* The black line shows the allowlist size when `CI` is *disabled*

		Exploits Blocked		False Positives		Benign Traces	Dangerous System Calls Available to Exploits
Application	Vulnerability	`CI` off	`CI` on	`CI` off	`CI` on	Line Coverage	
Drupal 7.0	CVE-2014-3704	y	y	0	0	†39.22%	openat, unlink
Drupal 7.5	drupal_restws_exec	y	y	0	0	†28.50%	chmod, openat, rename, symlink, unlink
Drupal 7.26	CVE-2014-3453	y	y	0	0	†37.12%	chmod, openat, rename, symlink, unlink
Drupal 7.57	CVE-2018-7600	y	y	0	0	†42.51%	chmod, openat, rename, symlink, unlink
Drupal 7.58	CVE-2018-7602	y	y	0	0	†43.63%	openat, unlink
Joomla 2.5.25	CVE-2014-7228	y	y	2	0	12.67%	chmod, openat, rename, unlink
Joomla 3.7	CVE-2017-8917	y	y	1	0	†27.95%	chmod, openat, rename, unlink
Magento 2.0.5	CVE-2016-4010	y	y	0	0	†40.61%	
Moodle 3.4	CVE-2013-3630	y	y	0	0	†28.00%	chmod
phpMyAdmin 3.3.10	CVE-2011-4107	y	y	0	0	13.72%	chmod, openat, rename, symlink, unlink
phpMyAdmin 4.8.1	CVE-2018-12613	y	y	0	0	†49.28%	chmod, openat, rename, unlink
Wordpress 4.6	11 Vulnerabilities	y	y	0	0	$^{\dagger *}$36.18%	

$*$ **Wordpress & Plugins Vulnerabilities** The WordPress vulnerabilities: WPVDB-7896, WPVDB-6680, WPVDB-6231, CVE-2014-9312, WPVDB-6225, CVE-2015-4133, CVE-2014-5460, CVE-2016-10033, WPVDB-7716, WPVDB-7118 wp_admin_shell_upload. Coverage including the vulnerable plugins is 17.88%. See Sec. 4.4.4 for evaluation over popular plugins.
Complete list of dangerous system-calls: *chmod, fchmod chown, fchown, lchown, execve, mount, rename, open(at), link, symlink, unlink, setuid, setresuid, setfsuid, setreuid, setgroups, setgid, setfsgid, setresgid, setregid, create_module*

Table 4.2: Exploits blocked for each configuration of Saphire. Coverage annotated with † was collected with the aid of unit-tests available for the web app.

using exploits from the Metasploit Framework (Metasploit, 2019) and consider an attack successful if it exposes a shell to the attacker via the network. Of course, I first verified that all exploits work against unprotected versions of the web applications and plugins. In Table 4.2, I present the results of the experiments. Specifically, I evaluate the defense capabilities of Saphire when `CI` is enabled and disabled.

Is Saphire too restrictive?

To properly apply the PoLP, Saphire should not prevent the normal operation of the web applications. A false-positive for Saphire is a system-call blocked during benign execution of application code. Saphire *does not* rely on any benign web application traces to build the allowlists, but I exercise the web applications to evaluate how prone Saphire is to false positives through an ensemble of three complementary techniques:

- I replay browsing traces collected while one of the authors explored the web-app as a user and administrator. The traces exercise functionality available to privileged and unprivileged users.

- I crawl the application with a web crawler included in the Burp Suite. The crawler is authenticated and has access to privileged web application functionality.
- When available, I execute test-suites packaged with the web-apps. The test-suites are collections of PHP scripts that exercise portions of the web application code.

I measure the combined line coverage of the three methods using the XDebug PHP debugger (Rethans, 2018) and present the coverage as a percentage of total lines of PHP code (as determined by sloccount (Wheeler, 2004)) in Table 4.2. The measurement accounts for possible coverage overlap between the three techniques and registers each covered line only once. However, sloccount greedily counts source lines that are not considered executable and are consequently not tracked by XDebug. Hence, the average coverage of 33% is a strict lower bound of the true coverage that the mechanisms achieve. I selected Less is More (LIM) (Azad et al., 2019) as, to the best of our knowledge, it is the most recent work to collect comprehensive coverage data over PHP web applications. Unlike Saphire, LIM collects coverage during an exploration stage to prune unused functions, thus cutting back on a web application's attack-surface. LIM presents coverage as the percentage of lines in functions with any lines covered during the exploration stage. I.e., for any partially covered functions, no lines are pruned. To mirror this technique, I calculated the percentage of lines contained in functions with *any* lines executed. According to this metric, Saphire covers 64% of WordPress, while LIM covers 57%. Additionally, I obtained the Selenium traces collected by LIM and executed them on my instance of WordPress. These Selenium traces increased the coverage by 1% without raising any false-positives. In summary, I found that the coverage is in-line with state-of-the-art PHP research. Moreover, the difficulty in obtaining high dynamic coverage for web applications highlights the utility of using a static analysis for Saphire's implementation of ②, which can generate profiles even for uncovered code.

The center column-group of Table 4.2 shows the number of false positives for different settings of `CI`. With `CI` enabled, Saphire did not raise any false positives during the evaluation, as it conservatively assumes that an unresolved include can refer to any script within the web application. While this conservative setting reduces false positives, it results in slightly larger allowlists (see the bottom two regions of the Figure 4·4 plots). When `CI` is disabled, system-call profile size is decreased (i.e., system-calls under the black line in Figure 4·4), but false positives do occur. Specifically, I encounter three false positives, all within Joomla versions 2.5.25 and 3.7. The reason for all three false positives is the automatic generation of aliases, as explained in §4.4.2. Concretely, `administrator/index.php` instantiates `JHttp`, which in turn relies on the built-in PHP function `curl_exec`. Although, Saphire's stage ① correctly determines that `curl_exec` requires the `getpeername` and `setsockopt` system-calls, stage ② misses the dependency introduced by instantiating the `Http` Joomla-class via its alias `JHttp`. The simple, yet Joomla-specific, modification to Saphire described above would remove these false positives. Saphire handles calls to APIs that depend on external binaries. For example, Drupal relies on the `mail()` API during user registration. Since Stage ① tracks the system-calls that the external `sendmail` binary performs, I observed no false positives from such functionality. Finally, though I did not have access to any popular PHP sites, I installed Saphire on a public web-server running WordPress. In total, the web-server received 13,261 HTTP requests. Though many of these requests originated from benign crawlers, some appeared to search for unsecured API endpoints, such as WordPress' `xmlrpc.php`. None of these requests triggered Saphire alerts. I performed a manual inspection of the web-server's filesystem to confirm that it had not been compromised.

Payload Constraints

Table 4.2 also presents the effect of Saphire's script de-privileging on the attackers using web application exploits to execute the Metasploit payload in the "Exploits Blocked" columns. Of course, adversaries are not limited to Metasploit payloads and can craft exploits that do not extend past the exploited script's system-call privileges.

To assess the impact of such attacks, I enumerate which *dangerous* system calls are present in the allowlist for scripts that contain RCE vulnerabilities with `CI` enabled. I consider a system-call dangerous if it is listed as a "Threat level 1 system call" in (Bernaschi et al., 2000). All remaining dangerous system calls for the corresponding vulnerable files are shown in the last column of Table 4.2. While attackers are free to modify the exploits, they can only use the dangerous system calls listed in the table. Notably, none of the payloads can spawn new processes outside the interpreter (no `execve`). `CVE-2016-10033` is specific to Apache. Though I primarily test against nginx, I configured an Apache server with Saphire protections. The steps for configuring Saphire for nginx and Apache were virtually identical, aside from differences in the config-file syntaxes. `CVE-2016-10033`, leverages parameter injection to the external sendmail executable through PHP's mail() function. Since Saphire collected an accurate profile for the mail() and the sendmail child processes, I defend against this CVE, without raising false positives on the same page. The remaining dangerous system-calls potentially allow the attacker to tamper with website-content, but are insufficient to achieve arbitrary code-execution. Saphire protects against RCE launched via file upload vulnerabilities by default. If an attacker exploits a file-upload vulnerability, the uploaded script will have an empty system-call allowlist, as the script was not present during the stage ② analysis. Thus, the uploaded script cannot make any system-call and cannot meaningfully contribute to the attacker's goals.

Additionally, I test Saphire against a set of 40 real payloads. Though there are few php-based payload datasets readily-available, I ran the payloads in one such dataset (Geniar,

		False Positives		
Plugin Name	Web App	`CI` off	`CI` on	Coverage
ContactForm	Wordpress	0	0	39.14%
Yoast	Wordpress	0	0	†28.20%
Akismet	Wordpress	0	0	32.53%
WooCommerce	Wordpress	0	0	†27.93%
Classic Editor	Wordpress	0	0	40.76%
Akeeba	Joomla	0	0	14.67%
Acymail	Joomla	0	0	15.66%
Ctools	Drupal	0	0	†27.60%
Views	Drupal	0	0	†43.69%

Table 4.3: False positive test for popular web app plugins. Coverage marked with † was gathered with the aid of available unit-tests.

2014) and found that every payload relies on system-calls missing from the profiles for vulnerable scripts listed in Table 4.2.

Analysis of Non-vulnerable Plugins

Most web applications in my evaluation dataset feature powerful plugin architectures. As such, I assess whether Saphire triggers false-positives in the plugins that leverage this infrastructure. Using the above ensemble of three methods to determine coverage, I exercise the popular plugins to assess Saphire for false positives (see Table 4.3). On average, I achieved 31.76% line coverage, which is in line with existing work focusing on web applications (Azad et al., 2019), though the statistics also cover plugins. Some plugins guard premium features behind paywalls. Since I did not pay for the plugins, these features contributed unreachable code, lowering the coverage I could achieve.

Runtime overhead

Response time is a critical metric for web-server workloads. I note that Saphire's analysis stages ① and ② are performed offline. Stage ③ uses a PHP extension to sandbox a web-

	Concurrency	Wordpress		Trivial Script		
		Default	Protected	Default	Protected	Optimized
nginx	1	328.252	328.78 (0.16%)	0.185	1.941	0.188 (1.62%)
	2	353.982	355.776 (0.51%)	0.192	2.316	0.194 (1.04%)
	4	348.242	348.639 (0.11%)	0.264	4.347	0.265 (0.38%)
	8	361.377	363.83 (0.68%)	0.512	8.62	0.516 (0.78%)
	16	416.639	419.342 (0.65%)	0.924	18.61	0.93 (0.65%)
	32	863.932	867.932 (0.46%)	1.71	43.38	1.713 (0.18%)
Apache	1	338.75	337.42	0.201	1.91	0.204 (1.49%)
	2	368.84	370.02	0.209	2.44	0.212 (1.43%)
	4	369.48	369.55	0.236	4.53	0.233 (1.28%)
	8	372.84	372.98	0.559	8.77	0.564 (0.89%)
	16	412.47	414.42	0.954	19.02	0.961 (0.73%)
	32	872.57	877.24	1.77	42.21	1.78 (0.56%)

Table 4.4: Response times for requests to WordPress index.php and a worst-case, trivial script. All response times in milliseconds.

app, by loading a system-call profile for the PHP script, at the beginning of each request. Additionally, Saphire relies on different system-call profiles for each script. Since `seccomp` does not allow Saphire to replace the system-call profile, by default, Saphire configures PHP to restart the process after serving each request. Modern web-servers, such as nginx with `php-fpm`, typically reuse PHP processes to handle multiple requests, and I consider this in the evaluation. I perform two experiments on a system with an 8-core Intel Xeon E5-2620v2 @2.10GHz, 256GiB DDR3, running Linux 4.17 with nginx 1.14, PHP 7.1 with `php-fpm`, and MySQL 5.7.

I measure Saphire's overhead by observing the response time for WordPress' `index.php` by using ApacheBench (Apache Software Foundation, 2018), over 15,000 requests, at multiple levels of request concurrency. I compare the default configuration of `php-fpm` against `php-fpm` configured to use processes for a single request. Table 5.3 presents the performance overhead of using Saphire, which indicates negligible overhead at all levels of concurrency. To further generalize these results, I repeated the experiments for Apache 2.4.

Additionally, I benchmarked a *worst-case* scenario for Saphire, where the interpreter executes a trivial script. The script prints a single line of text prior to exiting. I use ApacheBench to benchmark the trivial script across 50,000 requests under default and protected `php-fpm` configurations. The results are presented in Table 5.3, in the first two columns under the *Trivial Script* heading. I observe that disabling the reuse of PHP workers has a severe impact on performance for the worst-case script since each interpreter process is only active for a short time before it must be restarted.

To avoid the performance penalty due to the `php-fpm` configuration change, Saphire takes advantage of `php-fpm`'s built-in pooling feature and nginx URL-routing capabilities. First, an administrator specifies a set of high-demand PHP pages (this information is easily obtained from server logs). Saphire configures separate `php-fpm` pools for each specified page and creates nginx rules to route requests to the proper pool based on the URI. Saphire also creates a catch-all pool, where processes are not reused, for scripts that are in low-demand. Note that the total number of `php-fpm` processes does not increase, and `php-fpm` automatically assigns and removes workers to each pool based on demand.

This configuration change enables protected `php-fpm` workers to process multiple requests, without having to restart to re-apply the seccomp filter again. I present the benchmarks for this configuration in the last column of Table 5.3. Observe that by configuring nginx to route requests to script-specific pools, I eliminate virtually all overhead.

4.5 Limitations and Discussion

In this section, I discuss the limitations of the Saphire prototype and possible areas for future work.

eval and system: `eval()` evaluates a string as PHP code. Saphire does not consider includes, or calls to built-in PHP functions inside `eval()` arguments. `system()` executes an arbitrary shell command. None of the false-positives I observed resulted from `eval` or `system` calls, and Saphire supports execution of pre-determined external programs such as `sendmail` through the `mail()` API function. In future work, Saphire can be improved, to analyze static content in arguments to `eval` and `system`.

Mimicry: Saphire's goal is to apply the PoLP, as it relates to system-calls, to interpreted applications. This severely restricts the system-calls that the exploit and payload can rely on. In section 4.4.4, I discuss the scarcity of "dangerous" system-calls available to attackers. Even so, Saphire does not explicitly detect ACE attacks, and the attacker can attempt to craft a payload that only invokes allowed system-calls. For example, the attacker might still leverage vulnerabilities to add undesired content to content management systems.

Overwriting scripts: Saphire's system-call profiles are read-only to the PHP interpreter. If an attacker has write access to scripts on an upload path, they can, potentially, overwrite an existing script with a payload. Saphire will limit the uploaded script with the whitelist it built in Stage ②. Therefore, the attacker can overwrite a script with a larger system-call privilege-set. If scripts must be writeable, Saphire can be easily augmented to record a checksum for each PHP script during Stage ② and ensure that the checksum is unchanged, when the script is loaded in Stage ③.

Installing plugins: When a site administrator installs a new plugin into a web application, they run stage ② on the plugin source in a safe directory, and then Saphire merges the plugin's system-call profile into the profile for the rest of the web application. Currently, ② is run manually for new plugins, removing some of the convenience of web applications that support installing plugins directly through the web-interface.

Saphire does not filter system call arguments: Saphire applies the PoLP to PHP scripts, where it considers each system-call type as privilege. This idea can be further extended to consider system-call arguments as privileges. Though system-call arguments can be derived from user-input, at run-time, , unchanging arguments can be determined statically during Saphire's stage ② and filtered in ③. In my evaluation over 21 exploits, Saphire blocked all attacks by simply filtering based-on system-call type and I leave argument-based PoLP to future work.

Line coverage of evaluated web applications: I use an ensemble of human-driven and automatic techniques, as well as unit-testing to test for false-positives. In the evaluation, I achieve an average line coverage of 31.76% which is in line with similar work (Azad et al., 2019; Baek and Bae, 2016; Machiry et al., 2013; Artzi et al., 2010; Doupé et al., 2012; Antunes and Vieira, 2010). Unlike these works, the web applications contain large plugins. Since phpMyAdmin 3.3.10 and Joomla 2.5.25 do not include test-suites, their coverage is significantly lower. Another factor limitting possible coverage is the fact that web applications often rely on small fractions of large frameworks. For example my WordPress installation contains the wp-property plugin, which includes TCPDF (a PDF generator). The wp-property code does not reference TCPDF anywhere, so this idle code (39k lines, or 11% of my WP install) is likely unreachable.

Chapter 5

Mitigating SQL Injection attacks on PHP Web Applications

In this chapter, I describe the second contribution to improve the security of PHP web applications. I elaborate on the design of my system in Section 4.1, and then detail SQLBlock – the prototype implementation of this approach for PHP web applications in Section 4.3.

5.1 System Overview

In this section, I explain how SQLBlock records benign SQL queries and limits the access of functions in a web application to the database. Figure 5·1 shows an overview of how SQLBlock defends web applications against SQLi attacks. Specifically, SQLBlock records a profile by observing benign issued queries by a web application. SQLBlock then enforces the profile from inside the database for every query that the web application sends to the database.

In step ①, SQLBlock performs a static analysis over the web application to identify the database procedures that are used across the web application's scripts. This analysis is done once per web application and SQLBlock uses this information during training and enforcement of the profile.

In step ②, SQLBlock is in the training mode and records the benign issued SQL queries by the web application. SQLBlock can use benign browsing traces or the web application's unit tests in its training. SQLBlock creates a mapping between the benign SQL queries that MySQL receives and the functions in the web application that used the database access

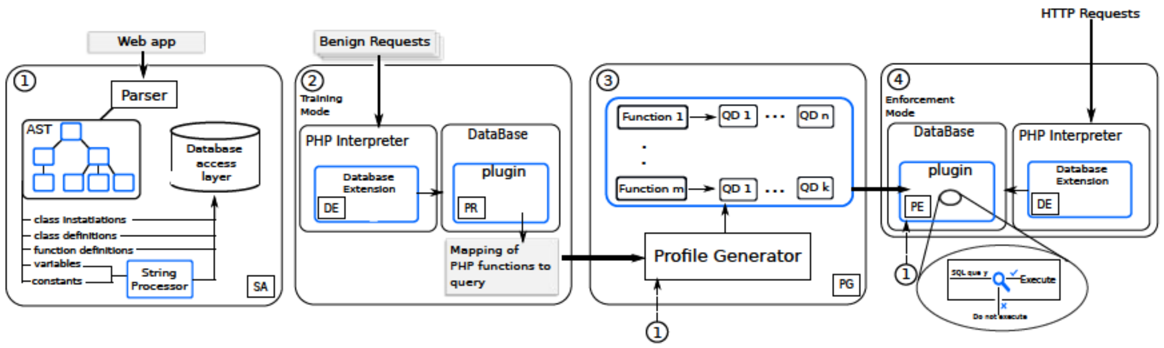

Figure 5·1: SQLBlock extracts the database access layer of the web application, builds a mapping between the function and the SQL queries it issues, creates a profile for each function in the web application and enforces the profile using a MySQL plugin.

layer to issue the query to the database.

In Step ③, SQLBlock leverages the information from the first two steps to assemble a trusted database-access profile. The profile is a set of allowed tables, SQL functions, and type of SQL queries that each function in the web application can issue. At the end of the third step, SQLBlock acquires the necessary information to protect the web application from SQLi attacks.

In step ④, SQLBlock protects the running web application against unauthorized database access by filtering access to the database according to the trusted profile generated in step ③. The modified database extension (e.g., PDO in PHP) appends the execution information (i.e., call-stack) at the end of each SQL query as a comment before sending it to MySQL. Prior to the execution of each SQL query, SQLBlock extracts the appended execution information from the SQL query and identifies the function that communicate with the database using the database access layer. SQLBlock checks the query against the profile that corresponds to the function that issued the query. Finally, if the SQL query matches the profile, MySQL executes the query and returns the results.

```
$id = $_GET["id"]
function get_public_info{
include dirname(__FILE__)."/db/database.php";
$users = executeQuery("public_info", $id);
...
}
get_public_info();
```

(a) `get_public_info.php`

```
class DatabaseConnectionmysqli
        extends mysqli {
  private $query;
  function __construct(){
      parent::__construct("localhost","admin","admin","mysqldb");
  }
  public function setQuery( $query ){
      $this->query = $query;
      ...
  }
  public function execute(){
      return parent::query($this->query);
  }
  public function multi_execute(){
      $result = parent::multi_query($this->query);
      ...
  }
}
function executeQuery( $tbl, $arg ) {
  $query = "SELECT * FROM ".$tbl." WHERE id > ".$arg;
  $classname = "DatabaseConnection".$this->getDriver();
  return new $classname()->setQuery($query)->multi_execute();
}
```

(b) `/db/database.php`

Listing 5.1: Illustrative PHP code snippets demonstrating dynamic inputs to *new* keyword

5.1.1 Static Analysis of Web applications

The web application database access layer provides a unified interface to interact with different databases. In step ①, SQLBlock identifies the database access layer by statically analyzing the web application. To this end, SQLBlock creates a class dependency graph (CDG). The CDG is a directed graph $CDG = (V, E)$, where the vertices (V) are classes and interfaces in the web application. An edge $e_{1,2} \in E$ is drawn between $v_1 \in V$ and $v_2 \in V$ if v_1 extends class v_2, implements interface v_2 .

After creating the CDG, SQLBlock extracts the list of classes and interfaces in the web application that extend database APIs (e.g., PDO in PHP). To do so, I manually identify database extension classes (e.g., `mysqli` in PHP). Afterwards, SQLBlock iterates over the vertices of the CDG and checks whether a vertex is connected to the database API. If a vertex is connected to the database API, SQLBlock adds it to the database access layer. SQLBlock also adds classes to the database access layer if their methods initialize an instance of database API in PHP (e.g., `mysqli_init`). At the end of this iteration, SQLBlock possess a list of all classes and interfaces in the web application that extends the database API.

