THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

Testability Tarpits - Navigating the Challenges of Static Tools in Web Applications

Thèse présentée et soutenue à Biot, le 26/06/2023, par

Feras Al-Kassar

Rapporteurs	Prof. Tamara Rezk	INRIA
	Prof. Martin Johns	TU Braunschweig
Examinateurs	Dr. Johannes Dahse	SonarSource
	Dr. Simone Aonzo	EURECOM
Directeur de thèse	Prof. Davide Balzarotti	EURECOM
Co-directeur de thèse	Dr. Luca Compagna	SAP

Acknowledgements

I would like to thank the reviewers for taking the time to read my thesis and provide me with feedback, and the jury members for attending my PhD defense.

I would also like to thank my supervisors, Prof. Davide Balzarotti and Dr. Luca Compagna, who guided me throughout my PhD. It was a pleasure to work with them and learn from their professional experiences. Their help and advice were invaluable in producing works that were accepted in top-tier conferences. Additionally, I would like to thank my team at SAP company and my colleagues at Eurecom University.

I wish to express my gratitude to my friends and family, including my father Zohair and my two sisters Rasha and Reem, as well as my new tiny family consisting of my wife Rawan and my cat Giroflee. They have provided me with all the love and support that I need, especially during the most difficult periods.

Finally and most importantly, I dedicate this dissertation to my mother, Samira, who passed away during my PhD. Without you, the child whom you taught for years and years would not have achieved this milestone. The journey of a PhD is a challenging one, and it becomes even more daunting when you lose someone you love. This work is for you, my mother, a small thank you for everything you did for me. I will always miss you.

Abstract

Improving the precision of static application security testing (SAST) is crucial in the battle against critical vulnerabilities and to boost the security of the Web. Nevertheless, modern code poses a challenge for even the most advanced commercial tools, as they have several blind spots that limit their ability to conduct a thorough analysis and detect intricate vulnerabilities. Furthermore, despite the well-known limitations of static application security testing tools (SAST), the effect of coding style on their vulnerability detection capability has yet to be thoroughly examined.

The goal of this thesis is to evaluate the effectiveness of a combination of commercial and open source security scanners. Through experimentation, we identified various code patterns that hinder the ability of state-of-the-art tools to analyze projects. By detecting these patterns during the software development lifecycle, our approach can offer valuable feedback to developers regarding the testability of their code. Additionally, it enables them to more accurately evaluate the residual risk that their code might still contain vulnerabilities, even if static analyzers report no findings. Our approach also suggests alternative methods to transform the code and enhance its testability for SAST.

Based on our experiments, it is evident that testability tarpits are prevalent. To address this issue, we introduce WHIP, the first approach that promotes collaboration among SAST tools by sharing information to overcome their individual limitations. Our technique operates solely on the application source code, leveraging various tools as an oracle to detect indications of disrupted data flows. When we identify such barriers, we insert alternative pathways that bypass the problematic code section, allowing SAST tools to effectively handle it.

Our final investigation focused on evaluating the performance of SAST tools when analyzing applications built using frameworks with specific design patterns, such as Model View Controller (MVC). In this study, we explored the interaction between MVC frameworks and the application source

code. We discovered the obstacles that SAST tools encounter in comprehending this relationship, and devised a solution to "disconnect" the application from the framework. Our novel approach operates exclusively on the application source code, converting the highly dynamic interaction challenges into static code that SAST tools can test. This approach facilitates the SAST of MVC-based web applications without necessitating any modifications to the SAST tools themselves.

Contents

List of Publications

- Feras Al Kassar, Giulia Clerici, Luca Compagna, Fabian Yamaguchi, Davide Balzarotti. "Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications". In NDSS, 2022.
- Feras Al Kassar, Luca Compagna, Davide Balzarotti. "WHIP: Improving Static Vulnerability Detection in Web Application by Forcing tools to Collaborate". In Usenix, 2023.
- Feras Al Kassar, Luca Compagna, Davide Balzarotti. "Mitigating the Impact of the MVC Design Pattern on Web Application Static Security Testing". Under submission to ESEC/FSE 2023.