Besides the object-oriented design of database APIs in web applications, operations on databases (e.g., `SELECT` operation) also have procedures (Drupal, 2016). Database procedures handle creation of objects from database API and setting correct parameters for modules in the web application. A database procedure returns an object from a sub-type of a database API. SQLBlock analyzes the body of the functions and procedures in the web application for the returned objects. If the returned object is from a sub-type of a database API in the web application, then SQLBlock considers it as a database procedure. At the end of this step, SQLBlock extracts information regarding the database API as well as database procedures. This step is necessary for SQLBlock to find the function that used the database access layer for communicating with the database during training and enforcing of the profile.

Listing 5.1(b) shows a snippet of PHP code from a class that extends the database API `mysqli`. There is also a database procedure called the `executeQuery` in Listing 5.1(b) that return an object from `DatabaseConnectionmysqli` that is a subclass of `mysqli`. Listing 5.1(b) shows another code snippet which implements a function called `get_public_info`, which uses `executeQuery` to retrieve data from the database. In such a case, SQLBlock identifies `DatabaseConnectionmysqli` as a subclass of `mysqli` and `executeQuery` as a database procedure.

5.1.2 Collecting Database Access Information

In step ②, I train SQLBlock using benign traces or unit tests to learn benign SQL queries. Step ② consists of two components that work together to create a mapping between the received SQL query in MySQL and the function that composed the SQL query. The first component, appends the execution information at the end of each SQL query before sending it to the database. The execution information includes the call-stack in the web application that led to sending a SQL query to the database using database extensions (e.g., `PDO` or `mysqli` in PHP).

The second part, a MySQL plugin, intercepts the execution of the incoming SQL queries to MySQL. When MySQL receives a SQL query through benign traces or unit tests, SQLBlock records the SQL query that MySQL receives including the execution information appended to the SQL query. Since SQLBlock has access to the parse tree of the SQL query, SQLBlock traverses the parse tree and records information regarding the type of nodes in the parse tree of the SQL query. SQLBlock also logs the list of tables that the SQL query accesses, as well as the type of operation (e.g, `SELECT` operation) in the SQL query.

5.1.3 Creating the Profile

SQLBlock in step ③, leverages the access logs collected from benign SQL queries in step ② and generates a profile that defines the access to the database for each function in the web application that interacted with the database. Particularly, the profile contains a set of query descriptors for each function in the web application. A query descriptor comprises four components. Each component specifies a different aspect of the database access, that we explain below.

- ***Operation:*** denotes the type of operation in the SQL query. The operation can be `SELECT, INSERT, UPDATE, DELETE`, etc. (Corporation, 2019). The profile records the type of operation in each SQL query. Enforcing the operation type removes the possibility of a SQLi attack performing a different operation. For instance, when the profile only specifies a `SELECT` operation, the SQL query cannot perform an `INSERT` SQL query.

- ***Table:*** determines the tables that the SQL query can operate on. Restricting the tables used in a SQL query prevents an attacker from executing a SQL query on a different table.

- ***Logical Operator:*** indicates the logical operators (Corporation, 2019) used in the SQL query. Logical operators limit the ability of an attacker to use a tautology attack in a SQL query for extracting data from a table.

- ***SQL function:*** determines the list of functions that the query uses. The component also records the type of arguments that are passed to each function. The list of functions restricts the attacker to use only the functions that are recorded during the training. This limits the attacker's capability to use alternate encoding and stored-procedures attacks against the database.

At the end of step ③, SQLBlock acquired a set of query descriptors for each function in the web application that issued a SQL query based on the training data that was obtained in step ②.

5.1.4 Protecting the Web Application

In the last step, SQLBlock is in enforcement mode and uses the profile created in step ③ to restrict access to the database for each function in the web application. When the database receives a SQL query, SQLBlock extracts information regarding the type of operation, table accesses, and parse tree of the received SQL query. Subsequently, SQLBlock extracts the function that issued the SQL query from the execution information appended to the incoming SQL query. Afterwards, SQLBlock looks up in the profile and retrieves query descriptors associated with the function that composed and issued the SQL query. For each query descriptor associated with function, SQLBlock compares each component of the query descriptor with the obtained information from the received SQL query. First, SQLBlock checks whether the type of operation in the received SQL query and in the query descriptor is the same or not. Second, SQLBlock examines the list of tables in the received SQL query. The list of table in the received SQL query must be a subset of the list of tables in the query descriptor. For the logical operators, SQLBlock checks whether the logical operators in the SQL query that MySQL received is subset of the logical operators in the query descriptor. Finally, SQLBlock inspects the functions used in the received SQL query as well as the type of arguments. The functions and the type of arguments must be in the recorded query descriptor. SQLBlock takes a conservative approach and allows the database to execute the SQL query only if all four components of a query descriptor associated with the function authorize the SQL query.

5.2 Implementation

In this section, I elaborate on the implementation challenges that needed to be addressed build SQLBlock. First, I explain how SQLBlock statically analyzes PHP web applications to identify the database access layer. Afterwards, I describe how SQLBlock uses the MySQL plugin API to record the SQL queries that the database receives. I explain how SQLBlock creates a precise profile for each PHP function based on the SQL queries issued to the database. Finally, I describe SQLBlock's approach for using a MySQL plugin API to restrict database accesses.

5.2.1 Static Analysis of web applications

In step ①, SQLBlock analyzes the web application to determine the database API and database interfaces across the PHP scripts in a web application. SQLBlock performs a flow-insensitive analysis, which focuses on finding database API, interfaces, and procedures.

SQLBlock identifies all PHP files in the web application, using `libmagic`. I use php-parser (Slizov, 2019) to parse each PHP script into an abstract syntax tree (AST). SQLBlock identifies classes, interfaces, and abstract definitions by scanning AST nodes that represent their corresponding definitions. SQLBlock examines interface and class definitions across the PHP web application to reason about the dependencies between classes and interfaces, During analysis, SQLBlock creates a class dependency graph (CDG) and draws an edge between interfaces and classes when: 1) An interface extends another interface. 2) A class implements an interface. 3) A class extends another class.

After creating the CDG, the static analyzer ($\boxed{\texttt{SA}}$) iterates over the nodes of the CDG to identify classes and interfaces that facilitate communication between the PHP web application and the database. To accomplish this, SQLBlock starts with the `PDO` and `mysqli` classes; two of the most popular database extensions in PHP. SQLBlock creates a list of classes and interfaces that share an edge with `PDO` or `mysqli` classes in the

CDG. For example, after creating the CDG for the code in Listing 5.1(b), $\boxed{\texttt{SA}}$ identifies `DatabaseConnectionmysqli` as a subclass of `mysqli`.

$\boxed{\texttt{SA}}$ must identify database procedures as well. $\boxed{\texttt{SA}}$ decides whether a procedure is a database procedure or not by analyzing the type of object it returns. If a procedure returns an object from a subclass of the database API, $\boxed{\texttt{SA}}$ marks that function as a database procedure. For determining the object type that a function returns, $\boxed{\texttt{SA}}$ analyzes the AST node of the return statement. There are two cases that $\boxed{\texttt{SA}}$ is interested to follow:

- **Instantiating an object using the `new` keyword:** If the function is instantiating an object using the `new` keyword in the return statement, $\boxed{\texttt{SA}}$ analyzes the argument that is passed to the `new` keyword. If the argument is the name of a subclass of a database API, $\boxed{\texttt{SA}}$ marks the function as a database procedure. If the argument is a variable, $\boxed{\texttt{SA}}$ performs a lightweight static analysis as a limited form of constant folding over strings that compose the value. $\boxed{\texttt{SA}}$ marks the function as a database procedure, if the resolved value is a subclass of database API.

- **Variable:** If the function returns a variable, $\boxed{\texttt{SA}}$ iterates backward on the AST to the last assignment of the variable and checks whether the assignment is a class instantiation or not. If it is a class instantiation, $\boxed{\texttt{SA}}$ tries to resolve the type of instantiated object as described above.

As discussed in Section 2.3, PHP web applications often use variables as an argument for creating objects from classes using the `new` keyword. During analysis, $\boxed{\texttt{SA}}$ keeps track of arguments passed to `new` in PHP scripts using a string representation.

String Representation

$\boxed{\texttt{SA}}$ encounters strings when handling variable assignments and constant definitions. Strings can be a mixture of literal components, function return values, and variables.

When $\boxed{\texttt{SA}}$ iterates over an assignment node in the AST, it records a set of information from the assignment node in a hash table. $\boxed{\texttt{SA}}$ keeps track of the name of the variable and the components on the right side of the assignment. $\boxed{\texttt{SA}}$ also records the name of the function or the name of the class and method that the assignment statement occurs in. For example, in Listing 5.1(b) at Line 21, the function `executeQuery` has an assignment statement. The right side of the assignment concatenates a constant string and a return value from a function. $\boxed{\texttt{SA}}$ records the name of the variable on the left side of the assignment as well as the value of the constant string and the return value from the function. $\boxed{\texttt{SA}}$ also records the type of operation on the right side (as discussed next, it is a concatenation operation). $\boxed{\texttt{SA}}$ implements common string operations to resolve the value of the assignment.

String Operations

SQLBlock manages frequent string-related operations.

Variables: The argument passed to `new` can contain variables defined in the script. $\boxed{\texttt{SA}}$ keeps track of variable definition in the scope of script, class, or functions. When there is a variable assignment, $\boxed{\texttt{SA}}$ creates an object for the variable and its value.

Concatenation: In PHP, strings can be constructed by joining multiple components with the `.` and `.=` operators. $\boxed{\texttt{SA}}$ handles string concatenation by creating an object for concatenation and adds components that exist in the concatenation statement.

Identifying Database Procedures

To identify database procedures, $\boxed{\texttt{SA}}$ iterates over the assignments and resolves the value of variables in the strings by looking for variables in the same class and function. If there is a variable without a value, $\boxed{\texttt{SA}}$ represents the value as a regular expression `.*` wildcard. $\boxed{\texttt{SA}}$ looks for a match between the generated regular expression and the list of database API subclasses. For example, in Listing 5.1(b), line 21, $\boxed{\texttt{SA}}$ cannot determine the return value of `$this->getDriver`. Instead, $\boxed{\texttt{SA}}$ represents the value as a `.*` wildcard. $\boxed{\texttt{SA}}$

searches the list of database API subclasses for a class that matches the regular expression `DatabaseConnection*`, and finds such a class named `DatabaseConnectionmysqli`. $\boxed{\texttt{SA}}$ marks *executeQuery* as a database procedure.

At the end of this step, $\boxed{\texttt{SA}}$ has a list of database access layer classes, interfaces, and procedures.

5.2.2 Profile Data Collection

This step trains SQLBlock to create a mapping between issued SQL queries and the web application's function that relied on the database access layer to issue the SQL query. The collected information in this step is necessary for generating query descriptors in step ③. As described in Section 5.1, the information collected for each SQL query contains the operation, the access tables, the logical operators, the SQL functions that the query used, and the type of arguments in each SQL function.

Attaching a PHP call stack:

When MySQL receives a SQL query, SQLBlock must infer which PHP function actually issued the SQL query. To achieve this, I modified the source code of the MySQL driver for the `PDO` and `mysqli` extensions. This modification appends the PHP call-stack at the end of the query as a comment before sending it to the database.

To access the PHP call-stack, I use the Zend framework's built-in function called `zend_fetch_debug_backtrace`. Zend keeps the information regarding the call-stack for the executing PHP script. This information includes the functions, class, their respective arguments, the file, and the line number that issued the call. The modified database extension ($\boxed{\texttt{DE}}$) extracts the PHP call stack and appends it as a comment to the end of the SQL query.

```
SELECT * FROM public_info where id > 0 # mysqli::
    multi_query@DatabaseConnectionmysqli::
    multi_execute@executeQuery@get_public_info
FIELD@FUNC:>@2@FIELD@LITERAL # recorded info regarding the nodes in
    the SQL query
public_info@0 # recorded info regarding the table and operation
    type of the SQL query
```

Figure 5·2: recorded information for the execution of `get_public_info`

Extracting information from the parse tree:

Recorder plugin (`PR`) acts as a post-parse MySQL plugin. `PR` has access to various information regarding the parsed SQL query in MySQL: the type of operation (e.g. `SELECT` operation, etc.), the name of the table, and the parse tree of the SQL query. MySQL provides a parse tree visitor function that `PR` uses to access the parse tree of SQL queries.

However, MySQL only allows plugins to access literal values of the query, such as user inputs in the parse tree. Because SQLBlock needs more information regarding the parsed SQL query, we modified the source code of MySQL-server so that the plugin can access non-Literal values as well. When MySQL invokes `PR`, `PR` records the SQL query that MySQL receives. Afterwards, `PR` iterates over the parse tree of the SQL query and records the type of each node. If the node represents a SQL function in the SQL query, `PR` also records the number of arguments used in the SQL function. The node that represents the SQL function in the SQL query also holds the number of arguments used in the SQL function. Afterwards, `PR` records the type of arguments passed to the SQL function as they appear in the parse tree of the SQL query. Lastly, `PR` logs the table and the type of operation for the SQL query that MySQL received. MySQL shows the type of operation for a SQL query as a number. Hence, `PR` logs the type of operation for a SQL query as an encoded number in the profile. Figure 5·2 shows the recorded information in the profile, when function `get_public_info` executes.

At the end of step ②, SQLBlock has detailed information on the received SQL queries for training.

5.2.3 Creating the Profile

In step ③, profile generator ($\boxed{\texttt{PG}}$) creates a profile for each PHP function in the web application that accesses the database. $\boxed{\texttt{PG}}$ relies on the training data from step ② as input.

$\boxed{\texttt{PG}}$ reads the recorded information from step ②. As shown in Figure 5·2, the first line is the SQL query including the PHP call-stack. Using the list created in step ①, $\boxed{\texttt{PG}}$ must infer which PHP used the database access layer to send the SQL query to the database. This is a difficult problem, because the last function on the call stack might be a helper function that issues all queries for the application (and, in fact, this is how modern real-world PHP applications such as Wordpress and Joomla are written). $\boxed{\texttt{PG}}$ iterates over the stack of functions in the PHP call-stack and checks whether the function or the method was recognized as a database procedure or database API method in step ①. $\boxed{\texttt{PG}}$ iterates over the stack starting from the last call in PHP call-stack until a function is not a database procedure or database API method. $\boxed{\texttt{PG}}$ identifies this function as the function that created the database query.

As an example, the Line 1 in Figure 5·2 shows the SQL query that MySQL receives including the PHP call-stack. $\boxed{\texttt{PG}}$ detects `mysqli` as a database extension in PHP and `DatabaseConnectionmysqli` as a class that extends `mysqli`. Then, $\boxed{\texttt{PG}}$ visits the next function `executeQuery`, which was identified as a database procedure in step ①. The next function in the PHP call-stack is `get_public_info`. `get_public_info` is not in the list of database procedures from step ①, therefore $\boxed{\texttt{PG}}$ identifies it as the PHP function that used database access layer to send the SQL query to the database. $\boxed{\texttt{PG}}$ will then update `get_public_info`'s query descriptor.

Afterwards, $\boxed{\texttt{PG}}$ iterates over the nodes of the SQL query's parse tree and extracts all the logical operators. If all the logical operators are the same, $\boxed{\texttt{PG}}$ updates the *cond* with the respective value. If both logical operators (i.e, both *OR* and *AND*) are in the nodes of SQL query's parse tree, $\boxed{\texttt{PG}}$ sets *cond* to *"Both"*. If there is no logical operators in the SQL query, $\boxed{\texttt{PG}}$ sets *cond* to *"None"*. Based on Figure 5·2, $\boxed{\texttt{PG}}$ specifies that `get_public_info` does not use any logical operators in its SQL query.

$\boxed{\texttt{PG}}$ iterates over the list of nodes from the parsed tree of the SQL query and extracts the name of the used functions in the SQL query as well as their respective arguments. Since the number of arguments passed to the SQL function can be variable, $\boxed{\texttt{PG}}$ does not record each argument's type. Instead, $\boxed{\texttt{PG}}$ summarizes the types of arguments that a SQL function relies on. There are multiple types of functions in MySQL such as numeric, string, comparison, and date function. All of the aforementioned types of SQL functions except the comparison type either receive less or equal to two arguments or modifies the content of the first argument passed to the function. Comparison functions in MySQL (e.g., `<`, `IN`, etc.) compare a single argument to a variable sized argument array. Moreover, the single argument appears as the first argument in the SQL comparison functions. Owing to this, $\boxed{\texttt{PG}}$ records the type of the first argument passed to a SQL function separately. If the argument is a table column, $\boxed{\texttt{PG}}$ records it as a `FIELD` argument, otherwise $\boxed{\texttt{PG}}$ records it as a `LITERAL` argument. Afterwards, $\boxed{\texttt{PG}}$ iterates over the rest of the arguments passed to the SQL function. If the type of all the other arguments are the same type (i.e., `FIELD` or `LITERAL`), then $\boxed{\texttt{PG}}$ records the value of the respective type in the profile. Otherwise $\boxed{\texttt{PG}}$ sets the type as *var*. For instance, based on Figure 5·2, $\boxed{\texttt{PG}}$ specifies that function `get_public_info` used function `">"`, that the first argument is a table column and the second argument is a `LITERAL`.

Lastly, $\boxed{\texttt{PG}}$ reads the information about the name of the table and the type of SQL query. For instance, based on line 3 in Figure 5·2, $\boxed{\texttt{PG}}$ deduces that function `get_public_info` accesses the table `public_info` using a `0`-type SQL query (i.e., `SELECT` SQL query).

At the end of step ③, $\boxed{\texttt{PG}}$ has a set of query descriptors for each PHP function in the web application that issued a SQL query during training in step ②

5.2.4 Protecting the Web Application

In step ④, the enforcer plugin ($\boxed{\texttt{PE}}$) is on enforcement mode. $\boxed{\texttt{PE}}$ uses the profile that was generated in step ③ and protects the database from queries that deviate from the profile. Similar to $\boxed{\texttt{PG}}$, $\boxed{\texttt{PE}}$ is implemented as a postplugin, which gives it access to the parse tree of the received SQL query. $\boxed{\texttt{PE}}$ also uses the same PHP database extensions as described in Sectin 5.2.2. $\boxed{\texttt{PE}}$ reads the profile for each PHP function and uses it to analyze the received queries.

After receiving a query, MySQL parses the SQL query and calls $\boxed{\texttt{PE}}$. $\boxed{\texttt{PE}}$ locates the call-stack and extracts the PHP function that issued the query with the same approach described in Section 5.2.3. Afterwards, $\boxed{\texttt{PE}}$ finds the query descriptors in the profile associated with the PHP function. $\boxed{\texttt{PE}}$ checks the query against all four components of each query descriptor found for the PHP function. For operation type, $\boxed{\texttt{PE}}$ checks whether the received SQL query has the same operation type as it is recorded in the profile. $\boxed{\texttt{PE}}$ also examines that the list of tables accessed for the received SQL query is a subset of table access listed in the query descriptor. The logical operators used in the received SQL query must be a subset of the logical operators in the query descriptor. Finally, the received SQL query can only use a subset of functions listed in the query descriptor. $\boxed{\texttt{PE}}$ also checks whether the arguments passed to each function has the same type as it is recorded in the query descriptor. Only if the SQL query matches with all four components of at least one query descriptor in the profile, $\boxed{\texttt{PE}}$ allows MySQL to execute the SQL query and return the results. Otherwise $\boxed{\texttt{PE}}$ returns `False` to MySQL-server, aborting execution of the query and returning an error to the web application, thus preventing a potentially malicious attacker-controlled SQL query from executing.

5.3 Evaluation

I assessed the ability of SQLBlock to prevent SQLi attacks on a set of popular PHP web applications. I also examined SQLBlock's false positive rate during the benign browsing of the web application. Additionally, I evaluated the performance overhead of step ③ for the benign browsing. For this evaluation, I answer the following research questions:

RQ1 How precise is SQLBlock's static analysis?

RQ2 Is SQLBlock effective against real world SQLi vulnerabilities in popular web applications?

RQ3 How practical is SQLBlock regarding performance overhead and false positives?

5.3.1 Evaluation Strategy

In this evaluation, I performed the static analysis once for each web application in Section 5.3.2. I evaluated the database access layer resolved by the static analysis in `RQ1`. Then I leveraged the database access layer to answer `RQ2` and `RQ3`. I trained and built the profile for SQLBlock using the official unit tests of each web application once and used the generated profile for the experiments to answer `RQ2` and `RQ3`. The official unit tests examine the correctness of functions in the web application by executing test-inputs and verifying their results. The advantage of unit tests over web crawlers is that there is no need for manual intervention of administrators, specifically for providing semantically correct inputs for each form in web applications. A web application's unit tests are specifically tailored to its implementation and therefore are likely to achieve higher code coverage. Figure 5·3 shows that Drupal's unit tests achieve higher line coverage compared to Burp suite and also covers almost all the lines that Burp Suite (PortSwigger, 2019) covered. However, alternative approaches such as web crawlers can also be used for training SQLBlock.