Chapter 1

Introduction

1.1 Problem Statement

According to an article in Nasdaq [26], it is estimated that 95% of purchases will be made online by 2040. Currently, besides the big e-commerce enterprises, there are 400 million small businesses worldwide [22], and 71% of these businesses have their own websites [9]. Unfortunately, The rapid growth and evolution of the web have been accompanied by a corresponding increase in the frequency and severity of web-based vulnerabilities, which ranked as the second leading cause of data breaches in 2019 [148]. The high number of vulnerabilities in the web has severe consequences for society, as attackers can exploit these weaknesses to routinely compromise millions of websites, steal personal and financial information, and penetrate private infrastructure. For example, CVE-2021-44228 [14] is a recent and widely known vulnerability (remote code execution) in log4j framework that has a significant impact on enterprise applications. Although the exact market impact of this vulnerability is unclear, the Google security team [27] mentions The ecosystem impact numbers for just log4j-core, as of 19th December are over 17,000 packages affected, which is roughly 4% of the ecosystem.

Software security testing is crucial in addressing security concerns. Developers frequently utilize static code analysis and automated testing tools to examine their applications and uncover vulnerabilities prior to deployment. However, current solutions often have limitations in fully identifying security issues automatically. With the pressure of development costs and time-to-market constraints, companies face challenges in allocating sufficient resources to improve security and ensure proper security measures during the software development phase.

Current methods fail to offer a clear feedback to developers on how to interpret their results. This is especially true for components of the application that are difficult to test or are beyond the scope of the testing tool being used, leaving developers with no insight into the issues encountered or possible solutions to achieve more resilient testing.

The central idea of this thesis is to restructure the traditional secure development life-cycle by prioritizing testability. As web applications grow in size and complexity, there is a need for automated testing techniques to analyze them, but these techniques may struggle to handle the intricacies of complex applications. To break this cycle, the thesis proposes incorporating testability as a key aspect of software development, along with a set of testability patterns. This shift will provide developers and managers with a flexible tool to evaluate the security of complex web applications and deter-

mine the most suitable strategy for mitigating vulnerabilities. For instance, while using a framework to develop an application may appear to reduce the likelihood of vulnerabilities, it may also make the rest of the code harder to test, potentially undermining the security of the entire web application. By assessing the risk associated with introducing the framework, developers and managers can make informed decisions on the best solution for their specific situation.

Our testability-driven development process offers testing tools providers a scientific and measurable approach to evaluating the efficacy of their tools across various technologies, code patterns, and application architectures. This feedback is crucial in identifying the limitations of a given technique and finding ways to enhance it to handle modern technologies. Additionally, it can be used to leverage the strengths of one tool to overcome the weaknesses of another, such as utilizing a static analysis tool in collaboration with another static tool. Finally, by understanding the effectiveness of tools and techniques for a particular application, our methodology can help developers choose the most suitable testing tool that aligns with the application's characteristics, thereby optimizing the effectiveness of security testing.

1.2 Contributions

This thesis makes three separate contributions to addressing testability tarpits in SAST tools. For each contribution, we investigate the limitations of static tools and propose solutions to resolve them and increase source code coverage.

1.2.1 Testability Tarpits

In the first contribution, we address the residual risk posed by challenges encountered by Static Application Security Testing (SAST) tools. Our approach is novel and is based on the concept of "testability tarpits". A tarpit is a code pattern that is known to cause problems for a certain class of SAST tools. The idea is that if an application was easy to analyze, the residual risk of undetected vulnerabilities would be lower, while if it presented many challenges to the analysis tool, the residual risk would be higher.

Our proposed framework, based on a comprehensive library of testability tarpits, enhances our understanding of the results from one or multiple SAST tools. It not only assesses the confidence of the reported results,

but also pinpoint the specific code that diminishes that confidence. This information empowers an analyst to make informed choices, such as adding more tools to overcome the effect of tarpits, manually reviewing code that is hard to test, or restructuring the application to increase its testability.

We analyze PHP and Javascript code using static analysis tools, as they are the two most commonly used languages for web application development. Our methodology is general and applies to any language or class of analysis tools. We have created a library of 122 testability tarpits for PHP and 153 tarpits for JS, covering various language features. We tested several commercial and open-source SAST tools with these tarpits. Although some of the best commercial tools have high accuracy, they could only process 50% of the tarpits, revealing possible gaps in analysis. To determine the impact of unsupported tarpits, we created automated discovery rules for all PHP patterns and scanned 3341 open-source PHP applications. Our experiments showed that these tarpits are widespread in the real world - the average project had 21 different tarpits and even the best SAST tool could not analyze more than 20 consecutive instructions without encountering a pattern that impeded its analysis.

The automated discovery of tarpits offers numerous advantages. One of the most significant benefits is that it gives developers prompt and accurate feedback regarding the tarpits in their code by integrating discovery rules into an IDE. This information can be utilized to determine the most suitable combination of SAST tools for analyzing the code, identify areas that may require a thorough code review, and recognize regions of code that could be restructured for better testability.

This contribution concludes by conducting two experiments to assess the impact of code refactoring on the testability of applications for SAST tools. In the first experiment, we manually examined 10 PHP and JS applications where SAST tools failed to detect known vulnerabilities. By addressing the testability obstacles, the tools were able to identify the vulnerabilities, leading to the discovery of over 200 additional bugs, including 71 confirmed vulnerabilities. In the second experiment, we utilized an automated tool to apply five tarpit transformations to thousands of popular real-world applications. The tool modified 1170 applications, transforming 32,192 instances of the five tarpits. Comparison of the results of SAST scans before and after the transformations revealed a significant improvement in overall testability, with hundreds of previously unknown vulnerabilities detected. We discovered 370 zero-day vulnerabilities in 43 different applications, 55 of which affected popular projects with over 1000 Github stars. All findings were

responsibly disclosed. These results demonstrate the value of our approach and the positive impact of removing tarpits on the testability of SAST tools.

Thus, we assist the open source community in detecting 441 high severity vulnerabilities and publish our findings in a research paper titled "Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications."

1.2.2 Tools collaboration

In the first contribution, we addressed the challenges and limitations of static analysis tools. In the second part of this thesis, we introduce the first approach to facilitate collaboration between SAST tools. Our novel technique only operates on the application source code, making it compatible with both research and commercial SAST tools without requiring access to their internal data structures. Our approach entails searching for evidence of disrupted data flows by utilizing the tools as oracles and then creating a bypass to circumvent the code that the tools were unable to handle correctly.

Our approach is universally applicable and can be used with any programming language. To demonstrate this, we created a fully automated prototype called Whip, which is specifically designed for the PHP language. PHP remains the most widely used language for web application development today.

We conducted experiments to demonstrate the effectiveness of Whip in improving the performance of two popular commercial SAST tools, Comm_2 and Comm_1. Our results showed that Comm_1 and Comm_2 reported 25% and 10% more security alerts respectively, when Whip was used. This resulted in the discovery of 9,226 new high-severity alerts, 35 of which were confirmed as zero-day vulnerabilities across 14 applications. Our manual investigation of 30% of the new alerts confirmed the validity of 24 of these vulnerabilities, which were acknowledged by the respective developers.

Finally, we compared the complexity of the new vulnerabilities found by Whip with a dataset of 100 previous CVEs. Our analysis reveals that using a tool to supplement the limitations of another tool allows both to delve deeper into the target data flow. For example, while previous CVEs had an average vulnerability path of only 7.8 lines of code (LOC), the shortest path among our 35 new discoveries was 12 LOC, with an average of 25 LOC.

This work was published in a paper entitled "Whip: Improving Static Vulnerability Detection in Web Applications by Forcing Tools to Collaborate". In addition to that, this research led to a new OWASP project to build a community around testability tarpits [19].