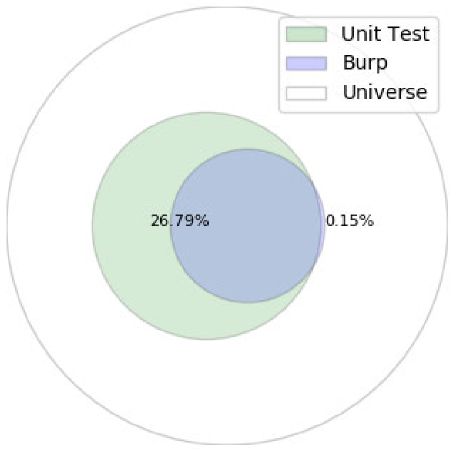

Figure 5·3: The line coverage for unit tests and Burp suite on Drupal 7.0

5.3.2 Evaluation Dataset

I evaluated SQLBlock on the four most popular PHP web applications, Wordpress, Joomla, Drupal, and Magento. According to W3Techs, these web applications hold 70.5% of the market share among all existing content management systems (CMS) and power 38.4% of all the live websites on the Internet combined (Q-Success, 2023b). Administrators install plugins and additional components to customize the web application and extend its functionality. To reflect this behavior in the evaluation, I also evaluate SQLBlock on plugins. I installed four vulnerable Wordpress plugins called *Easy-Modal*, *Polls*, *Form-maker*, and *Autosuggest*. I also installed three vulnerable plugins in Joomla named *jsJobs*, *JE photo gallery*, and *QuickContact*. To assess the defensive capability of SQLBlock, I selected recent versions of the web applications and plugins that contain known SQLi vulnerabilities. I also considered the type of SQLi vulnerability in the dataset to include all types of SQLi exploits for a comprehensive evaluation. I collected a total of 11 SQLi vulnerabilities in different web applications and plugins.

5.3.3 Resolving The Database Access Layer (RQ1)

In step ①, SQLBlock scans the PHP web application to identify the database access layer that is used to communicate with the database. Step ① is a crucial step to identify the correct function in the PHP call-stack that relies on the database access layer for interacting with the database.

Table 5.1 presents the resolved database access layer statistics. The *resolved subclasses* column specifies the number of classes that extends the database API in PHP. The *resolved database procedures* column presents the number of functions that returns an object from a subclass of the database API. Since there is no ground truth for the database access layer in the web applications, I manually analyze the output of $\boxed{\texttt{SA}}$ for true positives. Subclasses of database APIs in the PHP web applications also implement interfaces to facilitate actions such as iterating over elements in the object and counting elements. For instance, Drupal implements `Iterator` and `Countable` so that the PHP script can iterate over or count the number of records that the database returns to the PHP script. Since Drupal implements `Countable` and `Iterator` in the subclasses of database API, $\boxed{\texttt{SA}}$ adds these two interfaces to the database access layer. As shown in Table 5.1, the only false positives I observed during the evaluation are caused by the `Iterator` and `Countable` interfaces. All the web applications in the dataset except for Wordpress, use encapsulation in their database API subclasses and database procedures that show the necessity of identifying the database access layer for creating a profile. Without identifying the database access layer, SQLBlock would operate similar to SEPTIC and map the received queries to a single identifier.

Web application	**Resolved subclasses (FP)**	**Resolved database procedure**
Wordpress 4.7	1	-
Drupal 7.0	44 (2)	38
Joomla 3.7	30 (0)	-
Joomla 3.8	30 (0)	-
Mangeto 2.3.0	15 (0)	-

Table 5.1: Resolved database access layer by SQLBlock's static analysis.

5.3.4 Defensive Capabilities (RQ2)

I assessed the defense capabilities of SQLBlock against 11 SQLi vulnerabilities listed in Table 5.2. I built and deployed five Docker containers that run a vulnerable version of a web application and a plugin. I exploit the vulnerabilities using exploits from Metasploit Framework (Metasploit, 2019), exploit-db (Security, 2019), and sqlmap (Bernardo and Miroslav, 2019). I consider an attack successful if an attacker can inject malicious SQL code into the generated query in the web application and the database executes the malicious SQL query.

For this evaluation I used the results of the static analysis in `RQ1`. I trained SQLBlock using the official unit tests of web applications in their respective repositories. After creating the profile, I configured SQLBlock in the enforcement mode and assess whether the exploits in exploit-db and Metasploit Framework are successful or not. Adversaries are not limited to use exploits in the evaluation and can craft their SQL queries to circumvent SQLBlock. To evaluate the potential of such attacks, I also used sqlmap (Bernardo and Miroslav, 2019) to generate various exploits for the vulnerabilities listed in Table 5.2.

In Table 5.2, I present the list of SQLi vulnerabilities that SQLBlock defends the web applications against. The second column in Table 5.2, represents the ID assigned to each vulnerability. I marked the SQLi vulnerabilities that reside in the core of web applications by ©. The third column shows the type of attacks I performed to exploit the respective vulnerability. SQLBlock protects the web applications against all 11 SQLi exploits in the dataset, while SEPTIC can only defend against four SQLi exploits that only reside in Wordpress plugins.

To evaluate the potential of circumventing SQLBlock, I also listed the available query descriptors for the SQL queries that the vulnerable PHP function in each web application or plugin can issue. For instance, any potential exploit against the first vulnerability in Table 5.2 is restricted to an `UPDATE` query exclusively on table `wp_em_modals` without further logical operators. Furthermore, the exploit can only use SQL functions `"="` and

Application	Vulnerability	SQLi Type and Available query descriptors
Wordpress 4.7	CVE-2017-12946	Taut., Infer., Alt. Encoding (update, wp_em_modals, none, [(=,field,literal),(IN,field,literal)])
Wordpress 4.7	polls-widget 1.2.4	Taut., Infer., Alt. Encoding (update, wp_polls, none, [(=,field,lietral)])
Wordpress 4.7	CVE-2019-10866	Infer. (select, wp_formmaker_submits, and, [(=,field,literal)])
Drupal 7	WPVDB-9188	Taut., Infer. (select, wp_posts, and, [(=,field,literal)])
Joomla 3.7	CVE-2014-3704 ©	Taut., Union, Piggy-back, Stored Proc., Infer., Alt. Encoding (select, users, and,[(=,field,literal)])
Joomla 3.8.3	CVE-2017-8917 ©	Union, Infer.,Alt. Encoding (select, [users, languages, fields], both, [(=,field,literal),(=, field, field),(IN, field, literal)])
Joomla 3.8.3	com_jsjobs 1.2.5	Infer. (select, js_jobs_fieldsordering, none, [(=, field, literal)])
Joomla 3.8.3	com_jephotogallery 1.1	Union, Infer. (select, jephotogallery, none, [(=, field, literal)])
Joomla 3.8.3	CVE-2018-5983	Infer. (select, jquickcontanct_captach, none, [(=, field, literal)])
Joomla 3.8.3	CVE-2018-17385 ©	Second order inj. (select, template_styles, and, [(=, field, literal)])
magento 2.3.0	CVE-2019-7139 ©	Infer., Alt. Encoding (select, catalog_product_frontend_action, and, [(>=, field, literal),(<=, field, literal)])

Table 5.2: Exploits blocked by SQLBlock.

`"IN"`.

5.3.5 Performance (RQ3)

Performance/responsiveness is a crucial factor for web applications. Therefore, I evaluate SQLBlock's performance overhead. In SQLBlock, the first three steps can be performed offline. Steps ① and ③ are automatic and do not rely on help from the administrator. In step ②, the administrator must perform unit tests or create benign traffic in the web application to train SQLBlock. Step ④ is deployed as a MySQL plugin and a set of modified PHP database extensions to sandbox databases against malicious SQL queries. The MySQL server loads SQLBlock's protection plugin upon launch. SQLBlock loads the profile and waits for incoming SQL queries. I perform the experiments on a 4-core Intel Core i7-6700 with 4Gb of memory 2133Mhz DDR4 that runs Linux 4.9.0, with Nginx 1.13.0, PHP 7.1.20, and MySQL 5.7.

For the performance evaluation, I created a Docker (Docker, 2018) container that runs with a default configuration of PHP, Nginx, and MySQL containing the Drupal 7.0 web application. I measure the performance overhead of SQLBlock using ApacheBench (Apache Software Foundation, 2018), a tool for benchmarking HTTP web servers. I simulated a real-world scenario by increasing the level of concurrency in ApacheBench. The level of concurrency shows the number of open requests at a time. I measured the network response time of `index.html` in Drupal 7.0 that issues 26 queries to MySQL. For more precise results, I measured the response time for 10,000 requests at multiple levels of concurrency. Table 5.3 presents the results for the aforementioned scenario. The first column in Table 5.3 shows the level of concurrency for each test. The next two columns in Table 5.3 present the network response time for Drupal with/without SQLBlock. As shown in Table 5.3 SQLBlock incurs less than (2.5%) overhead to the network response time of the server. Based on the strong protections afforded by SQLBlock, I consider this overhead acceptable. Furthermore, SQLBlock is a prototype with no emphasis on performance optimization. Such optimizations likely could reduce the overhead even further.

I also measured the execution time of queries in MySQL. I modified the source code of MySQL to calculate the time it takes for MySQL to execute a SQL query. For this experiment, I used ApacheBench to send 10,000 requests to `index.html` in Drupal 7.0, which issued a total of 260,000 queries to MySQL. I measured the average execution time of issued queries for two different scenarios. The first scenario is MySQL without SQLBlock's plugin, and in the second scenario, I enabled SQLBlock's plugin in MySQL. The last two columns in Table 5.3 present the average execution time of all the received queries to MySQL. The performance overhead of SQLBlock in MySQL is less than 0.31 ms for each query.

	Server Response Time(ms)		MySQL Execution Time(ms)	
Concurrency	Unprotected	Protected	Unprotected	Protected
1	27.792	28.338 (1.96%)	0.150	0.23
4	11.644	11.813 (1.45%)	0.669	0.90
8	8.907	9.127 (2.46%)	0.732	1.02
16	8.885	9.084 (2.23%)	0.740	1.05
32	8.971	9.182 (2.35%)	0.747	1.02

Table 5.3: Response times for requests to Drupal index.php

5.3.6 False Positive Evaluation

I count an operation as a false positive if SQLBlock blocks a benign query to the database. For the false positive evaluation, I evaluated SQLBlock with Wordpress 4.7 and Drupal 7.0. For each web application I used the profile built in `RQ2`. Then, I configured SQLBlock in enforcement mode and replayed browsing traces collected by Selenium (Balde Samit, 2018). The browsing traces explored the web application as a user and administrator with the goal of covering the web application as much as possible.

Based on Table 5.4, only 10.11% of the issued queries during benign browsing and the unit test had the same query structure. This legitimate difference in the query structure of issued queries renders prior approaches that build their profile based on query structure unable to distinguish benign SQL queries from malicious ones. For instance, SEPTIC has above 89% false positive on the same test for Drupal 7.0. SQLBlock allows a query to execute in MySQL as long as the query matches at least one of the query descriptors associated with the PHP function in the profile. In the false positive test for Drupal, SQLBlock did not block any query from the benign Selenium browsing. This shows that although the PHP functions during training and testing used different queries, the query descriptors were the same.

Table 5.4 shows that 82.57% of queries in the benign browsing were similar to queries recorded for SQLBlock's profile. Although the rate of similar issued queries during training and testing of Wordpress is higher that Drupal, SQLBlock blocked 7 unique queries during the benign browsing, which corresponds to 5% of all issued queries. There are two main reasons for the false positives in Wordpress. The first reason is MySQL modifying the query based on the arguments passed to SQL function in the query. For instance, if the length of the array passed to the `IN` statement in a query is one, MySQL modifies the `IN` statement to an `equal (=)` statement. This modification in the query and subsequently in the parse tree of the query leads to false positives for SQLBlock since SQLBlock encounters a different function in enforcement than what is in the profile. The second reason is missing PHP functions in the profile. During the enforcement, SQLBlock blocks the SQL query if SQLBlock does not find any query descriptor for a PHP function that issued the SQL query. Six out of seven false positives in Wordpress was due to lack of query descriptors for the PHP function during benign browsing, which implies that covering all the functions that can issue a query during the training is an important factor for SQLBlock.

web application	Unit tests	Selenium	(Unit tests ∩ Selenium)	False Positive
Drupal	299961	336	34 (10.11%)	0
Wordpress	3099	132	109 (82.57%)	7

Table 5.4: Number of unique SQL queries during unit testing and Selenium browsing

5.4 Discussion and Limitations

In this section, I discuss the limitations of the SQLBlock and possible future works in this area.

eval Function: PHP web applications use dynamic features implemented in PHP extensively, such as the `eval` function, which evaluates a string argument as a PHP code. Currently, SQLBlock does not handle function and class definitions inside `eval`. A web application can use `eval` for defining the database API or procedures dynamically and use it across the web application. This leads to generating a non-complete list of PHP database API and interfaces for a PHP web application in the step ①. In such cases, SQLBlock maps the query descriptors to a small set of PHP functions that can allow the attacker to execute a malicious query. In future work, the static analyzer in SQLBlock can be improved to handle the static PHP code passed to `eval`, to determine a more precise database access layer.

Incomplete coverage during training: PHP web applications generate dynamic queries based on user inputs. This approach makes it impossible to issue all possible queries to the database during the training phase. Dynamic analyses suffer from incomplete training phases, and SQLBlock is not an exception. The Wordpress false positive test shows that the incomplete coverage of the issued queries leads to SQLBlock blocking benign queries.

Chapter 6

Debloating PHP web application through static analysis

In this chapter, I describe my third contribution to debloat the source-code of the PHP web application. First, I discuss the threat model and environmental conditions. Next, I elaborate the system architecture of the debloating mechanism in Section 6.1. Then, I evaluate my approach on most popular PHP web application in terms of reducing the size of source-code and removing security vulnerabilities from the PHP web applications in Section 6.2.

Threat Model and Environmental Conditions. The threat model targets PHP web applications which may contain unknown security vulnerabilities running atop a non-compromised OS. I assume that the administrators cannot host their web applications with the profiler code turned on due to its high overhead and negative effect on page-load time. My assumption also entails that operators/developers/administrators can invest time in developing custom static analysis which Minimalist then uses to debloat web applications. As the evaluation in Section 6.2 shows that Minimalist can remove up to 38% of security vulnerabilities from PHP web applications with a minimal effort from developers.

6.1 System Architecture

In this contribution, I aim to debloat PHP web applications. My tool consists of three main steps: 1) Generating a call-graph for the selected PHP web application. 2) Pruning the call-graph based on the PHP files that users (via their HTTP requests) accessed. 3) Debloating the unreachable functionality from the web application. I implemented this approach for PHP web applications in a prototype called Minimalist.

Call-graph in conjuction with the application entry-points can be used to identify unused functions within a web application, which can then be removed via debloating. The soundness and accuracy of the call-graphs directly affect the performance and correctness of debloated web applications. Leveraging an unsound call-graph to debloat applications can lead to false positives (i.e., removing parts of the code that are needed by the application). However, over-approximations due to the imprecise resolution of dynamic code constructs lead to the generation of a call-graph that, despite being sound, is unusable for debloating. For example, a fully connected graph (i.e., connecting all pairs of functions in the application) is trivially sound, but is not useful for debloating since every function is reachable from every entry-point. Therefore, over-approximation and lack of precision during the call-graph generation leads to degraded debloating results (i.e., keeping pieces of code that are not used in practice).

The system takes a multi-step approach to construct the call-graph by leveraging three analyses of inheritance, variables, and script inclusions to handle the dynamic features that the web application uses to invoke a function. Figure 6·1 demonstrates the overall architecture of my system. In this section, I first explain each analysis and how Minimalist combines the results to generate a call-graph. Finally, I explain the pruning process of the generated call-graph and the eventual debloating of the given PHP web application.

6.1.1 Generate the call-graph

The first step for Minimalist to debloat a PHP web application is to represent it in the form of a call-graph. To generate the call-graph, Minimalist performs three preliminary analyses on the web application to handle dynamic features in the PHP web applications: 1) Class Hierarchy Analysis, 2) Variable Analysis, 3) Script Inclusion Analysis. My tool uses the php-parser library (Slizov, 2019) to parse each PHP script in the web application

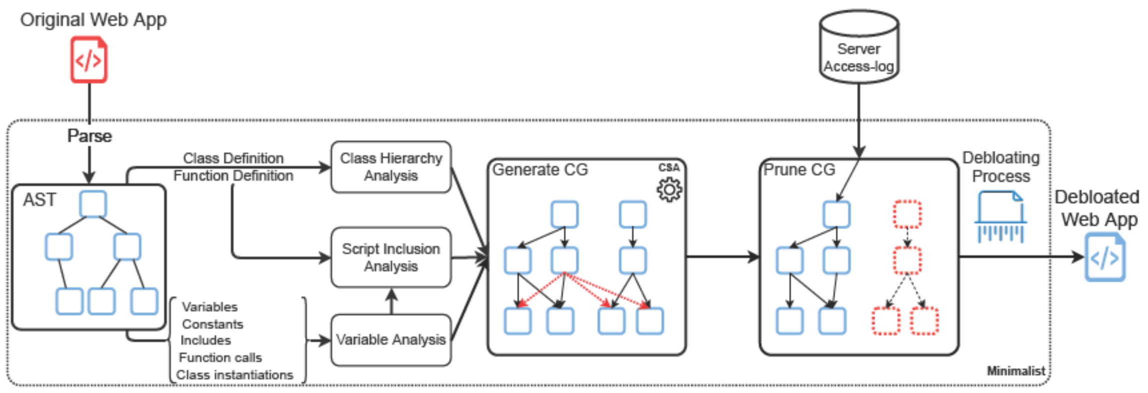

Figure 6·1: Minimalist statically generates a call-graph from the web application under analysis, prunes the call-graph of the web app based on a server access-log, and removes the unused functionality from the web application.

into its corresponding Abstract Syntax Tree (AST) and then performs each analysis. Next, I discuss the details of each analysis and how Minimalist incorporates this information to generate the call-graph.

Class Hierarchy Analysis

In this step, Minimalist performs the class hierarchy analysis on the given PHP web application. This analysis allows Minimalist to identify the inheritance relationships between the implemented classes in the web application. For the class hierarchy analysis, Minimalist identifies the class definition statements by iterating over the AST nodes of each PHP script. In a class definition statement, Minimalist extracts the name of the defined class and the extended class, which follow the keywords `class` and `extends` respectively. My tool generates a global hashmap called `Inherit`, where the key is the defined class and its value is the parent class name.

Variable Analysis

In this step, Minimalist performs a flow-insensitive analysis on the source code of the target web applications. PHP applications often use dynamic features such as variable invocation and script inclusion to deliver dynamic content. This analysis allows Minimalist to correctly resolve the list of target functions in dynamic invocations and included files in further analyses. The variable analysis in Minimalist involves tracking assignment statements in the web application and recording the assigned values in a hashmap. In this analysis, a variable can take any of the following values:

- **Constant:** The assignment statement contains only constant values.
- **Unbound:** The variable analysis cannot restrict the possible values for a variable such as assignments based on user-input.
- **Mixed:** The assigned value to the variable is a mixture of constants and unbounded values.

Each assignment statement is comprised of three components: the left hand side (lhs), the right hand side (rhs), and the operation. Minimalist tracks the variable assignments for each PHP script in a separate hashmap structure named `ValueSet`. In this hashmap, the key is the name of the variable on the lhs and the value is the string representing the assigned value. For each assignment, I resolve the lhs expression to extract the name of the target variable, which includes variables, arrays, and class property assignments. Similarly, the rhs is resolved iteratively by traversing the AST nodes provided by the PHP parser. The rhs is resolved to a string representing the assigned value. In PHP, variables are scoped. According to the PHP documentation (PHP, 2022), there are three different variable scopes in PHP: 1) global, 2) local, and 3) static. A variable's scope is global when the variable is defined outside a PHP function. Furthermore, a variable defined inside a function is by default limited to the local function scope. Similar to local scope, a static variable can only be accessed inside the local function scope (PHP, 2022). Minimalist conservatively promotes all variables to the global scope and combines the resulting `ValueSets` (i.e., set union) of all variables that share the same name, irrespective of the variables' scopes. This approach leads Minimalist to over-approximate the possible values a variable can hold. I categorize the rhs expressions into six groups, which Minimalist handles as follows:

Literal: There is no further analysis on string literals.

Magic Constants and PHP built-in functions: The PHP interpreter defines a set of constants with predefined values such as `__dir__` and `__function__`, which refer to the current directory and current function, respectively. Minimalist models the commonly-used PHP file operations functions such as `dirname` as well as magic constants. This way, Minimalist can resolve dynamic file inclusions statically.

Object Instantiation: For object instantiation statements, Minimalist extracts the name of the instantiated class and determines the type of the instantiated object.

If Minimalist cannot reason about the type of an object in rhs, it marks the variable as "unbound".

Variables: If the rhs contains a variable, this means that the variable must have been initialized previously in the web application. In this case, Minimalist resolves each variable on the rhs by looking up its assigned value in the `ValueSet` hashmap. If I cannot find the variable in the hashmap for the current script, I perform a global search across other PHP scripts for its definition. If the variable is not found, Minimalist marks the variable as "unbound".

User-defined Function Call: Minimalist only resolves direct function calls used in assignment statements. To do this, Minimalist identifies the implementation of the invoked function in the web application, and analyzes the return statement inside the function's body. Minimalist iteratively analyzes the AST nodes of the return statement, similar to the analysis of the assignment statements. Next, I translate the sequence of nodes that compose the return statement into a string representing the returned value. I then replace the function call in the assignment statement with this string. In the case of recursion in function calls, Minimalist only analyzes the return statement once. If Minimalist cannot determine the target function (i.e., variable invocations) or its returned value, it marks the return value as "unbound".

Unbound: For any other type of node that does not belong to the above categories, Minimalist marks the string representation of the node as "unbound".

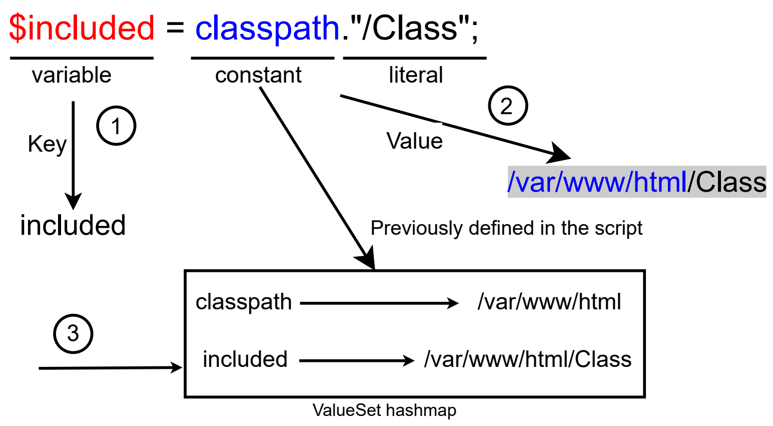

Figure 6·2: Minimalist analyzes the assignment statements by 1) Extracting the name of the variable from LHS, 2) Resolving the RHS to a string representing the assigned value, 3) Storing the mapping in the ValueSet hashmap.

Minimalist applies this procedure recursively on every node of the rhs in the assignment statement until it includes only literals, unbounds, and string concatenation. Minimalist then translates the rhs into a regular expression (regex). In doing so, Minimalist over-approximates the unbounds (e.g., user-input, database records) by replacing them with a wildcard (.*) in the generated regex. Over-approximating the values of variables in this step allows Minimalist to include all possible values assumed by a variable in the regex. In the case of multiple assignments to the same variable, Minimalist joins the regexes for each option with the `or` operator. In the end, Minimalist creates an entry in the `ValueSet` hashmap with the name of the variable as the key and the generated regex as the value. Figure 6·2 demonstrates how Minimalist analyzes different types of nodes in the AST for the assignment statements in Listing 2.1 and stores the mapping of their values in the `ValueSet` hashmap.

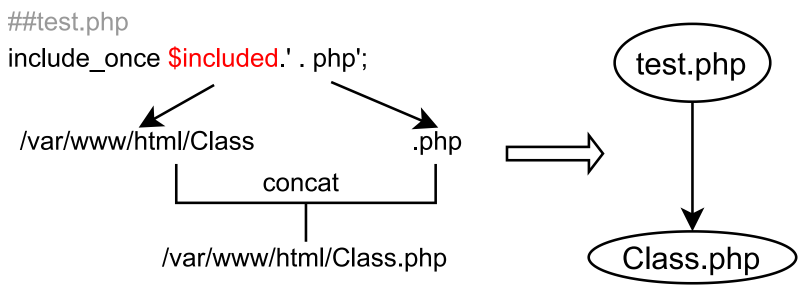

Figure 6·3: An example include resolution by Minimalist.
Minimalist analyzes the include statements and generates a script dependency graph for the web application under analysis.

Analyze Script Inclusions

In this analysis, Minimalist generates a script dependency graph for the PHP web application. A script dependency graph is a directed graph where the nodes are the files in the web application and a directed edge between two nodes (i.e., two files) represents the inclusion of scripts. The PHP interpreter always executes the main body (i.e., global context) of an included script. This is critical to constructing the call graph since each included script can invoke a series of functions or include other scripts. Minimalist iterates over the AST of each script in the PHP web application and identifies script inclusion expressions. For each script inclusion expression, Minimalist iterates over all the nodes that compose the string passed to the expression.

Minimalist handles dynamic script inclusions in web application by resolving the value of variables using the variable analysis results. If there is a variable in the argument, I replace the variable with its value in the `valueSet` hashmap. Next, Minimalist translates the sequence of arguments into a regex. Finally, I draw edges between the file under analysis and every file that matched the regex. If the passed argument to the script inclusion function is a wildcard regex, we draw edges from the file under analysis to every file in the web application. In Figure 6·3, I demonstrate how Minimalist resolves the arguments into the files matching the regex, and generates the script dependency graph for Listing 2.1.

PHP scripts frequently use auto-loaders to instantiate objects from classes without explicitly including the file which contains the implemented class. Minimalist handles auto-loaded classes in scripts by analyzing the `new` expression used for instantiating the object. As with resolving the argument passed to include statements, Minimalist resolves the arguments passed to the `new` expression. Afterward, Minimalist draws a dependency edge from the current file under analysis to the script(s) which contain(s) the class implementation(s).

Generate the Call-graph

In this step, Minimalist generates a call-graph for the web application under analysis. To generate the call-graph, Minimalist must identify the caller-callee relationships between functions in the web application. This is accomplished by iterating over the AST of each PHP file, to identify function call expressions residing in the caller. The target of this expression is to identify the callee. As callers and callees are functions, Minimalist maintains a special caller corresponding to the global script context (i.e., function invocations not part of a function body). For direct invocations, Minimalist adds a node to the call-graph for the caller and callee, if they do not exist, and draws an edge from the caller to the callee.

In case of variable invocations, Minimalist leverages the collected information in the variable and class hierarchy analysis to resolve the values of variables. Minimalist extracts the nodes that compose the variable and performs a lookup in the `ValueSet` hashmap to find the regex for the assigned values. For keywords within the object's context (e.g., `parent`), Minimalist uses the `Inherit` mapping to replace the keyword with the name of the current class or its parent. Next, Minimalist resolves the variable invocation by matching the regex against all defined functions and methods in the web application. Finally, Minimalist draws edges in the call-graph between the caller and each of the matching functions. Note that the over-approximation of variable values in the variable analysis leads Minimalist to draw edges to every possible invoked function at each call-site. Figure 6·4 demonstrates how Minimalist resolves the assigned values to variables in a function call and draws the edge

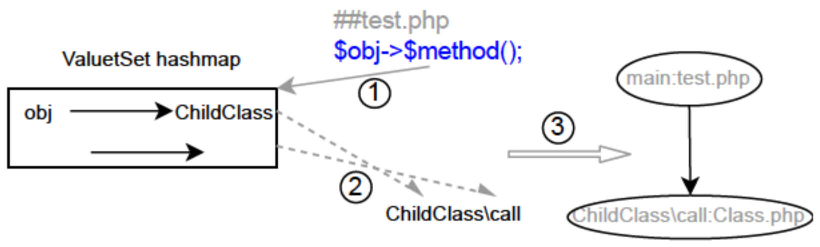

Figure 6·4: Minimalist uses the information from previous analysis, 1) Looks up for variables values in `ValueSet` hashmap, 2) Retrieves and replaces the values in the function call , and 3) Draws the associated edges in the call-graph. `main:test.php` represents the global scope of the `test.php` script.

between the caller and the callee in the call-graph.

If the variable involved in the variable invocation is unbound (i.e., wildcard (.*)), Minimalist cannot resolve the function call to a subset of defined functions in the web application. In such a case, I draw edges to every defined function in the web application. Minimalist also creates a report of the unresolved instances, including the target file, function, and line number. This report provides the necessary information for implementing custom static analysis (described in Section 6.1.1).

Furthermore, Minimalist models the set of higher-order functions provided by the PHP interpreter that take the name of a function as an argument, which is then invoked by the interpreter. Higher-order functions affect the call-graph by invoking the functions passed as arguments. Hence,Minimalist needs to take such functionality into account while generating the call-graph of the web application. I identified the set of higher-order PHP functions by manually analyzing the arguments and return values of the functions according to the PHP documentation (PHP, 2021). I then modeled their behavior according to the arguments accepted by each higher-order function and their return values in Minimalist. In the case of calling a higher-order PHP function, Minimalist infers the target function's name passed as arguments by leveraging the corresponding variable's `ValueSet`. Then, Minimalist adds a node to the call-graph for the caller and callee (if they do not already exist) and draws an edge between them. Note that, if there are multiple functions that match the passed argument (i.e., function to be invoked) to the higher-order function, Minimalist draws an edge between the caller and each of the matching functions. Similar to higher-order function invocation, I modeled the behavior of PHP's reflection API in Minimalist. Specifically, to address reflection, Minimalist extracts the argument that represents the function to be invoked and draws the respective edges in the call-graph. Analogous to variable function calls, Minimalist generates a report for unresolved instances of the invoked functions by higher-order functions or the reflection API, which should be addressed by an analyst through CSA. For script inclusion functions, Minimalist uses the script inclusion analysis result and creates a dummy node for the main body of the included script if it does not exist and draws an edge from the caller to the dummy node.

At the end of this step, Minimalist generated a call-graph for the web application using the information acquired from the previous analyses. My approach needs to construct the call-graph once per web application. Whenever there is a modification in the source-code of the target web application, such as upgrading to a new version or installing a new module, Minimalist needs to repeat the call-graph construction step, including the preliminary analyses.

```
function test() {
  //Retrieve the callable action from the database
  $query ="SELECT * FROM actions WHERE ".$conds;
  $result_db = mysql_query($query);

  //Assign the value to the variable action
  $action = mysql_fetch_row($result_db);
  // Invoke the retrieved function name
  // from the database
  $result = $action();
}
```

Listing 6.1: Drupal retrieves the name of the function to invoke from database. The function `test` is implemented in `actions.php`.

Custom Static Analysis

In this analysis, Minimalist resolves the problematic function calls and script inclusions into a small set of functions/scripts. For a small subset of function calls and file inclusions, Minimalist cannot statically resolve the callees or target scripts. In the absence of such information, Minimalist draws edges to every node in the call-graph or the script dependency graph. This level of abstraction can render the debloating process ineffective if an invoked function has edges to all the functions in the web application.

Since Minimalist is not able to fix the unresolved instances alone, alternative methods are necessary. My tool leverages an analysts' knowledge in order to resolve the missing function calls. Using the report from the previous step, a human analyst can inspect the source code of the web application and provide the annotations using the CSA API. Using these annotations, Minimalist can resolve the specific challenging call sites and file inclusions to a subset of functions and files.

To put this into perspective, I investigate an unresolved function call in Drupal 7.34 in Listing 6.1. Drupal registers a set of functions called "actions" in the database while getting installed or whenever there is a new module installed. Drupal retrieves the function names from the database to invoke under certain conditions, such as when a user comments on a post or replies to a comment. In Listing 6.1, Drupal issues a query to the database on line 4 to extract the name of the target function from the database and store it in the `action` variable on line 7. Given that the values fetched from dynamic queries executed on a database are not accessible to the static analysis, such cases pose a challenge to any static analysis tool, including Minimalist. In such a case, an analyst can assist Minimalist by providing the routine to query the database and retrieve all possible invoked target function calls and update the call-graph.

Listing 6.2 demonstrates the CSA for Drupal, which adds the edges in the call-graph for the function `test` based on the query results on the first line. Note that Minimalist needs

```
1  list_actions=db.Query('SELECT callbacks FROM actions')
2  foreach list_actions.Next() {
3    // grab items from the list of actions
4    var item
5    list_actions.Scan(&item)
6    // update the callgraph of function test
7    // with the retrieved action called item
8    update_callgraph("test", "actions.php", item)
9  }
```

Listing 6.2: The code snippet in the CSA to resolve the actions retrieved from the database in Drupal

to rebuild the call-graph whenever there is a change in the web application source-code (e.g., a new installed module). Considering that new installed modules in Drupal have their own actions in the database, communicating with the database allows the CSA to update the call-graph with the latest target function calls. The function `test` in Listing 6.1 only retrieves one action to invoke, which is determined by the provided conditions in variable `conds`. Since Minimalist cannot reason about the value of `conds` in Line 3 of Listing 6.1, the analyst needs to identify all possible invoked functions on Line 10. On lines 2 to 8 of Listing 6.2, I iterate over the values of the variable `action` retrieved from the database and add the target functions in the call-graph for the function `test` in `actions.php` inside Drupal.

6.1.2 Debloating the Web application

Up to this point, Minimalist generated the call-graph of the entire web application. In this step, Minimalist removes the pieces of code from the web application that are not necessary to respond to users' requests. Each individual request from the users of web applications invokes a small subset of the files within the whole codebase of the application. Moreover, not all functions contained within these files get invoked to respond to users' requests. The debloating process in Minimalist consists of identifying the reachable files and functions from the set of files accessed by users within the call-graph and then removing the unreachable parts of the graph.

First, I use access-log files to obtain the set of files that users access during their interaction with a web application. There are alternatives to this approach, including instrumenting the PHP interpreter and the web application to log every executed file, function, and line at runtime. This approach slows down the server's response-time by up to 17x in certain cases. Moreover, recording synthetic interaction with a web application for a short period of time does not encompass the behavior of real users' interactions. My approach infers the accessed entry points in the application by analyzing existing access-log files, which are readily available on the web servers. The web server records the requests that users and administrators send to the server for browsing the website, exercising the offered functionality, and debugging problems (Apache, 2021). Compared to instrumentation approaches, access-log files allow Minimalist to obtain real users' interaction over longer periods without causing additional performance overhead.

Second, for every file recorded in the access-log file, Minimalist identifies the node associated with the global context of the accessed file in the call-graph. Afterwards, Minimalist performs a reachability analysis to identify all the files and functions reachable from each accessed file. Minimalist repeats this process for all unique entries from the access-log file to build its overall reachable call-graph and prunes the nodes of unreachable files and functions.

In the last step, I debloat the web application at a function-level granularity based on prior users' interactions. Leveraging function-level debloating allows Minimalist to selectively remove functions and PHP files from the web application. To achieve this, Minimalist determines the set of line numbers associated with the body of reachable functions and the global scope of the scripts. Finally, it iterates over the PHP files in the web application and removes any lines that are not associated with the set of line numbers for the functions or scripts remaining in the pruned call-graph.

6.2 Evaluation

I assess the effectiveness of Minimalist from different perspectives on a set of popular PHP web applications. First, I assess the static analysis and its capability to resolve function calls in the web applications. Next, I analyze the CSAs implemented for the web application in our dataset. Finally, I evaluate the impact of debloating web applications in terms of reducing bloated code and removing security vulnerabilities. This evaluation aims to answer the following research questions:

RQ1. How precise is Minimalist in resolving function calls and generating the call-graph for a web application? (§ 6.2.2)

RQ2. How much effort do analysts need to implement a CSA for Minimalist? (§ 6.2.3)

RQ3. How effective is Minimalist in debloating web applications in terms of reducing the lines of code? (§ 6.2.4)

RQ4. What is the impact of Minimalist on removing severe security vulnerabilities? (§ 6.2.4)

RQ5. What is the effect of different debloating techniques on the usability of debloated web applications? (§ 6.2.5)

6.2.1 Evaluation Dataset

I evaluated Minimalist on four popular PHP web applications. The evaluation dataset includes three open-source PHP content management systems (CMS): WordPress, Joomla, and Drupal, and phpMyAdmin as a database administration tool. In practice, administrators customize CMSes by installing plugins. To reflect this, I installed the top five (at the time of writing) featured plugins (WordPress, 2022) on WordPress 4.6.0 in accordance to official WordPress website: Jetpack, Akismet, Health-check, classic editor, and classic widgets. According to W3Tech, these open-source CMSes account for 45.2% of all the websites on the Internet (Q-Success, 2023a). For each web application in the dataset, I selected the versions with the largest number of high-severity vulnerabilities based on the vulnerability CVSS score (NIST, 2021). Collectively, I analyzed 12 different versions (see Table 6.1) of the aforementioned web applications in the dataset and mapped 45 security vulnerabilities to their source code.

For the evaluation of Minimalist, I compared my tool with Less is More (LIM). LIM (Azad et al., 2019) is a dynamic debloating approach that records the executed lines of code in the web application while performing a series of interactions using Selenium scripts. Next, LIM removes the lines of code that were not exercised during the above interaction. I used LIM's source code, which is publicly available (Azad, 2022). In order to assess Minimalist using the analyst-provided CSAs, I implemented a custom static analysis for each web application in the dataset, which I describe in Section 6.2.3.

6.2.2 Static Analysis Evaluation

The static analysis in Minimalist is an integral part of my debloating scheme. This tool analyzes a PHP web application to generate a call-graph which is then used to debloat the given web application based on prior user interaction. The debloating performance of Minimalist is directly affected by the accuracy of its static analysis.

Table 6.1 presents the function call resolution statistics for the web applications in the dataset. The *Direct calls* column shows the total number of function calls that simply use the name of the function for invocation. The *Dynamic calls* column shows the number of function calls in a given web application that are not string literals. The *Resolved, Fuzzy-Resolved, and Unres Function calls* provide a breakdown of how the static analysis resolved each function call in a given web application. Namely, the *Resolved function call* column contains the number of function calls that are resolved to a single function definition. The *Unres function call* column presents the number of function calls that Minimalist cannot

		Static Analysis							Vulnerability Reduction		
		Function Calls						New LoC	Total CVEs	Total Removed CVEs	
Web app	Version	Total	Direct	Dynamic	Resolved	Fuzzy Resolved	Unres			LIM	Minimalist
WordPress	4 6 0	64,692	60,010	4,682 (7%)	63,719 (98%)	927 (1 4%)	46	768	2	0	0
	4 6 0 + Plugins	102,328	93,773	8,555 (8%)	100,416 (98%)	1,888 (0 33%)	24	123	-	-	-
	4 7 1	65,575	60,664	4,911 (7%)	64,631 (98%)	888 (1 3%)	56	37	2	1	0
	4 7 19	66,080	61,161	4,919 (7%)	65,157 (98%)	874 (1 3%)	49	0	-	-	-
	5 0	71,030	56,906	5,124 (7%)	70,055 (98%)	926 (1 3%)	49	10	-	-	-
PhpMyAdmin	4 0 0	26,077	23,424	2,653 (10%)	25,817 (99%)	253 (0 9%)	7	215	8	7	5
	4 4 0	29,232	25,009	4,223 (14%)	28,352 (97%)	874 (2 9%)	6	14	7	7	4
	4 6 0	44,415	34,503	9,912 (22%)	42,986 (97%)	1,421 (3 2%)	8	54	9	7	3
	4 7 0	46,119	34,792	11,327 (24%)	43,802 (95%)	2,271 (4 9%)	46	274	1	0	0
Drupal	6 15	14,298	14,101	197 (1%)	14,152 (99%)	90 (0 6%)	56	302	2	1	1
	7 34	29,833	23,434	6,399 (21%)	27,354 (92%)	2,435 (8 1%)	44	200	7	6	2
Joomla	3 4 2	89,087	59,834	29,253 (32%)	79,389 (89%)	9,677 (10 8%)	21	680	3	1	0
	3 7 0	101,477	67,673	33,804 (33%)	88,608 (87%)	12,850 (12 6%)	19	167	4	3	2
Average (w r t	Total Calls)	100%	83 21%	16 79%	95 23%	**4.72%**	**0.05%**				
Total									45	33 (-73%)	17 (-38%)

Table 6.1: A break down of the static and dynamic function calls for each web application in our dataset. I also include the result of function call resolution precisely and approximately for each web application. The last two columns in the *Static Analysis* section present the number of unresolved function calls in each web application and the number of new implemented lines in their CSAs. The *Vulnerability Reduction* section presents the number of removed security vulnerabilities from the web application debloated by LIM, and Minimalist.

resolve to a subset of defined functions in the web application. Finally, the *Fuzzy-Resolved function call* column shows the number of function calls that Minimalist resolves to a subset of defined functions which is less than total number of functions in the web application.

The static analysis in Minimalist resolved 99.95% of all function calls in the web application to a single function (95.23%) or a subset of defined functions (4.72%) in each web application in the dataset. To handle unresolved function calls, Minimalist requires an analyst to provide the CSA annotations. For the evaluation, I implemented the CSA for all the web applications in our dataset. The last column in Table 6.1 present the manual effort required to implement a CSA in terms implemented lines of code (LoC) per version.

In a further analysis of Minimalist's call-graph generation, I assessed the number of resolved higher-order functions. Higher-order functions in PHP take a target function name as an argument, which gets invoked by the interpreter. Such behavior poses a challenge for any static analysis, including Minimalist. Thus, I investigated all 4,143 invocations of higher-order PHP functions in the dataset of web applications and counted the number of resolved higher-order functions. Minimalist resolved 99.92% of all higher-order functions. To handle the remaining 0.08%, Minimalist relies on the implemented CSAs for the web applications.

6.2.3 Custom Static Analyses

In this section, I quantify the effort required by an analyst to implement a CSA for a given web application and maintain it over time across multiple web application updates and new releases. As described in Section 7.2.1, I implemented a CSA for each web application in the dataset to handle instances of unresolved call-sites. First, I look into the development of CSAs for different versions of web applications and the reusability of previous CSAs when migrating them to a new version of the same web application. Next, I investigate the major version changes in web applications and the underlying changes that affect the CSA implementation. Finally, I examine the use of third-party libraries in different web applications and their effects on implementing CSAs.

Custom Static Analysis. The data in Table 6.1 indicate that the manual effort required to implement a CSA varies between versions of a web application. We see in the last column of *Static Analysis* section in Table 6.1 that the first version of a CSA often requires the largest number of new implemented lines. This is because the first version needs to implement annotations for all the function calls that Minimalist does not resolve. The analysis of the unresolved function calls shows that the majority of the unresolved functions remain unchanged across versions. In such cases where there is little to no change in the unresolved dynamic function calls, analysts can reuse the same CSA annotations from the previous versions of the web application with zero to minimal change.

Figure 6·5 plots the number of new lines of code (y-axis) implemented over time for multiple versions of phpMyAdmin and WordPress. According to Figure 6·5, the first implemented CSA for WordPress requires the highest number of implemented lines, which then drastically reduces for the next versions of WordPress. During the evaluation, I observed that on average, 80% of the code in the CSA remains unchanged between two consecutive versions of the same web application. In the case of WordPress, from 2016 to 2020, analysts only need to add or modify an average of 10 lines of code each year. For instance, although WordPress 4.7.19 was released *three years* after 4.7.1, an administrator can fully reuse the CSA on version 4.7.1 with zero modifications.