1.2.3 Testability Tarpits in MVC design pattern

The third contribution focuses on addressing the trend of widespread usage of model-view-controller (MVC) frameworks in modern web applications, which makes the SAST tasks even more difficult. Despite the advantages that MVC provides in terms of simplifying development and maintenance, it also introduces a significant degree of dynamic interaction between the application code and the framework code, making SAST infeasible.

We propose a novel solution to disconnect an application's source code from its MVC framework, making it easier for SAST tools to analyze. Our approach focuses solely on the application source code, making it applicable to both research and commercial SAST tools without needing access to their internal implementations. Our approach consists of two phases: first, identifying problematic flows between the application and the MVC framework, then transforming these dynamic flows into static ones between the application and a transformed version of itself, thereby making them more suitable for SAST analysis.

We conducted two experiments to demonstrate the efficacy of our approach in enhancing the source code coverage analyzed by SAST tools. In both experiments, SAST tools reported a significantly higher number of security alerts. In the first experiment, we applied our approach to 10 known-to-be-vulnerable applications built on popular PHP MVC frameworks (Laravel and CodeIgniter). Our manual transformations enabled SAST to detect the known vulnerabilities. In the second experiment, we automated the transformation process for 20 CodeIgniter projects from Github and Sourcecodester. SAST reported over 2,000 new findings, including 826 previously unknown stored XSS vulnerabilities and 103 reflected XSS vulnerabilities affecting 18 out of 20 projects analyzed. All impacted projects acknowledged our findings through responsible disclosure, and CVEs were released. Finally, we explore this subject in the research paper "Mitigating the Impact of the MVC Design Pattern on Web Application Static Security Testing".

1.2.4 Vulnerabilities Discoveries

With the three contributions, we improve the security of open-source projects by detecting 1,404 high-severity vulnerabilities in 82 projects. Each time we confirmed that an alert was a true positive, we contacted the developers to initiate a responsible disclosure process. In each communication, we described the issue and provided feedback on how to fix the vulnerability. In some cases, we even submitted pull requests on Github containing the

patch. When a project had multiple vulnerabilities, we only requested one CVE to cover all the corresponding cases, resulting in 55 CVEs for our discoveries.

Table 1.1: Our CVEs discoveries

Cont.	Project	Vul.	CVE
Cont. 1	Lychee-v3	XSS	CVE-2021-43675
Cont. 1	Librenms	XSS	CVE-2021-44279
Cont. 1	MantisBT	XSS	CVE-2021-33557
Cont. 1	Matyhtf framework	Path M.	CVE-2021-43676
Cont. 1	Dzzoffice	XSS	CVE-2021-43673
Cont. 1	Vesta	File I.	CVE-2021-43693
Cont. 1	ThinkUp	Path M.	CVE-2021-43674
Cont. 1	Wechat-php-sdk	XSS	CVE-2021-43678
Cont. 1	SakuraPanel	XSS	CVE-2021-43681
Cont. 1	nZEDb	XSS	CVE-2021-43686
Cont. 1	Thinkphp-bjyblog	XSS	CVE-2021-43682
Cont. 1	Pictshare	XSS	CVE-2021-43683
Cont. 1	Chamilo-lms	XSS	CVE-2021-43687
Cont. 1	Fluxbb	XSS	CVE-2021-43677
Cont. 1	Libretime	Path M.	CVE-2021-43685
Cont. 1	IssabelPBX	XSS	CVE-2021-43695
Cont. 1	Twmap	XSS	CVE-2021-43696
Cont. 1	Tripexpress	Path M.	CVE-2021-43691
Cont. 1	YurunProxy	XSS	CVE-2021-43690
Cont. 1	Manage	XSS	CVE-2021-43689
Cont. 1	Youtube-php-mirroring	XSS	CVE-2021-43692
Cont. 1	Workerman-ThinkPHP-Redis	XSS	CVE-2021-43697
Cont. 1	PhpWhois	XSS	CVE-2021-43698
Cont. 1	Attendance management system	SQLI	CVE-2021-44280
Cont. 1	Docsify	XSS	CVE-2021-23342
Cont. 2	Vesta	XSS	CVE-2022-36305
Cont. 2	Jukebox-RFID	XSS	CVE-2022-36749
Cont. 2	ICEcoder	Path M.	CVE-2022-34026
Cont. 2	Dokuwiki	XSS	CVE-2022-28919
Cont. 2	PicUploader	XSS	CVE-2022-41442
Cont. 2	Phoronix	XSS	CVE-2022-40704
Cont. 2	Librenms	XSS	CVE-2022-36746
Cont. 2	Phpipam	XSS	CVE-2022-41443