Major releases and architectural changes. Major changes in the architecture of web applications can affect the reusability of CSAs. In the dataset, phpMyAdmin from version 4.7.0 started incorporating the Composer package manager and its provided third-party libraries. This resulted in a 45% increase in Logical Lines of Code (LLOC) between versions 4.6.0 to 4.7.0 in this application and 41 unresolved function calls that need to be included in the CSA. This increase in the number of unresolved function calls is evident by the increase in the required number of new implemented lines of code in the CSA in Figure 6·5.

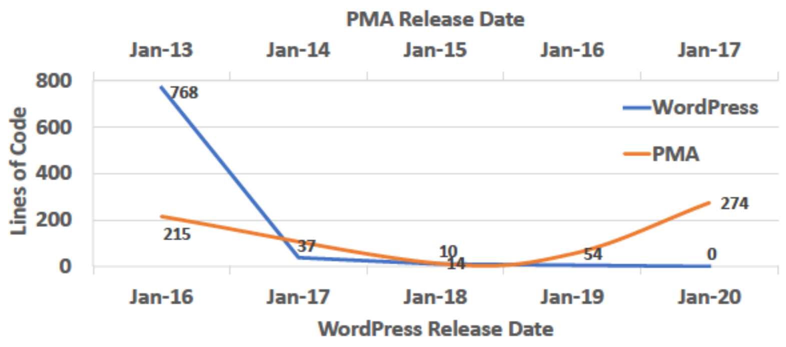

Figure 6·5: The number of new lines of code implemented in CSAs for various versions of WordPress and PMA over time.

Reusability of CSAs for third-party libraries across web applications. Web applications rely on third-party libraries to provide common functionality, such as sending emails (e.g., `PHPMailer`). WordPress and Joomla in the dataset both use `PHPMailer` in their source-code, which allows Minimalist to reuse the CSA for unresolved function calls of `PHPMailer` between WordPress and Joomla. Although WordPress and Joomla use different versions of `PHPMailer`, the list of unresolved function calls remains unchanged. This enables analysts and developers to provide the CSA for popular libraries, which can then be shared and reused to debloat a wide range of web applications and their third-party libraries.

Cross-validation of CSA: While plugins bring new features to web applications, they also introduce unresolved function calls to the analysis, which an analyst needs to address via CSA. In the evaluation of WordPress' plugins, Minimalist required the analyst to create CSAs that resolve 24 unbound function call-sites. Note that not all plugins introduce unresolved function calls. In the dataset, only two out of five WordPress plugins did: Jetpack and Health-check. During this evaluation, I measured the time it takes for two different analysts to resolve each instance in the CSA. To achieve this, two authors independently implemented the CSA for the 24 instances of unresolved calls in the WordPress plugins while recording the time required to address each instance.

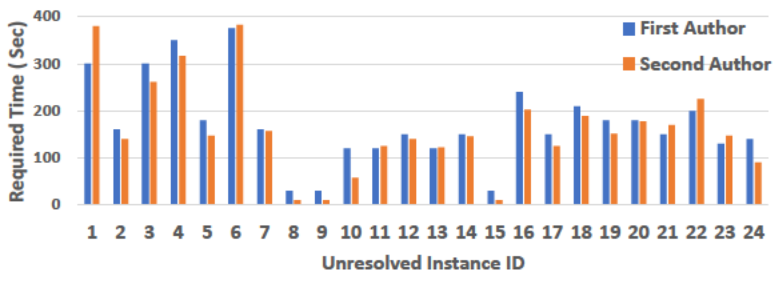

Figure 6·6: Time distribution of implementing a CSA for each unresolved instance in WordPress plugins. On average, each author spent less than 3 minutes to resolve each instance.

Figure 6·6 shows the distribution of times it took both authors to implement the CSA for each unresolved instance. The experiment shows that the time needed for each unresolved instance varies depending on the complexity of the code. However, there are instances (e.g., instances 8, 9, and 15 in Figure 6·6) that take less than 30 seconds to resolve. The reason behind such short analysis times is the similarity of the instance with previously handled cases, which reduces the time of analysis and implementation. Overall, author A and B spent 75 and 65 minutes implementing the CSA for WordPress plugins, respectively.

In the examination of the CSAs, I also investigated the differences in the authors' implementation of the CSAs. Since different analysts can create different CSAs for the same unresolved instance, I assessed whether such differences affect the overall accuracy of Minimalist. For this evaluation, I inspected the implemented CSAs by the two authors. The analysis shows that, while each author follows different coding practices, the differences in the implemented CSAs do not lead to any discrepancies in the generated call-graph or later on in the debloating process.

Overall, it took less than 20 person-hours for the authors to implement the first version of a CSA for each web application in the dataset. This process includes inspecting the source-code of the the unresolved instances listed in Table 6.1 and writing the CSA plug-ins. The reusability of CSAs among different web applications and versions of the same web application amortizes the effort of implementing one for newer web applications. Furthermore, crowd-sourcing the tasks in the implementation of CSAs among developers and administrators of web applications (CSAs are *globally* valid) can further minimize the effort of authoring CSAs.

6.2.4 Debloating results of Minimalist

In this section, I evaluate the effectiveness of the debloating scheme by measuring the reduction in lines of code and security vulnerabilities after debloating. Minimalist reuses the same usage profiles as LIM to generate the entry-point information and feature usage. I collected the access-log files for each web application in the dataset using the Selenium scripts available on LIM's website and exercised the web applications. For Drupal and Joomla, I adopted the same approach as LIM, produced the Selenium scripts based on online tutorials, and collected access-logs to get the ground truth of coverage information from the LIM framework.

LLOC Reduction

According to McConnel (McConnell, 2004), the number of programming errors in an application is proportional to the size of the program. Given the correlation between the size of an application and its overall security, I look into the reduction of web applications' size in terms of LLOC. LLOC represents the number of lines in the source code, excluding comments and empty lines.

Figure 6·7 demonstrates the LLOC reduction for different versions of web applications that Minimalist debloated. On average, Minimalist debloated 17.78% of LLOC in all the web applications in our dataset while using the implemented CSAs. LIM debloats 53.47% of the web applications in the dataset. As discussed before, LIM is a dynamic debloating mechanism that removes all functions and scripts that are not exercised during its training with Selenium scripts. As seen in Figure 6·7, relying on dynamic traces for debloating

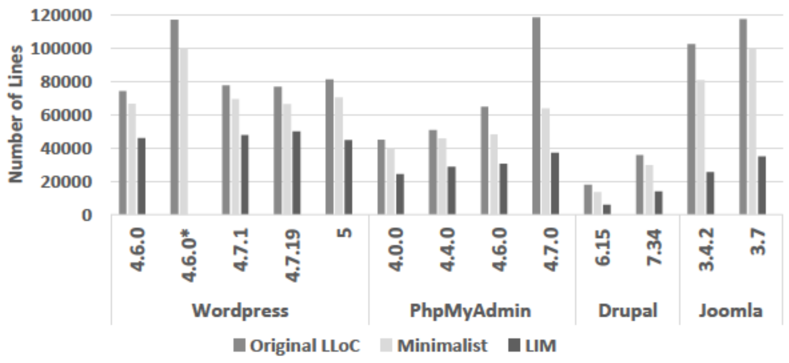

Figure 6·7: LLoC reduction of Minimalist. The 4.6.0*, presents the LLoC reduction in WordPress and its five featured plugins.

leads to more removed lines in debloated web applications. At the same time, any slight variation in user interactions from the dynamic training data with the web application can lead to breakage. I discuss this issue in Section 6.2.5 in more detail.

According to Figure 6·7, I observe a sudden expansion in phpMyAdmin 4.7 compared to its previous versions. As noted by Amin Azad et al. (Azad et al., 2019), the sudden expansion of the source code of phpMyAdmin 4.7 is due to changes in development practices. Namely, phpMyAdmin 4.7.0 started relying on external libraries, which introduced a large amount of unused code and increased the size of phpMyAdmin 4.7.0 by 45% compared to phpMyAdmin 4.6.0. I observed that Minimalist removes 62% of the lines in the external libraries that reside in the `vendor` directory of phpMyAdmin 4.7.0.

Security Vulnerability Reduction

In addition to LLoC reduction, I evaluated Minimalist's security benefits by analyzing its debloating effect on security vulnerabilities. Minimalist removes a vulnerability if it resides in an unreachable function with respect to the users' interaction with the web application.

In the *Vulnerability Reduction* section of Table 6.1, we compare the number of removed vulnerabilities after debloating each web application in the dataset using Minimalist and LIM. In Table 6.1, I present the total number of security vulnerabilities in the dataset for each web application. The last two columns present the number of removed vulnerabilities in our work and LIM. On average, my debloating scheme can remove 38% of vulnerabilities in web applications, while LIM removes 73% of the vulnerabilities. The analysis of the removed vulnerabilities by LIM that Minimalist preserved shows that all the vulnerable functions are reachable from the entry-points in the analyzed access-log files. Thus, Minimalist does not remove the vulnerable functions to preserve the functions required by the users in the debloated web application.

Compared to Minimalist, LIM favors a more aggressive approach on debloating web applications and consequently removing vulnerabilities. A case study for this argument is the CVE-2016-6609 vulnerability in phpMyAdmin 4.4.0 and 4.6.0. This vulnerability resides in an export module, where an attacker can run arbitrary PHP commands using a specially crafted database name. The excessive debloating of LIM removes the security vulnerability as well as all but one of the exporting functions from phpMyAdmin. Compared to LIM, Minimalist preserves all exporting functions, thereby retaining the vulnerable code but also all the export features that users might require. This demonstrates the clear dichotomy between the dynamic, LIM-like approaches that favor aggressive debloating gains (accepting breakage while doing so) vs. Minimalist that aims to provide a balance between debloating-based security gains and preserving the functionality and usability of the debloated software.

Overall, in the debloating experiments, I demonstrated the reduction of LLOC in web applications and its effect on eliminating vulnerabilities. Compared to prior work, I observed higher debloating numbers in LIM. LIM was built as a means to quantify the benefits of debloating and its potential to remove security vulnerabilities, assuming the system is provided with complete dynamic traces. In contrast, Minimalist is a practical debloating scheme that provides a balance between debloating source-code, removing vulnerabilities, and keeping debloated web applications usable.

6.2.5 Robustness of Debloated Web applications

In this experiment, I evaluated the robustness against false positives of web applications debloated by Minimalist. False positives (i.e., breakage) in a debloated web application occur when a user's interaction causes the invocation of an (incorrectly) removed function. To this end, I used two different approaches to investigate the occurrence of false positives in debloated web applications: 1) Automatic random testing and 2) Official testsuites.

Automatic Random Testing

In this experiment, I evaluated the robustness of debloated web applications using the crawling feature of Burp-suite to mimic random user behavior. I argue that there should not be any false positives in the debloated web applications as long as Burp-suite targets the already visited PHP scripts by Selenium. Note that I debloated the web application based on the prior user interaction recorded in the LIM's Selenium. To assess robustness, I crawled the debloated web applications using Burp-suite with a custom-defined scope. This scope forces Burp-suite to only crawl a predefined set of PHP scripts in the debloated web application, which, in this case, are the PHP scripts visited by Selenium. While Burp-suite will target the same web application entry points, it will randomly vary the passed parameters and values leading to execution paths that differ from those observed during the Selenium interactions. Whenever Burp-suite invokes a removed PHP function, the debloated web application raises an alert. Thus, I calculated the number of alerts raised by Burp-suite to examine the robustness of the debloated web application. In the experiment, I crawled the debloated WordPress and phpMyAdmin for one hour each using Burp-suite. Collectively, Burp-suite sent 1,055 requests to both debloated WordPress (603) and phpMyAdmin (452) and raised no false positives.

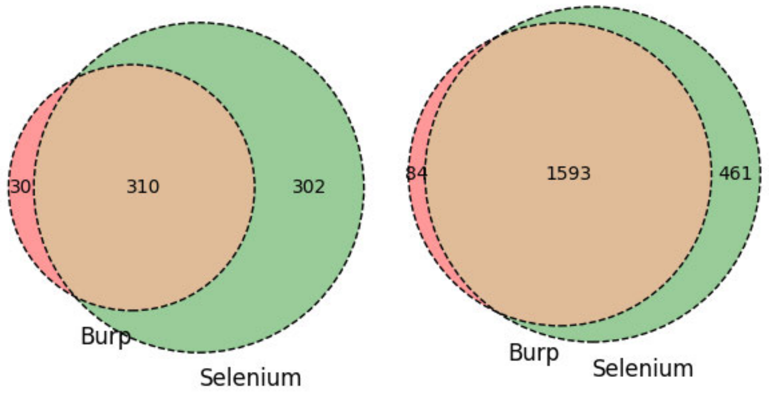

Figure 6·8: The function coverage of Burp random testing compared to Selenium browsing for debloated PhpMyAdmin (left) and WordPress (right).

In the next step of the analysis, I looked into the function coverage of the web applications while browsing with Burp-suite compared to Selenium scripts. Note that the Burp-suite browsing tests are only meaningful if they cover a different set of functions in the web applications during their browsing compared to the Selenium scripts. Thus, I recorded the set of invoked functions during both Burp-suite and Selenium browsing. Figure 6·8 shows the set of different invoked functions during both browsing patterns. Burp-suite browsing led to the invocation of 114 (7.5%) functions that were not covered during Selenium browsing. Importantly, I note that the invocation of 114 new functions by Burp-suite would yield up to 114 false positives in a dynamic debloating approach such as LIM. However, Minimalist correctly debloated the web applications using the access-log files and preserved the necessary functionality to respond to users' requests.

Official Testsuites

In a further experiment, I evaluated the breakage of debloated applications by using the official testsuites obtained from their respective Github repositories. In order to execute the official testsuite of the web applications, we manually prepared the testing environment, which included creating configuration files, database tables, and inserting sample data to the database. For this evaluation, I executed all 7,238 test cases from the official testsuites of both phpMyAdmin and WordPress on Minimalist-debloated web applications.

Table 6.2 presents the results of this experiment. Each set of tests from the official testsuites belongs to a category, which is shown in the second column of Table 6.2. The next two columns present the total number of tests in each category and the number of failed tests. On average, the debloated web applications in the dataset failed 12% of the official testsuite (885 out of 7,238 total unit-tests). During the experiment, I randomly chose 6% of failed test cases (52 out of 885 total failed unit-tests) and investigated the cause of failure. All 52 failures were rooted in a few correctly debloated functions. The last column in Table 6.2 shows the name of the debloated function that failed the test cases in each group. Note that, these failed test cases are not false positives of Minimalist. Specifically, the analysis of both web applications reveals that neither of these functions are reachable from the PHP files in the access-log, and were either deprecated (e.g., `wp_shrink_dimensions`) or exclusively invoked from entry points not found in the access-log. Hence, Minimalist correctly debloated the functions.

In a further evaluation, I examined the set of features that Minimalist preserves in the web application but LIM removes. During this experiment, I observed that unlike Minimalist, LIM causes up to 33% false positives in new real-user browsing patterns on average. Overall, in the evaluation, I performed several experiments on web applications debloated by Minimalist and the state-of-the-art approach, LIM (Azad et al., 2019). I evaluated Minimalist and LIM in terms of reducing the LLoC of web applications and its effect on removing vulnerabilities. I observe that although dynamic debloating techniques such as LIM have higher debloating numbers compared to Minimalist, their debloating approach causes false positives in debloated web applications.

Web app	Test Group	Tests		Example Reason
		Total	Failed	
WordPress	Admin	741	22	`get_help_tab(2)`, `comment_exists(5)`
	Authentication	16	0	
	Comment	311	21	`unreigster_taxonomy(6)`
	File Operation	20	0	
	Others	5152	709	`parseISO(6)`, `getISO(4)`, `wp_shrink_dimensions(4)`, `is_comment_feed(5)`, `remove_permastruct (8)`
Total		6240	752 (12%)	
phpMyAdmin	Unit	509	39	`npgettext(2)`, `StringReader::currentpos(3)`
	Engines	26	0	
	Classes	463	93	`HasErrors(1)`, `HasUserErrors(1)`, `getVersion(3)`, `getPrintPreview(1)`, `locale_emulation(1)`
Total		998	132 (13%)	

Table 6.2: On average, Minimalist-debloated web applications fail 12% of official testsuite. The last column presents the name of functions and the number of failed tests due to debloating each function in paranthesis.

6.3 Discussion and Limitations

In this section, I discuss the limitations of Minimalist. Of particular interest are the code practices that challenges Minimalist to generate a call-graph, and the fact that manually-created CSAs might introduce unsoundness into the generated call-graph. Furthermore, I elaborate on extending Minimalist to debloat web applications using other languages such as JavaScript.

Soundiness in CSAs: Minimalist resolved the majority of function calls (99.95%) in the web applications in our dataset. The remaining 0.05% of call-sites required the development of CSAs, inducing small amounts of developer attention (even zero in the case of WordPress 4.7.19). Obviously, unsound CSAs can render the resulting call-graph unsound too. As Minimalist cannot assess whether a CSA is sound, it is the developer's responsibility to ensure the developed CSA preserves soundiness. CSAs are necessary in scenarios where Minimalist cannot reason about specific program constructs (e.g., custom call-back schemes), or where control flow is determined based on factors external to the web application's code (e.g., by information stored in a database). In theory, these challenges can become arbitrarily complex. In practice, I observed that during the development of *all* the CSAs used in this work, soundiness can be manually ascertained. Based on the observation that our evaluation covers the largest and most popular web applications in use today, we are confident that CSAs for other web applications can be created in a soundy manner too.

Unsupported PHP features in Minimalist: Minimalist's static analysis is soundy with respect to most features of the PHP interpreter. models most features in the PHP interpreter to generate call-graphs. However, there are features in the PHP interpreter that challenge any static analysis, including Minimalist. Among the PHP features, there are two that Minimalist does not support in its current implementation: 1) dynamically loaded code through `eval` and `assert` and 2) arguments passed by reference. `eval` and `assert` evaluate their string arguments as PHP code, which can originate from arbitrary origins (e.g., a remote URL) or computation (e.g., the decryption of encrypted content). Such functionality is widely recognized to be beyond the reach and capability of static analysis techniques.

Besides that, there exists a set of PHP features that Minimalist partially supports, which includes, 1) dynamic file inclusion, 2) reflection API, 3) higher-order functions, and 4) variable function calls. All the above features use variables to either include a dynamic script or invoke a function that is determined at runtime. Thus, resolving the variables is an essential step to identifying the invoked function. Minimalist over-approximates the value of variables used in dynamic function calls. Thus, in cases where the system cannot constrain the value of variables, it draws edges to all defined functions in the web application. However, such aggressive over-approximation limits the utility for debloating purposes, and hence Minimalist calls the analyst's attention to these instances, which have to be resolved via CSA. Table 6.3 includes the full list of features that Minimalist partially supports or does not support. I identified the features listed in Table 6.3 by relying on prior work such as

Pixy (Jovanovic et al., 2006), RIPS (Dahse and Holz, 2014a), and Hills et al. (Hills et al., 2013), as well as our expertise on analyzing PHP applications. Note that I cannot guarantee the completeness of the features listed in Table 6.3 due to the complexity of the PHP interpreter as well as its large codebase (1.3M LOC).

Type	**Function name**
	Partially Supported Features
Higher-order function	*call_user_func, call_user_func_array, array_map, preg_replace_callback, array_walk, array_walk_recursive, array_reduce, array_intersect_ukey, array_uintersect, array_uintersect_assoc, array_intersect_uassoc, array_uintersect_uassoc, array_diff, array_diff_ukey, array_udiff_assoc, array_diff_uassoc, array_udiff_uassoc, array_filter, array_udiff, usort, uasort, uksort, ob_start, session_set_save_handler, assert_option, sqlite_create_function, register_shutdown_function, register_tick_function, set_error_handler, set_exception_handler, iterator_apply, spl_autoload_register*
Reflection API	*ReflectionClass, ReflectionMethod, ReflectionFunction*
Dynamic file inclusion	use of variables in script inclusion functions
Variable function call	use of variables for invoking a function
	Unsupported Features
dynamic loaded code	`eval`, `assert`
Pass by reference	

Table 6.3: The list of dynamic PHP features that Minimalist partially supports or does not support while generating call-graph.

Extend Minimalist to Other Languages: In the current implementation of Minimalist, I focus on PHP web applications, which power more than 77% of all live web sites (Q-Success, 2023b). While each programming language has unique characteristics, there are similarities between PHP and other server-side languages such as JavaScript or Python. For example, both JavaScript and Python support variable function calls in scripts, which is similar to PHP. Furthermore, both Python and JavaScript also allows the dynamic inclusion of modules, which is similar to `include` in PHP. These similarities suggest that our approach of handling dynamic features in the PHP interpreter is applicable to other interpreted applications such as JavaScript and Python. Of course, the technical details and idiosyncrasies of other languages would still require significant engineering efforts. However, not all programming languages provide such a diverse set of dynamic features. For example, Java only provides a fraction of the dynamic features (e.g., the reflection API) that are available in PHP. As a result, the challenges of analyzing dynamic features to debloat Java applications might be fewer than those of interpreted languages such as PHP.

Chapter 7

Detection of Vulnerability Injection Sinks in PHP runtime

In this chapter, I describe my fourth contribution to improve the security of PHP web applications by detecting the set of vulnerability injection sinks in the PHP interpreter. Injection sinks are a set of PHP APIs which can cause an injection vulnerability in a PHP application, if used with improper user-input sanitization. First, I elaborate the implementation of my approach on the PHP interpreter called Argus in Section 7.1. Next, I evaluate Argus on three most popular versions of the PHP interpreter, and explain how Argus' results improve existing detection tools to identify XSS and deserialization vulnerabilities in PHP web applications in Section 7.2.