Cont. 2	Razor	XSS	CVE-2022-36747
Cont. 2	Pfsense	XSS	CVE-2022-42247
Cont. 3	Wscats-cms	XSS	CVE-2023-23016
Cont. 3	Corn-manager	XSS	CVE-2023-23017
Cont. 3	Kalkun	XSS	CVE-2023-23015
Cont. 3	InventorySystem	XSS	CVE-2023-23014
Cont. 3	Hr-payroll	XSS	CVE-2023-23013
Cont. 3	Classroombookings	XSS	CVE-2023-23012
Cont. 3	InvoicePlane	XSS	CVE-2023-23011
Cont. 3	Ecommerce Bootstrap	XSS	CVE-2023-23010
Cont. 3	Sales Management System	XSS	CVE-2023-23018
Cont. 3	Book Store	XSS	CVE-2023-23024
Cont. 3	Expense Management System	XSS	CVE-2023-23027
Cont. 3	Sales Management System	XSS	CVE-2023-23026
Cont. 3	Hotel System	XSS	CVE-2023-23025
Cont. 3	Laundry System	XSS	CVE-2023-23023
Cont. 3	Employees Payroll	XSS	CVE-2023-23022
Cont. 3	Point Of Sale	XSS	CVE-2023-23021
Cont. 3	Blog Site	XSS	CVE-2023-23019

1.3 Thesis Outline

The thesis is structured as follows:

Chapter 2 provides the background information and key concepts necessary to understand the thesis contributions.

Chapter 3, based on the paper "Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications" presented at the Network and Distributed Systems Security Symposium (NDSS 2022), defines a list of testability tarpits, assesses their prevalence in open-source projects, and outlines approaches to eliminate these tarpits and enhance the testability of the projects.

Chapter 4, based on the paper "Whip: Improving Static Vulnerability Detection in Web Applications by Forcing Tools to Collaborate" presented at the Usenix Security Symposium 2023, presents a novel method to encourage collaboration between static tools by sharing information that can help them overcome each other's limitations.

Chapter 5, based on the paper "Mitigating the Impact of the MVC Design Pattern on Web Application Static Security Testing" and under submission to The ACM Joint European Software Engineering Conference

and Symposium on the Foundations of Software Engineering (ESEC/FSE), investigates MVC frameworks and their interaction with application source code. This chapter identifies the challenges faced by SAST in understanding the interactions between the application and framework and provides a solution to "disconnect" the application from the framework.

Finally, Chapter 6 concludes the thesis and suggests potential future research directions.

Chapter 2

Background

This Chapter provides background information to contextualize the thesis. An overview of Web Application Security Testing and solutions for white-box, black-box and test case generation is presented in Section 2.1. Section 2.2 focuses on previous studies that developed static tools, and the selection of tools for our study. In Section 2.3, we discuss different types of injection vulnerabilities, the methods used by attackers to exploit them, and how developers can defend the system.

2.1 Web Application Security Testing

The detection of vulnerabilities in web applications has been a major focus of previous research, leading to various proposed solutions. The chosen technique depends on the information available to the tester and the type of vulnerability being investigated.