7.1 System Design

In this section, I discuss the salient characteristics of my approach – Argus – and how it identifies the set of PHP APIs that leads to deserialization or writing user-input to output buffer. For consistency, I use the term *output* APIs to refer to the set of APIs that writes to output buffer, which leads to an XSS vulnerability. Similarly, I refer to set of APIs that invoke the deserialization in the PHP interpreter, as *deserialization* APIs. Figure 7·1 illustrates the overall process. First, Argus combines static and dynamic analysis techniques to generate a call-graph of a PHP interpreter in Step ①. Subsequently, for Step ②, Argus uses the call-graph to perform a reachability analysis to determine the set of API functions that invoke the *vulnerability indicator functions* (VIFs). Furthermore, Step ③ discusses

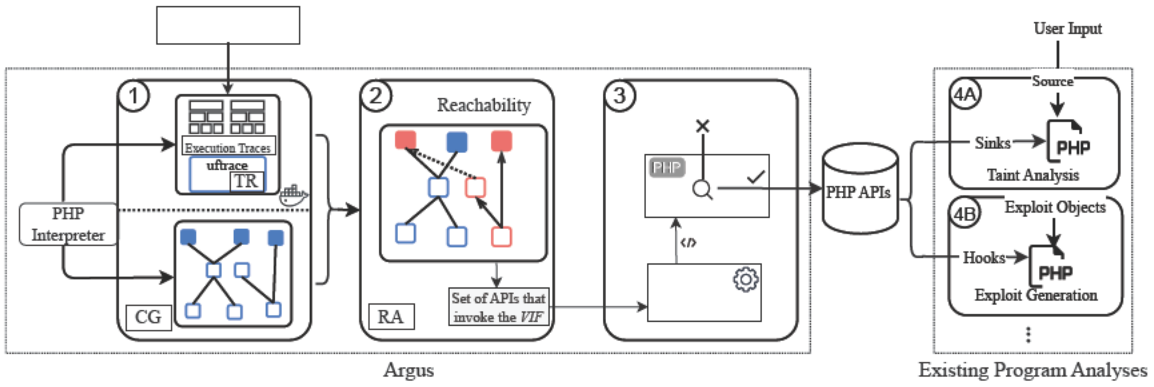

Figure 7·1: Argus performs a hybrid static-dynamic analysis on the PHP interpreter to generate a call-graph. Next, Argus identifies a comprehensive set of output and deserialization APIs through reachability analysis and validation tests. The output of Argus can be used to improve the existing program analysis tools to identify POI and XSS vulnerabilities.

the validation mechanism in Argus to confirm the injection-sinks. Finally, I discuss incorporation of Argus' results into existing program analyses to detect and exploit previously unknown vulnerabilities.

The implementation of the call-graph analysis for the PHP interpreter consists of approximately 700 LoC of Python and C code. In addition, I implemented my extension of two existing program analyses (i.e., Psalm and FUGIO) with less than 600 LoC of PHP.

7.1.1 Call-graph Generation

In Step ①, Argus generates a call-graph for the PHP interpreter. To achieve this, Argus performs a static analysis on the PHP interpreter to generate the initial call-graph, which it then refines using dynamic execution traces.

Static Analysis of the PHP Interpreter

To construct the call-graph, Argus analyzes the PHP interpreter. The PHP interpreter's core consists of approximately 120K lines of C code. Additionally, the interpreter relies on extensions to deliver features such as image processing, database communication, and communication protocols such as LDAP and IMAP. These extensions are free to augment the PHP API, including adding additional injection-sinks, and frequently do so. For example, in a standard PHP deployment, the *GD* graphics library, the *PDO* database communication extension, and the FTP extension are provided as separate libraries and all add additional injection-sinks to the runtime. To complicate matters further, extensions can be written in different programming languages (e.g., (copernica, 2022; Zephir, 2022)), provide their own build environments, and are usually simply loaded by the interpreter as shared dynamic libraries. However, injection vulnerabilities can arise from any API provided by the runtime, including APIs provided by the core interpreter and those provided by extensions. As such, it is imperative to analyze the interpreter's core along with the code that comprises the extensions. At first glance, the open source nature of the PHP interpreter would suggest a source-based analysis to infer the call-graph. However, the variety of frameworks, languages, and build systems used for extensions would require an analysis catering to all these characteristics. Thus, instead of deriving call-graph information from various interconnected source-based analyses, Argus instead performs its call-graph analysis on the compiled binaries of the interpreter and its extensions. To facilitate this analysis, I build the runtime and include debug symbols.

The call-graph analysis ($\boxed{\texttt{CG}}$ in Figure 7·1) first disassembles the PHP interpreter and all of its shared libraries using the `objdump` tool. Argus builds the call-graph by adding a node for each binary symbol in the disassembled PHP interpreter. Subsequently, Argus performs a linear scan over the interpreter and library disassembly. For every call instruction, $\boxed{\texttt{CG}}$ draws an edge in the call-graph from the caller (i.e., the currently analyzed symbol) to the callee (i.e., the target of the call). This analysis works well for direct calls and calls to symbols provided by extensions. That is, direct calls will invoke symbols that have corresponding names in the debug information. Argus handles calls to imported symbols by launching the PHP interpreter with the `LD_DEBUG=binding` environment variable set to infer symbol binding information from extension libraries. The `LD_DEBUG` option allows Argus to resolve the external symbols to the library and address where the symbols are implemented. Unfortunately, indirect calls (e.g., those that are used to implement the concept of stream wrappers) elide this analysis.

Refining Call-graph using Dynamic Analysis

Argus uses dynamic analysis to handle indirect calls in the PHP interpreter and refine the statically generated call-graph created in the previous step. For instance, PHP's `fopen` can be used to access local or remote files over protocols such as HTTP, HTTPS, or FTP. Depending on the argument passed to `fopen`, the PHP interpreter decides which stream wrapper (see Section 2.3.5) should handle the underlying resource. Internally, PHP stream wrappers rely on function pointers to dispatch operations (e.g., `fread()`) to functions that handle the protocol corresponding to the opened resource. The static analysis in Argus cannot handle such cases implemented in the PHP interpreter and runtime. To address this issue, Argus improves the statically generated call-graph by tracing the execution of the PHP interpreter while executing its high-quality test-suite (i.e., the PHP test-suite achieves a remarkable 73% line coverage). Argus then uses this dynamic information and adds any edges not already detected by the static analysis to the call-graph.

To achieve this, I compile the PHP interpreter with the `-pg` flag. This flag instruments each function with two additional hook functions at the entry and exit of each function, which allow Argus to perform dynamic tracing (GCC, 2022). The first function call occurs just after each function entry, which invokes the function `__cyg_profile_func_enter`. The next function which is `__cyg_profile_func_exit`, get invoked before exiting each function. After the recompilation of the PHP interpeter, Argus uses the `uftrace` tool (Kim, 2022) ($\boxed{\texttt{TR}}$ in Figure 7·1) to implement both hook functions and record dynamic traces. Finally, Argus executes the PHP unit tests while `uftrace` records the execution traces for each test-case.

After recording the execution traces, Argus iterates over the sequence of invoked functions by each test-case and examines the statically generated call-graph for the missing edges. For every invoked function during the dynamic analysis, Argus draws an edge between the pair of functions in the execution trace if there is no edge representing the recorded invocation.

At the end of this step, Argus has assembled a static call-graph of the PHP interpreter and refined it using dynamically-collected traces of PHP unit tests.

7.1.2 Reachability Analysis

In Step ②, Argus performs a reachability analysis on the generated call-graph, which requires the identification of sources and sinks on the call-graph. The key observation of analyzing the PHP interpreter demonstrates that, the PHP interpreter uses a single internal function that is responsible for all deserialization operations, called `php_var_unserialize` (VIF). The PHP interpreter uses a customized parser to parse serialized strings, which are then converted to PHP objects. The analysis of this custom parser across PHP's source-code yielded a single deserialization function inside the PHP interpreter. In the case of output APIs that write to an output buffer, I observed a similar pattern during my analysis of PHP's source-code, where the function `php_output_write` is responsible for outputting the buffer. With the VIF identified as sinks, Argus labels all API functions in the call-graph as sources.

Unfortunately, the symbols for API functions are indistinguishable from those of internal (i.e., non API) functions, and text-based techniques that parse documentation are rarely, if ever, accurate. However, a running PHP process must be aware of any and all APIs exposed to the web applications running on top of it. Thus, to identify the set of APIs, Argus uses a PHP extension that, once loaded into the PHP interpreter, iterates over all available APIs. Specifically, the extension first invokes PHP's `get_defined_functions` API to obtain the list of all API functions. Unfortunately, the results of `get_defined_functions` cannot be directly mapped to the call-graph. The reason is that the name of an API function available to a web application is commonly different from the name of the symbol that implements the actual functionality. For example, the `session_decode` PHP API is implemented by a function called `zif_session_decode`. Unfortunately, the `zif` prefix is not a consistent pattern. However, as the nodes in the call-graph correspond to symbol names rather than API names, the API names have to be translated. To this end, the extension leverages an interpreter-internal data structure (i.e., `executor_globals.function_table`), which maps API names to the names of the functions that implement the APIs' functionalities. Finally, the extension relays this information to Argus, which labels the symbols that map to API functions in the call-graph accordingly.

Once all APIs are labeled as sources, Argus traverses the call-graph for each source node and follows any call edges captured in the graph. Argus identifies an API as a deserialization or output API if this traversal includes the graph node corresponding to its VIF function.

7.1.3 Validation

The reachability analysis presented above might inappropriately label an API as an injection-sink if the underlying implementation in the runtime performs input sanitization or filtering. Thus, Step ③ filters APIs and only passes those that propagate their input VIF unmodified. To this end, Argus automatically generates PHP snippets to test each identified API for this characteristic. More specifically, these snippets contain a class definition (i.e., `test`) that, if deserialized (i.e., its `__wakeup` method is invoked), prints the content of one of its properties (i.e., `msg`) as a success message. Subsequently, the template calls the API in question with a serialized `test` object that has `msg` set to "SUCCESS". Thus, if the execution of the PHP snippet prints the success message, Argus validates that the API in question passes the input argument unmodified to VIF, and hence the API is for sure a deserialization API.

As the evaluation in Section 7.2 will show, Argus confirmed 284 deserialization APIs in the PHP interpreter, which warranted an automated validation step. However, the number of confirmed output APIs in the PHP interpreter is 22, which prompted us to manually validate whether the invocation of the API with user-input can cause an XSS attack. To this end, I created a Docker container with a running Nginx web-server for each PHP interpreter version. For each output API, I created a PHP template that invokes the API and passes a constant user-input containing a Javascript snippet (i.e., `<script>alert(1)</script>`). I visited the generated PHP template on the client side's browser, and if the browser displays a dialog, Argus marks the tested API as an output API.

7.1.4 Extend program analyses

Argus' results comprise a comprehensive list of injection-sinks. This is in contrast with the exclusively manually-crafted lists of injection-sink used by all existing XSS and POI detection and automatic exploit generation systems. Thus, to demonstrate the value of Argus' principled approach, I extend two existing program analysis tools, Psalm (Vimeo, 2021) and FUGIO (Park et al., 2022) as examples of downstream analysis that benefit from my work.

Psalm Extension

Psalm is a static analysis tool featuring code refactoring and taint analysis (Vimeo, 2021). For its taint analysis, Psalm attempts to find unwanted flows between user-controlled inputs (e.g., $GET variables) and a set of sink functions (e.g., `system`). The set of sink functions in Psalm differs depending on the type of vulnerability that the user is trying to detect. For instance, to identify insecure deserialization, Psalm exclusively considers `unserialize` as a sink.

As shown in previous reports (Thomas, 2018) and in the evaluation, this assumption results in false negatives (i.e., missed detections). To extend Psalm's static analysis, I modify the list of taint sinks to include all deserialization and output APIs identified by Argus. In Listing 7.1, I show the generated annotation for adding the `md5_file` PHP API function as a taint sink to Psalm's configuration. Lines 2-4 in Listing 7.1 define the PHP function and the type of arguments it accepts. Furthermore, Line 1 in the listing annotates that the argument `file_name` to the function `md5_file` is a taint sink for a deserialization vulnerability.

```
1 // @psalm-taint-sink unserialize file_name
2 function md5_file(string $file_name, bool $binary=false)
3 { }
```

Listing 7.1: We extended Psalm static taint analysis by modifying its list of taint sinks for POI and XSS vulnerabilities.

FUGIO Extension

FUGIO is an automatic exploit generation tool which uses a combination of static and dynamic analysis to detect and generate a proof of concept exploit for POI vulnerabilities. In the first step, FUGIO submits requests to a target web application where request parameters (e.g., GET, POST, and COOKIE values) contain serialized data. During the processing of these requests, FUGIO hooks the invocation of deserialization APIs and verifies if the passed arguments correspond to the parameters supplied in the request. To this end, FUGIO hooks a subset of 27 PHP deserialization APIs – the explicit `unserialize` API along with 26 implicit APIs first mentioned by Thomas (Thomas, 2018). If FUGIO detects that parameters are indeed forwarded to deserialization APIs, its second step will attempt to morph the parameter into a complete POP chain, forming a POC exploit. While FUGIO's second step (i.e, the exploit generation itself), is independent of my work, the first step (i.e., recognizing the invocation of vulnerable deserialization APIs) is directly affected by the (in-)completeness of the list of deserialization APIs.

To extend FUGIO, I integrated the set of deserialization APIs identified by Argus such that FUGIO hooks all these APIs in its first analysis step. Listing 7.2 shows how I added a hook for the `get_meta_tags` PHP deserialization API which was among the dozens not considered by FUGIO to begin with. The extended FUGIO intercepts a more comprehensive set of PHP APIs which allows it to identify and exploit previously unknown POI vulnerabilities (see Section 7.2.3 for details).

```
$hook_get_meta_tags_func = function()
{
  global $argv_list_r353t;
  saveDatas_r353t($argv_list_r353t,
        'get_meta_tags', func_get_args(), '1');
};
uopz_set_hook('get_meta_tags',
        $hook_get_meta_tags_func);
```

Listing 7.2: We extended FUGIO by modifying its detection mechanism in order to intercept and hook into the list of functions reported from Minimalist.

In summary, Argus generates a call-graph for the PHP interpreter by leveraging hybrid static-dynamic analysis. Furthermore, Argus performs a reachability analysis to identify a comprehensive set of deserialization and output APIs in the PHP interpreter, and optionally validates APIs that pass their inputs unchecked to the underlying VIF deserialization and output functions. I augmented two existing detection and exploit generation systems as examples that demonstrate the value of Argus' results.

7.2 Evaluation

In this section, I evaluate Argus along two orthogonal dimensions. First, I focus on identifying deserialization and output APIs in the three most popular major versions of the PHP interpreter. The reason for evaluating different interpreter versions is that the number and names of deserialization and output APIs are implementation and version dependent. In the second thrust of the evaluation, I assess how Argus' analysis results improve the accuracy of two example PHP analysis systems – Psalm and FUGIO. To cover these two dimensions, the evaluation answers the following research questions:

RQ1: On the interpreter's call-graph, how many PHP APIs reach the VIF functions (Section 7.2.2), and how many of the reachable APIs pass their arguments to VIF unmodified (Section 7.2.2)?

RQ2: How does the number and identity of deserialization and output APIs change across PHP versions and what are the reasons for the observed changes (Section 7.2.2)?

RQ3: How do Argus' results improve the current state-of-the-art PHP program analysis that target injection vulnerabilities? Does Argus' comprehensive list of injection-sinks lead to the identification of previously unknown POI and XSS vulnerabilities (Section 7.2.3)?

7.2.1 Evaluation Dataset

The evaluation dataset for Argus is divided into two categories corresponding to the two evaluation dimensions. For the experiments on the PHP interpreter, I evaluated Argus on the three most popular major versions (i.e., versions 5, 7, and 8) of the PHP interpreter. As of August 2022, PHP engines of these versions power 99.8% of all live PHP websites, according to W3Tech data (Q-Success, 2023b).

The second dataset is used to evaluate the benefit of Argus' results to existing POI and XSS detection systems as well as exploitation systems. As these systems operate on the code of web applications, rather than the PHP interpreter, I aggregated a dataset corresponding to that purpose. I collected the most popular PHP applications and plugins from a variety of sources. On the one hand, I downloaded the 60 most popular PHP applications based on the reported popularity provided by W3Tech (Q-Success, 2023b). On the other hand, I recognize that large web applications frequently feature a plugin model that allows administrators to customize their sites. As such, I also collected the most downloaded plugins for the popular WordPress, Drupal, and Typo3 web applications from their respective

PHP App Repository	**# of Projects**	**Psalm**	**Minimalist+Psalm**
Drupal plugins	521	0	0
Typo3 plugins	400	0	36
WordPress plugins	996	30	670
Web Apps	60	35	816
Total	1977	**65**	**1522**

Table 7.1: Extending a static analysis tool such as Psalm using our results improved its detection rate by more than 23x.

repositories. Overall, I collected more than 1,950 PHP artifacts (i.e., web applications and plugins). Table 7.1 provides a detailed breakdown in the first two columns.

7.2.2 Analysis of the PHP Interpreter

As the PHP language and ecosystem evolves, the interpreter must provide support and functionality accordingly. Unsurprisingly, this evolution also affects the number and identity of the injection-sink functions provided by different versions of the PHP interpreter. To assess these changes, I evaluate Argus on three different versions of the PHP interpreter (versions 5.6, 7.2, 8.0) as detailed in Table 7.3.

Reachable APIs

The first set of sub-columns labeled as *Detected* for both injection vulnerabilities in Table 7.3 shows the number of APIs that Argus identified as reaching VIF for the three different PHP versions. As the table shows, the number of deserialization APIs for versions 5 and 7 is similar, and two orders of magnitude larger than for version 8. I discuss the difference in the number of reachable deserialization APIs in Section 7.2.2. Furthermore, the number of output APIs for the analyzed PHP versions is almost constant throughout the evaluation.

Validated APIs

The second set of sub-columns labeled *Validated* in Table 7.3 shows the number of APIs that Argus successfully validated by directly passing their input argument to VIF. That is, if an adversary can control input to any of these APIs, the existence of an injection vulnerability (i.e., insecure deserialization or XSS) is a certainty. Validated APIs are a strict subset of reachable APIs. The table shows that Argus was able to consistently validate around 66% and 43% of the deserialization and output APIs, respectively. A closer look at the reachable APIs that failed the validation test shows that either the user is not in control of the input to the VIF or the input is sanitized. For instance, Argus detects the function `highlight_string` reaches the output VIF function (i.e., `php_output_write`), however, the input is sanitized by replacing "<" with "`&lt;`". As a result, the attacker's input does not cause an XSS attack and the function `highlight_string` fails the validation test. In case of deserialization, the `SplTempFileObject::__construct` opens a temporary file object that the user cannot control. As a result, an attacker cannot trick the API to open a malicious PHAR file and validation failed. Table 7.2 contains the complete list of deserialization APIs for PHP version 7.2. Note that the set of APIs in version 7.2 is a strict superset of the APIs in version 5.6. The table also highlights the APIs that still show deserialization capabilities in version 8.0 by typesetting their names in bold. As shown in Table 7.3, all three versions of the PHP interpreter have 22 output APIs, which are exactly the same among all the versions.

Reasons for Differences

Comparing the results for PHP 5.6 with those from 7.2 only shows three additional deserialization APIs (all of which Argus validated). The reason for this increase is the addition of support for the `BMP` image format in PHP 7.2's GD standard graphics library. Specifically, the new `createimagefrombmp` and `imagebmp` functions serve as implicit (i.e., undocu-

	Deserialization API
Category	**PHP API functions**
Phar	*phar::__construct*, *phar::unlinkArchive*, *phar::loadPhar*, *phar::setAlias*, *phar::delete*, *phar::offsetSet*, *phar::setSignatureAlgorithm*, *phar::isValidPharFilename*, *phar::buildFromIterator*, *phar::setDefaultStub*, *phar::mount*, *phar::getType*, *phar::covertToExecutable*, *phar::offsetUnset*, *phar::stopBuffering*, *phar:getATime*, *phar::setStub*, *phar::isLink*, *phar::addFromString*, *phar::isFile*, *phar::addFile*, *phar::compress*, *phar::extractTo*, *phar::hasChildren*, *phar::getInode*, *phar:getFileInfo*, *phar::decompressFiles*, *phar::mapPhar*, *phar:isReadable*, *phar::addEmptyDir*, *phar::compressFiles*, *phar:getOwner*, *phar:getGroup*, *phar::offsetGet*, *phar::setMetadata*, *phar:getPerms*, *phar::isExecutable*, *phar::loadPhar*, *phar::copy*, *phar::convertToData*, *phar::isWritable*, *phar:getSize*, *phar:getCTime*, *phar:getMTime*, *phar:isDir*, *phar::getStub*, *Phar::delMetadata*, *PharFileInfo::__construct*, *PharFileInfo::chmod*, *PharFileInfo::getContent*, *PharFileInfo::getType*, *PharFileInfo::isReadable*, *PharFileInfo::isDir*, *PharFileInfo::isWritable*, *PharFileInfo::openFile*, *PharFileInfo::decompress*, *PharFileInfo::compress*, *PharFileInfo::getInode*, *PharFileInfo::getCTime*, *PharFileInfo::getMTime*, *PharFileInfo::getSize*, *PharFileInfo::isExecutable*, *PharFileInfo::isLink*, *PharFileInfo::isFile*, *PharFileInfo::getATime*, *PharFileInfo::getGroup*, *PharFileInfo::getPerms*, *PharFileInfo::getOwner*, *PharFileInfo::getFileInfo*, *PharFileInfo::setMetadata*, *PharFileInfo::delMetadata*, *PharData::unlinkArchive*, *PharData::loadPhar*, ***phar::getMetadata***, ***PharFileInfo::getMetadata***,
SPL	*FileInfo::openFile*, *FileInfo::getCTime*, *FileInfo::getSize*, *FileInfo::getATime*, *FileInfo::getFileInfo*, *FileInfo::getGroup*, *FileInfo::getType*, *FileInfo::getPerms*, *FileInfo::getOwner*, *FileInfo::isWritable*, *FileInfo::isDir*, *FileInfo::getMTime*, *FileInfo::isReadable*, *FileInfo::getInode*, *FileInfo::isExecutable*, *FileInfo::isFile*, *FileInfo::isLink*, *SplFileObject::__construct*, *SplFileObject::getType*, *SplFileObject::isReadable*, *SplFileObject::isDir*, *SplFileObject::openFile*, *SplFileObject::getInode*, *SplFileObject::isWritable*, *SplFileObject::getFileInfo*, *SplFileObject::getCTime*, *SplFileObject::getPerms*, *SplFileObject::getOwner*, *SplFileObject::getGroup*, *SplFileObject::getATime*, *SplFileObject::getGroup*, *SplFileObject::isExecutable*, *SplFileObject::isFile*, *DirectoryIterator::__construct*, *DirectoryIterator::getType*, *DirectoryIterator::isReadable*, *DirectoryIterator::isDir*, *DirectoryIterator::openFile*, *DirectoryIterator::getInode*, *DirectoryIterator::isWritable*, *DirectoryIterator::getFileInfo*, *DirectoryIterator::getATime*, *DirectoryIterator::getCTime*, *DirectoryIterator::getPerms*, *DirectoryIterator::getOwner*, *DirectoryIterator::getGroup*, *DirectoryIterator::isLink*, *DirectoryIterator::isFile*, *DirectoryIterator::isExecutable*, *RecursiveDirectoryIterator::__construct*, *RecursiveDirectoryIterator::getType*, *RecursiveDirectoryIterator::isReadable*, *RecursiveDirectoryIterator::isDir*, *RecursiveDirectoryIterator::openFile*, *RecursiveDirectoryIterator::getInode*, *RecursiveDirectoryIterator::isWritable*, *RecursiveDirectoryIterator::getFileInfo*, *RecursiveDirectoryIterator::getCTime*, *RecursiveDirectoryIterator::getPerms*, *RecursiveDirectoryIterator::getOwner*, *RecursiveDirectoryIterator::getGroup*, *RecursiveDirectoryIterator::isLink*, *RecursiveDirectoryIterator::current*, *RecursiveDirectoryIterator::isFile*, *RecursiveDirectoryIterator::isExecutable*, *RecursiveDirectoryIterator::hasChildren*, *FileSystemIterator::__construct*, *FileSystemIterator::getType*, *FileSystemIterator::isReadable*, *FileSystemIterator::isDir*, *FileSystemIterator::openFile*, *FileSystemIterator::getInode*, *FileSystemIterator::isWritable*, *FileSystemIterator::getFileInfo*, *FileSystemIterator::getPerms*, *FileSystemIterator::getOwner*, *FileSystemIterator::getGroup*, *FileSystemIterator::getATime*, *FileSystemIterator::current*, *FileSystemIterator::getSize*, *FileSystemIterator::isLink*, *FileSystemIterator::getMTime*, *FileSystemIterator::isExecutable*, *FileSystemIterator::isFile*, ***SplQueue::unserialize***, ***SplStack::unserialize***, ***SplDoublyLinkedList::unserialize***, ***ArrayIterator::unserialize***, ***RecursiveArrayIteratorunserialize***, ***SplObjectStorage::unserialize***, ***ArrayObject::__unserialize***
DOM & XML	*DOMDocument::loadHTMLFile*, *DOM::C14NFile*, *DOMDocument::load*, *DOMDocument::loadXML*, *DOMDocument:saveHTMLFile*, *DOMDocument:relaxNGValidate*, *DOMDocument:validate*, *DOMDocument:save*, *xmlwrite_open_uri*, *xmlreader::open*, *SimpleXMLElement::__construct*, *simplexml_load_file*, *simplexml_load_string*
File Operation	*get_meta_tags*, *is_dir*, *scandir*, *is_writable*, *is_file*, *opendir*, *file*, *move_uploaded_file*, *rmdir*, *fileowner*, *touch*, *gzfile*, *file_get_contents*, *mkdir*, *finfo_file*, *fileatime*, *bzopen*, *fileperms proc_open*, *readgzfile*, *is_link*, *file_put_contents*, *finfo_buffer*, *gzopen*, *getdir*, *unlink*, *is_readable*, *filegroup*, *finfo_open*, *filectime*, *filemtime*, *rename*, *fileinode*, *copy*, *filesize*, *mime_content_type*, *stat*, *filetype*, *fopen*,*readfile*,*file_exists*, *is_executable*
Hash	*md5_file*, *hash_hmac_file*, *sha1_file*, *hash_file*
DataBase	*PDO::pgsqlCopyFromFile*, *PDO:pgsqlCopyToFile*, *pg_trace*
Image Processing	*imageloadfont*, *exifimagetype*, *exif_read_data*, *read_exif_data*, *exif_thumbnail*, *getimagesize*, *imagecreatefromjpeg*, *imagecreatefrompng*, *imagecreatefromgd2*,*imagecreatefromgif*, *imagecreatefromwebp*, *imagecreatefromgd*, *imagecreatefromxbm*, *imagecreatefrombmp*, *imagecreatefromwbmp*, *imagecreatefromavif*, *imagejpeg*, *imagepng*, *imagegif*, *imagegd*, *imagegd2*, *imageavif*, *imagebmp*, *imagewbmp*,*imagexbm*, *imagewebp*
Session Function	***session_decode***, ***session_start***
Communication	*ftp_nb_put*, *ftp_nb_get*, *ftp_get*, *ftp_append*, *ftp_put*, ***msg_recieve***
Deserialization	***unserialize***
	Output API
Database	*pg_loreadall*, *pg_lo_read_all*, *odbc_result_all*
File Operation	*fpassthru*, *readfile*, *readgzfile*, *gzpassthru*, *SplFileObject::fpassthru*
OOP	*class_alias*
Closures	*Closure::bind*, *Closure::bindTo*
Iterators	*CachingIterator::offsetGet*, *RecursiveCachingIterator::offsetGet*
Error Handling	*trigger_error*, *user_error*, *die*, *exit*
General	*echo*, *print*, *print_r*, *vprintf*

Table 7.2: The categories of output and deserialization API.

PHP interpreter	Deserialization API		XSS-leading API	
	Detected	**Validated**	**Detected**	**Validated**
PHP 5.6	419	281 (67%)	54	22 (41%)
PHP 7.2	425	284 (67%)	52	22 (42%)
PHP 8.0	20	13 (65%)	46	22 (48%)
Total	864	578 (66%)	152	66 (43%)

Table 7.3: Our analysis of PHP interpreter shows PHP interpreters prior to version 8.0, contained more than 300 PHP functions that deserialize their arguments or write to output buffer.

mented) deserialization APIs. The last implicit deserialization API missing from PHP 5.6 is the `ftp_append` API which is supported in PHP versions 7.2 and above. All deserialization APIs available in version 5.6 also exist in version 7.2.

In contrast to the small change of deserialization APIs between versions 5.6 and 7.2, the drop from 284 to merely 13 deserialization APIs in version 8.0 is significant. As discussed in Section 2.3.5, prior to version 8.0, any file operation on a phar archive results in the implicit deserialization of the archive's metadata. Fortunately, the PHP developers recognized the negative security consequences this behavior entails in 2020 and voted unanimously to change the default behavior of the phar stream wrapper (Group, 2020). Thus, since PHP 8.0 metadata in phar archives is only deserialized upon an explicit call to the `getMetadata` function in the `Phar` module, and not implicitly on any file operation on the archive. While this change certainly benefits the security of web applications, PHP 8.x is still not widely used by PHP-powered websites (less than 5% at the time of writing) (Q-Success, 2023b). The challenging process of migration prevents most web applications from adopting PHP 8. Therefore, most websites still rely on older versions of the PHP interpreter that include 284 deserialization APIs.

7.2.3 Extending Prior Program Analyses Tools

Argus' value arises from the comprehensive list of output and deserialization APIs it identifies within a PHP interpreter. To demonstrate the value of this information, I extend two PHP analysis systems – Psalm, a static data flow analysis system, and FUGIO, a dynamic automatic exploit generation system targeting POI vulnerabilities.

Psalm Extension

Psalm is a static analysis tool for PHP applications, providing taint analysis and code refactoring (Vimeo, 2021). Psalm's taint analysis operates based on a set of configuration files that specify the taint sources and sinks in the PHP application. For the evaluation, I downloaded the latest available version[1] at the time of writing from its GitHub repository.

Psalm's taint analysis identifies exactly one PHP API function as a taint sink for insecure deserialization: `unserialize`. Furthermore, Psalm includes six functions as taint sinks for XSS vulnerabilities. Argus identified and confirmed 16 additional sinks that are missing in Psalm. To improve Psalm's taint analysis, I extended the set of taint sinks for both XSS and insecure deserialization to include the APIs Argus identified for PHP 7.2. Subsequently, I performed a comparative evaluation between upstream Psalm, and the version incorporating the APIs identified by Argus on the set of 1,977 PHP artifacts described in Section 7.2.1.

The findings in Table 7.1 show a significant increase (i.e., over 10X) in the number of detected insecure deserialization vulnerabilities by the extended version of Psalm. To compare the quality of the results produced by upstream Psalm and our extended version, I manually analyzed all 656 insecure deserialization reports. As Psalm is a static analysis, I expect the results to contain false positives. Furthermore, as the extended version features 284 times as many deserialization sinks, it is unsurprising that it reports 10 times as many

[1]Psalm 4.x-dev@832fc35d8da6e5bb60f059ebf5cb681b4ec2dba5

Category	# of Detected Threats	
	Deserialization	XSS
Phar	0	-
SPL Functions	3	0
DOM & XML	6	0
File Operations	530	15
Hashing Functions	0	-
Image Functions	35	-
Database Funtcions	0	0
Session Function	0	-
Communication Functions	0	-
Unserialize	63	-

Table 7.4: More than 79% of the detected POI vulnerabilities by extended Psalm are related to file operation functions.

potential vulnerabilities. However, what I did not expect is that all 65 reports (i.e., 100%) arising from upstream Psalm are false positives. False positives can arise from web applications that sanitize inputs or, more prevalent in my POI vulnerability analysis, arise from the fact that the application sets a fixed prefix for file-paths. A "fixed" file-path-prefix, even if it is derived from an API such as `dirname` essentially thwarts any attack that relies on the phar module, as the attacker will no longer be able to specify the `phar://` prefix that triggers the stream wrapper.

I confirmed that the extension to Psalm's taint analysis detected 12 previously unknown POI vulnerabilities (i.e., 2% true positives) in the dataset (see Table 7.5). In addition, I confirmed that the extended Psalm detected one previously unknown XSS vulnerability in the core of the WordPress web application. As I will show in Section 7.2.3, FUGIO generated POC exploits for all 12 POI reports supporting the notion that these are actual vulnerabilities. As a case study, I will describe three of the vulnerabilities that we discovered among WordPress and its plugins and how Argus' comprehensive results were necessary to detect them.

Case Study - Feed Them Social

Feed Them Social is one of the most popular plugins on WordPress that allows developers to share social media content on their websites (WordPress, 2022). Using this plugin, a developer can mirror the posts and tweets of a specific account on Twitter, Facebook, and other social media sites.

The detected vulnerability for this plugin is an unauthenticated insecure deserialization which resides in the functionality of the module's Twitter feed. The Twitter feed in this plugin retrieves and shows the content of tweets including any referenced media on a WordPress page. Whenever a tweet contains a URL, the plugin attempts to retrieve the URL's title, image, and description to display on the WordPress page. To do this, the plugin uses the function `get_meta_tags` with unsanitized user-input directly from the tweet to retrieve the metadata of the specified URL. Listing 7.3 shows the simplified version of this vulnerability in this plugin, where the unsanitized user-input is passed to the implicit deserialization API `get_meta_tags` on line 4.

In order to exploit this vulnerability, an attacker can simply set the `fts_url` request parameter to the path of a phar file with malicious metadata. Note that, the `$_REQUEST` variable populates the variables sent through `$POST` and `$GET` requests by the user to the server. When the plugin tries to read and parse the metadata of the passed URL, it will automatically deserialize the metadata of the malicious phar file. `get_meta_tags` is an implicit deserialization API identified by Argus and not taken into consideration by prior work demonstrating the necessity of Argus' comprehensive analysis.

```
function fts_twitter_share_url_check() {
  $twitter_external_url = $_REQUEST['fts_url'];
  // ...
  $tags = get_meta_tags($twitter_external_url);
  ...
}
```

Listing 7.3: The feed them social plugin passes unsanitized user-input to the function get_meta_tags.

Case Study - Ajax Load More

The Ajax Load More plugin is one of the most popular plugins in WordPress that provides infinite scrolling for posts, pages, and comments. The vulnerability identified by the extended Psalm resided in the so-called repeater template in this plugin. The function *alm_repeaters_export* exports repeater templates for backup or sharing with other WordPress sites. Listing 7.4 abbreviates the vulnerable code in the Ajax Load More plugin. As the attacker has full control over the `file` variable (Line 3) which gets passed to the implicit deserialization API (i.e.,`file_exists`) at Line 4 without sanitization the plugin suffers from insecure deserialization.

```
1 function alm_repeaters_export() {
2   // check privileges
3   $file = $_POST['alm_repeaters_export'];
4   if ( file_exists( $file ) ) {
5           // ...
6   }
7 }
```

Listing 7.4: The Ajax load more plugin passes unsanitized user-input to the function file_exists.

Fortunately, Ajax Load More checks whether the user is authorized to export templates before any file operation, preventing a devastating pre-authentication vulnerability. Unfortunately, however, the plugin does not check cross site request forgery (CSRF) nonces, allowing an adversary to exploit the vulnerability via a CSRF attack against an authorized user.

Case Study - XSS in WordPress Core

WordPress is the most popular CMS, as it powers more than 43% of all live websites (**?**). The detected vulnerability in the core of WordPress is a cross-site scripting vulnerability that resides in the multi-site feature of the web application. This feature allows developers to host multiple websites on one installation of WordPress and create a network of websites. However, this feature also allows attackers to perform a XSS attack on WordPress. In particular, this vulnerability resides in a file called `ms-files.php`, which was detected by the extended Psalm. Listing 7.5 shows the simplified version of this vulnerability, where the function `readfile` writes the content of a file chosen by an attacker to the output buffer (i.e., the HTML response) without any sanitization (Line 5). Note that, none of the existing tools identify `readfile` as a vulnerable API to XSS attacks. As a result, Psalm can only detect this XSS vulnerability if it is extended using Argus' results.

```
1 // wp-includes/ms-files.php
2 ...
3 $file = rtrim(BLOGUPLOADDIR, '/').'/'.
4         str_replace('..', '', $_GET['file']);
5 ...
6 readfile( $file );
7
```

Listing 7.5: WordPress uses the function readfile to output the content of a file to the browser without sufficient sanitization.

In order to exploit this vulnerability, an attacker sets the `file` request parameter to the path of the malicious file (Line 3). The malicious file contains JavaScript code, which gets written in the HTML response by the function `readfile`. Next, the browser executes the malicious JavaScript on the client's browser.

FUGIO Extension

FUGIO (Park et al., 2022) is an automatic exploit generator for deserialization vulnerabilities in PHP applications. FUGIO's exploit generation operates based on hooking a set of predefined deserialization functions while sending serialized objects as request to the web application under test. The analysis of FUGIO shows that FUGIO hooks into 26 file operation functions in the PHP interpreter as well as the `unserialize` function to intercept deserialization of user-input. Similar to Psalm, FUGIO obtained the list of hooked functions through manual analysis of PHP documentation and prior works such as Thomas (Thomas, 2018). For the evaluation, I downloaded FUGIO from its GitHub repository at `https://www.github.com/WSP-LAB/FUGIO`.

One should note that FUGIO states that it is not a vulnerability detection tool. Rather its core contribution is to generate exploits for already known deserialization vulnerabilities (Park et al., 2022), such as those identified by Psalm. As a result, I evaluated FUGIO on the 12 vulnerabilities that the extended version of Psalm detected. To extend FUGIO, I modified its source code to hook the comprehensive set of deserialization API functions identified by Argus. The last two columns in Table 7.5 show the results of extending FUGIO using Argus when generating exploits for the discovered vulnerabilities by Psalm+Argus.

As a dynamic analysis system, FUGIO requires a runtime environment. To this end, I created an experimental environment for WordPress plugins consisting of Nginx, PHP 7.2, MySQL 8, and WordPress 5.4. FUGIO creates attacks by stitching together so-called gadgets into a POP-chain. However, WordPress alone does not contain any gadgets that could be used for remote code execution attacks. In practice, administrators customize their WordPress installations using plugins and themes. Thus to ensure that FUGIO has gadgets to work with, I installed the latest versions of the top ten most popular plugins in WordPress in the experimental environment (WordPress, 2022). During the experiment, FUGIO without Argus' results does not hook into the image functions listed in Table 7.2. As a result, FUGIO was unable to generate an exploit for two of the discovered vulnerabilities in Table 7.5. However, the extended FUGIO+Argus successfully generated exploits for *all* the discovered vulnerabilities listed in Table 7.5. On this small sample, this indicates a 20% increase in the number of generated exploits.

Web App	**Plugin**	**Vuln. Type**	**CVE**	**P**	**P+A**	**F**	**F+A**
Xoops	-	1	-	✗	✓	✗	✓
WordPress	Feed them Social	1	CVE-2022-2437	✗	✓	✗	✓
	ImageMagick	2	CVE-2022-2441	✗	✓	✓	✓
	String locator	2	CVE-2022-2434	✗	✓	✓	✓
	Ajax load more	2	CVE-2022-2433	✗	✓	✓	✓
	Broken link checker	3	CVE-2022-2438	✗	✓	✓	✓
	wp editor	3	CVE-2022-2446	✗	✓	✓	✓
	Visualizer	3	CVE-2022-2444	✗	✓	✓	✓
	Easy digital download	3	CVE-2022-2439	✗	✓	✓	✓
	Theme Editor	3	CVE-2022-2440	✗	✓	✓	✓
	wPvivid Backup	3	CVE-2022-2442	✗	✓	✓	✓
	Download manager	3	CVE-2022-2436	✗	✓	✓	✓
	Core	XSS	-	✗	✓	-	-
Total	-	-	-	0	13	10	12

Table 7.5: I verified the reports of Psalm+Argus by discovering 13 previously unkown POI and XSS vulnerabilities. The vulnerability types 1, 2, and 3 refers to Unauthenticated Phar deserialization, CSRF to Phar deserialization, and Authenticated Phar deserialization, respectively.

Disclosure. Of course, I responsibly reported all the vulnerabilities to their corresponding developer teams and notified the WordPress plugin review team of my findings. Seven teams already patched their WordPress plugins, and WordFence assigned CVE numbers to the vulnerabilities as shown in Table 7.5.

To summarize the evaluation, I evaluated Argus along two dimensions. First, I evaluated Argus across three different versions of the PHP interpreter. I identified, analyzed, and explained the reasons for the varying numbers of deserialization APIs in each version. Along the second dimension, I used Argus' results to extend two existing PHP security analysis tools – Psalm and FUGIO. I showed how Argus' comprehensive list of injection-sinks improves the results of both tools yielding 13 previously unknown XSS and POI vulnerabilities and corresponding POC exploits.

7.3 Discussion

In this section, I discuss the limitations of Argus and other aspects affecting the relevance of this work.

As mentioned in Section 7.1.2, Argus relies on a reachability analysis on the call-graph to identify the serialization and output APIs in the PHP interpreter. However, I perform a validation step to verify the output of the reachability analysis since the call-graph analysis does not reason about any sanitization or filtering the PHP interpreter might perform. While it seems more pertinent to perform a data-flow analysis rather than the reachability analysis, I argue that Argus needs to reason about the PHP interpreter and its extensions that it is linked against. Ignoring additional challenges to practicality (e.g., extensions relying on non-C code), the analysis needs to scale to millions of lines of code across PH (one million lines of C code alone). Needless to say, resolving function pointers is still a prominent challenge for existing data-flow analysis including the state-of-the-art SVF tool (Sui and Xue, 2016), which leads to imprecise control-flow graphs. As a result, I opted for a reachability analysis and subsequently validation in Argus to identify injection-sinks in the PHP interpreter.

Furthermore, the evaluation identified two sets of injection-sink APIs for PHP: 1) APIs that directly deserialize or output their arguments, and 2) APIs that operates on malicious files. As mentioned in Section 2.3.5, the phar stream wrapper in the PHP interpreter only operates on local phar files. As a result, to exploit any APIs in the second category, the attacker needs to upload the phar file prior to invoking the insecure deserialization. Therefore, in order to confirm the detected vulnerabilities, I made the assumption that the attacker had already uploaded the malicious phar file to the web application's server. I argue that this assumption is realistic since there are a plethora of approaches where an attacker can upload malicious phar files, which include exploiting arbitrary file upload vulnerabilities (Huang et al., 2019; Huang et al., 2021). Furthermore, web applications and their plugins provide upload functionality for many purposes, such as uploading plugins, profile pictures, and PDF files. An attacker can use such functionality to upload malicious files to the web application's installation. In order to bypass the upload restrictions such as file type and metadata checks, one can use PHPGCC (Security, 2018) to modify the phar file into ZIP, PDF, and image formats such as JPEG.

Finally, as the evaluation demonstrates, the PHP developers noticed the security consequences of automatic deserialization of phar files and fixed this issue in PHP 8.0 (released in November 2020). However, the PHP usage statistics indicate that, at the time of writing, only 4.5% of all websites that rely on PHP actually operate on PHP 8.0 (Q-Success, 2023b). The reason for this low adoption rate is probably that transitioning to PHP 8.0 is a non-trivial procedure for most PHP-powered websites. The major changes in the PHP interpreter 8.0 compared to previous versions lead to backward incompatible changes (Group, 2022) which can potentially cause fatal errors in the web applications. The challenge of (in-)compatibility is evidenced by the most popular PHP application – WordPress. Although efforts within the WordPress project to support PHP version 8.0 began way before the official release of the language, at the time of writing, WordPress still warns users that even its latest stable version (released in 2022) is not fully compatible with version 8 yet (WordPress, 2022). While the PHP interpreter has addressed the threat arising from the automatic deserialization of phar files in version 8, history suggests that web sites relying on older versions of PHP are likely to remain publicly accessible on the Internet for a long time to come. These will continue to include the over 280 deserialization APIs provided by their PHP runtimes.

Chapter 8

Conclusions

In this thesis, I present and evaluate a range of methodologies and frameworks to improve the security of PHP web applications through runtime defenses, debloating the source-code of PHP web applications, and detecting injection sink APIs in the PHP runtime.

8.1 Scope of Contributions

The work presented in this thesis focuses on the PHP interpreter as well as the web application developed in the PHP programming language. As of 2022, the PHP interpreter had 77.4% of the global market share among all live websites (Q-Success, 2023b). Furthermore, the most popular web applications, including WordPress, are written in PHP (Q-Success, 2023a). Moreover, PHP web applications are among the web applications with the highest number of detected vulnerabilities. In 2019, 2,652 vulnerabilities were detected in PHP web applications, which increased by 12.7% compared to 2018 (Bekerman and Yerushalmi, 2019). Due to the dominance of PHP among web applications, in this thesis I focus on the PHP interpreter and its web applications.

8.2 Summary of Major Contributions

Existing prevention techniques against various attacks on web applications are limited due to a lack of consideration for how the PHP web applications communicate with the host OS and its resources as well as a lack of fine-grained analysis of the PHP web applications. Furthermore, the absence of an integrated analysis of both the PHP interpreter and web applications leaves PHP web applications vulnerable to injection vulnerabilities such as insecure deserialization. This thesis claims that to improve the security of PHP web applications, an integrated analysis of both the interpreter and the web application is required in the application and the interpreter source-code. To this end, this thesis provided techniques to analyze PHP web applications and the interpreter to provide a defense mechanism against SQLi and RCE attacks, remove security vulnerabilities, and detect previously unknown XSS and deserialization vulnerabilities in PHP web applications.

This thesis proposes a hybrid static-dynamic technique to defend PHP web applications against RCE attacks. Prior work mitigated RCE through static taint analysis, code property graphs, and dynamic taint analysis. However, the dynamic language features of the PHP programming language (e.g., dynamic includes and class autoloaders) and the runtime overhead of dynamic analysis can be challenging for such techniques. I first introduced an abstraction-aware technique to create a system-call sandbox for PHP web applications in a tool called Saphire. I then evaluated our system on the most popular PHP web applications, which reduces the set of available system-calls to each PHP file by 80.5%, on average. Furthermore, Saphire successfully defended against 21 RCE vulnerabilities in our dataset.

I then introduced an approach to protect web applications against SQLi attacks. While existing systems are only able to defend against specific types of SQLi attacks, SQLBlock provides a defense mechanism against more SQLi attacks compared to prior techniques. SQLBlock is a hybrid static-dynamic approach that generates a profile of benign issued queries by PHP web applications. I demonstrated that SQLBlock successfully defends against all attacks on 11 SQLi vulnerabilities in the dataset, compared to defending against only five in prior works.

This thesis then proposed Minimalist, a semi-automated debloating technique to remove unnecessary functionality from PHP web applications. Minimalist statically generates the call-graph of a web application and utilizes prior user interaction with the web application to identify unnecessary functionalities. Our evaluation shows that, compared to prior work, Minimalist debloats 17.78% of the source-code in web applications while raising no false positives.

I finally demonstrated a principled and systematic analysis of the PHP interpreter to identify injection sink APIs. Compared to manual analysis in prior work, Argus detected more than 280 APIs in the PHP interpreter that can deserialize user-input or lead to XSS vulnerabilities through an analysis of the PHP interpreter and its extensions. To demonstrate the effectiveness of Argus, I extended existing vulnerability detection systems and identified 13 unknown deserialization and XSS vulnerabilities.

In conclusion, this thesis claimed that an array of hybrid static-dynamic protection and detection mechanisms over the PHP interpreter and web applications can improve the underlying security of PHP web applications. In response, this work provides the following:

- Hybrid static-dynamic analysis on the PHP interpreter and the web applications with Saphire, which generates a system-call profile to sandbox the PHP web applications and protect them against RCE attacks (Bulekov et al., 2021).

- Hybrid static-dynamic analysis on the PHP web applications, which generates SQL profiles for benign issued queries by the web application to defend against SQLi attacks (Jahanshahi et al., 2020).

- A semi-automated debloating mechanism, which statically analyzes PHP web applications and takes prior user-interaction into account to identify and debloat unnecessary functionalities.

- Hybrid static-dynamic analysis of the PHP interpreter to identify the set of APIs that can lead to insecure deserialization or XSS vulnerabilities.

References

(2022). Check point research: Third quarter of 2022 reveals increase in cyberattacks and unexpected developments in global trends. https://blog.checkpoint.com/2022/10/26/third-quarter-of-2022-reveals-increase-in-cyberattacks/.

(2022). Cost of a data breach 2022 | ibm. https://www.ibm.com/reports/data-breach.

Abubakar, M., Ahmad, A., Fonseca, P., and Xu, D. (2021). *shard*: Fine-grained kernel specialization with context-aware hardening. In *Proceedings of the 30th USENIX Security Symposium*. Available on https://www.usenix.org/conference/usenixsecurity21/presentation/abubakar.

Acharya, A. and Raje, M. (2000). Mapbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Conference on Security Symposium*.

Agadakos, I., Jin, D., Williams-King, D., Kemerlis, V. P., and Portokalidis, G. (2019). Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*.

Antunes, N. and Vieira, M. (2010). Benchmarking vulnerability detection tools for web services. In *IEEE International Conference on Web Services*, pages 203–210.

Apache (2021). Log files - apache http server. https://httpd.apache.org/docs/2.4/logs.html.

Apache Software Foundation (2018). ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., and Ernst, M. D. (2010). Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Transactions on Software Engineering*, 36(4):474–494. https://doi.org/10.1109/TSE.2010.31.

Azad, B. A. (2022). Less is More Source Code. https://debloating.com.

Azad, B. A., Laperdrix, P., and Nikiforakis, N. (2019). Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium*.

Backes, M., Rieck, K., Skoruppa, M., Stock, B., and Yamaguchi, F. (2017). Efficient and flexible discovery of php application vulnerabilities. In *IEEE European Symposium on Security and Privacy*.

Baek, Y.-M. and Bae, D.-H. (2016). Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 238–249.

Balde Samit, Barantsev Alexei, B. E. (2018). Selenium - web browser automation. `https://docs.seleniumhq.org/`.

Bandhakavi, S., Bisht, P., Madhusudan, P., and Venkatakrishnan, V. (2007). Candid: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 12–24.

Bekerman, D. and Yerushalmi, S. (2019). Imperva. `https://www.imperva.com/blog/the-state-of-vulnerabilities-in-2019/`.

Bekerman, D. and Yerushalmi, S. (2020). Imperva. `https://www.imperva.com/resources/resource-library/reports/the-state-of-vulnerabilities-in-2020/`.

Bernardo, G. and Miroslav, S. (2019). sqlmap: automatic sql injection tool. `https://sqlmap.org`.

Bernaschi, M., Gabrielli, E., and Mancini, L. V. (2000). Enhancements to the linux kernel for blocking buffer overflow based attack. In *Proceedings of the 4th Annual Linux Showcase & Conference*.

Bernaschi, M., Gabrielli, E., and Mancini, L. V. (2002). Remus: A security-enhanced operating system. *ACM Transactions on Information and System Security*, 5(1):36–61. `https://doi.org/10.1145/504909.504911`.

Boyd, S. W. and Keromytis, A. D. (2004). Sqlrand: Preventing sql injection attacks. In *International Conference on Applied Cryptography and Network Security*, pages 292–302.

Buehrer, G., Weide, B. W., and Sivilotti, P. A. (2005). Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113.

Bulekov, A., Jahanshahi, R., and Egele, M. (2021). Saphire: Sandboxing php applications with tailored system call allowlists. In *Proceedings of the 30th USENIX Security Symposium*.

Carettoni, L. (2021). SerialKiller. `https://github.com/ikkisoft/SerialKiller`.

CISA (2021). 2021:Top Routinely Exploited Vulnerabilities. https://www.cisa.gov/news-events/cybersecurity-advisories/aa22-117a.

Commons, A. (2021). ValidatingObjectInputStream. https://github.com/apache/commons-oi.

copernica (2022). A c++ library for developing PHP extension. http://www.php-cpp.com/documentation/.

Corporation, O. (2019). https://dev.mysql.com/doc/refman/8.0/en/.

Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., and Gligor, V. (2000). Subdomain: Parsimonious server security. In *Proceedings of the 14th USENIX Conference on System Administration*.

Dahse, J. and Holz, T. (2014a). Simulation of built-in php features for precise static code analysis. In *Network and Distributed Systems Security Symposium*.

Dahse, J. and Holz, T. (2014b). Static detection of second-order vulnerabilities in web applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium*.

Dahse, J., Krein, N., and Holz, T. (2014). Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*.

Docker (2018). Docker: Enterprise container platform. https://www.docker.com/.

Doupé, A., Cavedon, L., Kruegel, C., and Vigna, G. (2012). Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Conference on Security Symposium*, pages 26–26.

Drupal (2016). https://www.drupal.org/docs/7/api/database-api.

Esser, S. (2010). Utilizing code reuse/rop in php application exploits. In *BlackHat USA*. Available on http://media.blackhat.com/bh-us-10/presentations/Esser/BlackHat-USA-2010-Esser-Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits-slides.pdf.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*.

Garfinkel, T., Pfaff, B., Rosenblum, M., et al. (2004). Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium*.

GCC (2022). Code Gen Options - using the GNU Compiler Collection. https://gcc.gnu.org/onlinedocs/gcc-4.4.7/gcc/Code-Gen-Options.html.

Geniar, M. (2014). Code obfuscation, php shells & more: what hackers do once they get passed your (php) code. `https://github.com/mattiasgeniar/php-exploit-scripts`.

Ghavamnia, S., Palit, T., Benameur, A., and Polychronakis, M. (2020a). Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses*.

Ghavamnia, S., Palit, T., Mishra, S., and Polychronakis, M. (2020b). Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*.

Goldberg, I., Wagner, D., Thomas, R., and Brewer, E. A. (1996). A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th USENIX Conference on Security Symposium, Focusing on Applications of Cryptography*.

Group, T. P. (2020). PHP:rfc phar stop autoloading metadata. `https://wiki.php.net/rfc/phar_stop_autoloading_metadata`.

Group, T. P. (2022). PHP:The Backward Incompatible Changes. `https://www.php.net/manual/en/migration80.incompatible.php`.

Halfond, W., Orso, A., and Manolios, P. (2008). Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *IEEE Transactions on Software Engineering*, 34:65–81.

Halfond, W. G., Viegas, J., Orso, A., et al. (2006). A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*.

Halfond, W. G. J. and Orso, A. (2005). Amnesia: Analysis and monitoring for neutralizing sql-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 174–183.

Heo, K., Lee, W., Pashakhanloo, P., and Naik, M. (2018). Effective program debloating via reinforcement learning. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

Hills, M., Klint, P., and Vinju, J. (2013). An empirical study of php featuers usage: a static analysis perspective. In *Proceedings of the International Symposium on Software Testing and Analysis*.

Hofmeyr, S. A., Forrest, S., and Somayaji, A. (1998). Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180.

Huang, J., Li, Y., Zhang, J., and Dai, R. (2019). Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*.

Huang, J., Zhang, J., Liu, J., Li, C., and Dai, R. (2021). Ufuzzer: Lightweight detection of php-based unrestricted file upload vulnerabilities via static-fuzzing co-analysis. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*.

IBM (2021). What is LAMP stack? `https://www.ibm.com/topics/lamp-stack`.

Jahanshahi, R., Doupé, A., and Egele, M. (2020). You shall not pass: Mitigating sql injection attacks on legacy web applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 445–457.

Jain, K. and Sekar, R. (2000). User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium*.

Jamrozik, K., von Styp-Rekowsky, P., and Zeller, A. (2016). Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering*.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*.

Kim, N. (2022). Function graph tracer for c/c++/rust. `https://github.com/namhyung/uftrace`.

Kim, T. and Zeldovich, N. (2013). Practical and effective sandboxing for non-root users. In *Proceedings of the 22nd USENIX Conference on Annual Technical Conference*.

Koishybayev, I. and Kapravelos, A. (2020). Mininode: Reducing the attack surface of node. js applications. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses*.

Koo, H., Ghavamnia, S., and Polychronakis, M. (2019). Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*.

Koutroumpouchos, N., Lavdanis, G., Veroni, E., Ntantogian, C., and Xenakis, C. (2019). Objectmap: Detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics*.

Lei, L., Sun, J., Sun, K., Shenefiel, C., Ma, R., Wang, Y., and Li, Q. (2017). SPEAKER: Split-Phase Execution of Application Containers. In Polychronakis, M. and Meier, M., editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 230–251. Springer. `https://doi.org/10.1007/978-3-319-60876-1_11`.

Linn, C., Rajagopalan, M., Baker, S., Collberg, C. S., Debray, S. K., and Hartman, J. H. (2005). Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Conference on Security Symposium*.

Liu, A., Yuan, Y., Wijesekera, D., and Stavrou, A. (2009). Sqlprob: a proxy-based architecture towards preventing sql injection attacks. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 2054–2061.

Loscocco, P. and Smalley, S. (2001). Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*.

Machiry, A., Tahiliani, R., and Naik, M. (2013). Dynodroid: An input generation system for android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234.

McConnell, S. (2004). *Code complete*. Pearson Education.

Medeiros, I., Beatriz, M., Neves, N., and Correia, M. (2016). Hacking the dbms to prevent injection attacks. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 295–306.

Medeiros, I., Beatriz, M., Neves, N., and Correia, M. (2019). Septic: Detecting injection attacks and vulnerabilities inside the dbms. *IEEE Transactions on Reliability*, 68(3):1168–1188. https://doi.org/10.1109/TR.2019.2900007.

Merlo, E., Letarte, D., and Antoniol, G. (2007). Automated protection of php applications against sql-injection attacks. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 191–202. IEEE.

Metasploit (2019). metasploit. https://www.metasploit.com.

Mishra, S. and Polychronakis, M. (2018). Shredder: Breaking exploits through api specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference*.

Mishra, S. and Polychronakis, M. (2020). Saffire: Context-sensitive function specialization and hardening against code reuse attacks. In *IEEE European Symposium on Security & Privacy*.

Mishra, S. and Polychronakis, M. (2021). Sgxpecial: Specializing sgx interfaces against code reuse attacks. In *Proceedings of the 14th European Workshop on Systems Security*.

Moore, P. (2018). libseccomp. https://github.com/seccomp/libseccomp.

Mutz, D., Valeur, F., Vigna, G., and Kruegel, C. (2006). Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):61–93. https://doi.org/10.1145/1127345.1127348.

Naderi-Afooshteh, A., Nguyen-Tuong, A., Bagheri-Marzijarani, M., Hiser, J. D., and Davidson, J. W. (2015). Joza: Hybrid taint inference for defeating web application sql injection attacks. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 172–183.

Nguyen-Tuong, A., Guarnieri, S., Greene, D., Shirley, J., and Evans, D. (2005). Automatically Hardening Web Applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing*.

NIST (2021). Nvd - vulnerability metrics. `https://nvd.nist.gov/vuln-metrics/cvss`.

OpenJDK (2021). Jep 290: Filter incoming serialization data. `https://openjdk.org/jeps/290`.

OWASP (2017). OWASP Top Ten 2017 | OWASP Foundation. `https://owasp.org/www-project-top-ten/2017/Top_10`.

OWASP (2023). A03 Injection-OWASP Top 10. `https://owasp.org/www-community/Injection_Flaws`.

Park, S., Kim, D., Jana, S., and Son, S. (2022). FUGIO: Automatic exploit generation for PHP object injection vulnerabilities. In *Proceedings of the 31st USENIX Security Symposium*.

PCWorld Staff (2011). phpmyadmin. `https://www.pcworld.com/article/233948/phpmyadmin.html`.

PHP (2021). Php built-in functions and methods. `https://www.php.net/manual/en/indexes.functions.php`.

PHP (2022). PHP: PHP Manual. `https://www.php.net/manual/en/index.php`.

PortSwigger (2019). `https://portswigger.net/burp`.

Prevelakis, V. and Spinellis, D. (2001). Sandboxing applications. In *Proceedings of the FREENIX Track: USENIX Annual Technical Conference*.

Provos, N. (2003). Improving host security with system call policies. In *Proceedings of the 12th USENIX Conference on Security Symposium*.

Q-Success (2023a). Usage Statistics and Market Share of Content Management Systems for Websites, November 2023. `https://w3techs.com/technologies/overview/content_management/all`.

Q-Success (2023b). Usage Statistics and Market Share of PHP for Websites. `https://w3techs.com/technologies/details/pl-php`.

Qian, C., Koo, H., Oh, C., Kim, T., and Lee, W. (2020). Slimium: Debloating the chromium browser with feature subsetting. In *CCS'20: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 461–476. https://doi.org/10.1145/3372297.3417866.

Quach, A., Prakash, A., and Yan, L. (2018). Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*.

Ray, D. and Ligatti, J. (2012). Defining code-injection attacks. In *POPL '12: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 47, pages 179–190. https://doi.org/10.1145/2103656.2103678.

Redini, N., Wang, R., Machiry, A., Shoshitaishvili, Y., Vigna, G., and Kruegel, C. (2019). Bintrimmer: Towards static binary debloating through abstract interpretation. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

Rethans, D. (2018). Xdebug: debugger and profiler tool for PHP. https://xdebug.org/.

Saxena, P., Molnar, D., and Livshits, B. (2011). Scriptgard: Automatic context-sensitive sanitization for large-scale legacy web applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*.

Seaborn, M. (2019). Plash: tools for practical least privilege. http://plash.beasts.org.

Security, A. (2018). PHPGCC:PHP Generic Gadget Chain. https://github.com/ambionics/phpggc.

Security, C. (2021). contrast-rO0. https://github.com/Contrast-Security-OSS/contrast-rO0.

Security, O. (2019). Exploit database. https://exploit-db.com.

Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P. (2001). A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*.

Shcherbakov, M. and Balliu, M. (2021). Serialdetector: Principled and practical exploration of object injection vulnerabilities for the web. In *Network and Distributed Systems Security (NDSS) Symposium*.

Slizov, V. (2019). php-parser. https://github.com/z7zmey/php-parser.

Snyder, P., Taylor, C., and Kanich, C. (2017). Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

Somayaji, A. and Forrest, S. (2000). Automated response using system-call delays. In *Proceedings of the 9th USENIX Conference on Security Symposium*.

Son, S., McKinley, K. S., and Shmatikov, V. (2013). Diglossia: detecting code injection attacks with precision and efficiency. In *Proceedings of the 20th ACM SIGSAC conference on Computer*, pages 1181–1192.

Son, S. and Shmatikov, V. (2011). Saferphp: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*.

Su, Z. and Wassermann, G. (2006). The essence of command injection attacks in web applications. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

Sui, Y. and Xue, J. (2016). Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*.

Sun, S.-T. and Beznosov, K. (2008). Sqlprevent: Effective dynamic detection and prevention of sql injection attacks without access to the application source code. Technical report LERSSE-TR-2008-01, Laboratory for Education and Research in Secure Systems Engineering University of British Columbia. Available on `http://lersse-dl.ece.ubc.ca/record/146/files/146.pdf`.

Tanaka, K. and Saito, T. (2018). Python deserialization denial of services attacks and their mitigations. In *International Conference on Computational Science/Intelligence & Applied Informatics*.

Thomas, S. (2018). File Operation Induced Unserialziation via the phar Stream Wrapper. In *Blackhat - USA*. Available on `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`.

Vimeo (2021). Psalm - a static analysis tool for PHP. `https://psalm.dev`.

Wagner, D. (1999). Janus: An approach for confinement of untrusted applications. Technical report, University of California at Berkeley. Available on `https://www2.eecs.berkeley.edu/Pubs/TechRpts/1999/CSD-99-1056.pdf`.

Wagner, D. and Dean, D. (2001). Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*.

Wan, Z., Lo, D., Xia, X., Cai, L., and Li, S. (2017). Mining sandboxes for linux containers. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

Wassermann, G. and Su, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Wheeler, D. A. (2004). `https://dwheeler.com/sloccount/sloccount.html`.

WordPress (2022). Plugins Categorized as Popular. `https://wordpress.org/plugins/browse/popular/`.

WordPress (2022). Server Environment: Make WordPress Hosting. `https://make.wordpress.org/hosting/handbook/server-environment/`.

WordPress (2022). Wordpress developer resources. `https://wordpress.org/plugins/browse/featured/`.

Zephir (2022). Building php extensions with zephir. `https://docs.zephir-lang.com/`.

Zheng, C. and Huang, H. (2018). Daemon-guard: Towards preventing privilege abuse attacks in android native daemons. In *Proceedings of the First Workshop on Radical and Experiential Security*.

Zheng, Y. and Zhang, X. (2013). Path sensitive static analysis of web applications for remote code execution vulnerability detection. In *35th International Conference on Software Engineering (ICSE)*.

CURRICULUM VITAE

[redacted]