In the one hand, white-box solutions rely on source code analysis and examine the data flow between a source (a location where user data is inputted) and a sink (a security-sensitive operation, such as an SQL query) to identify input validation vulnerabilities [104, 94, 152, 66]. White-box solutions can also use model checking (e.g.,[92, 156, 112]) or custom models of application behavior to detect logic flaws (e.g., [71, 51]). Previously, server-side source code analysis was more prevalent, but with the rise of modern web applications, the focus has shifted to the analysis of client-side JavaScript. Various JavaScript analysis techniques have been proposed, including static analysis [55, 79, 80, 81, 109], dynamic analysis [134, 140], hybrid analysis [62, 70, 145, 154], taint analysis [135], and string analysis [41, 42, 54].

On the other hand, black-box solutions for web application security probe the target application without knowledge of its source code. These techniques can discover various classes of vulnerabilities, including traditional input validation threats (e.g., [67, 96, 113, 49]) and more recently, logic vulnerabilities (e.g., [78, 128, 150]). Whereas a classification of black-box testing techniques is still missing, two dominant groups can be distinguished: vulnerability scanners (e.g., Nikto [119] and Nessus [136]) and web application scanners (e.g., [96, 67, 129]). Vulnerability scanners can only detect instances of vulnerabilities (e.g., Nikto [119] and Nessus [136]), while web application scanners detect classes of vulnerabilities by looking for general vulnerable behavior in a web application-agnostic manner [96]. A typical web application scanner has three components: a crawler, a test case generator, and an analysis module. The crawler covers the entire ap-

plication, the test case generator probes the target with specially crafted inputs, and the analysis module processes the result pages to detect vulnerabilities. The precision of these components is crucial for the effectiveness of the black-box solution. However, web scanners are limited in their performance with dynamic URLs and the classes of vulnerabilities they can discover [68, 53]. Recent research has focused on the shortcomings of web scanners but only in the crawler module(e.g., [67, 129]), leaving the detection components out of scope.

The generation of test cases and analysis has been studied in a separate field of research. This line of work has centered on detecting more serious vulnerabilities such as faults in the web application's logic [128, 141] and the user authentication and authorization components [78]. These techniques combined concepts from other areas, including model inference [128, 78], model-based testing [57], and generation of test cases based on attack patterns [128]. Although these approaches have demonstrated that black-box analysis can detect deeper vulnerabilities, they are largely specialized to only a few vulnerability types, so their impact is limited.

Existing security testing techniques are not always effective in detecting vulnerabilities in modern web applications. Code-based techniques struggle with the complexities of popular programming languages such as PHP and JavaScript, leading to unsound analyses that leave some execution paths unexplored [106]. Black-box techniques, meanwhile, lack the ability to detect deeper vulnerabilities in complex applications [68, 53]. This results in limited impact and difficulty in quantifying the effectiveness of current security testing methods.

In this thesis, we address the challenge of limited effectiveness of existing security testing techniques for modern web applications by introducing a collection of "web testability patterns" for static analysis. These patterns have two significant outcomes. Firstly, they accurately identify which features and parts of a web application contribute to the limitations of a testing methodology, providing a quantifiable measure of the testing tool's effectiveness. Secondly, the pattern library acts as a test suite to evaluate various tools and assist their developers in improving their ability to handle current web technologies. The cycle introduced in this thesis will lead to more advanced web application testing techniques and quantifiable results in testing experiments.

2.2 Static Tools

In this section, we will review tools and techniques that can be utilized when the source code of the program being tested is available. These approaches take advantage of the source code artifacts to examine the different behaviors of the component under examination. Thus, we present a list of static tools in different programming languages.

CodeQL. CodeQL [76] is a static semantic code analysis engine that enables analysts to query code as if it were data. It is capable of performing variant analysis and discovering vulnerabilities in codebases. CodeQL is free for research and open-source projects and can be run locally or on projects hosted on GitHub. It comes with a set of standard queries that can be augmented with custom-written ones. Additionally, CodeQL provides data flow analysis for a subset of supported programming languages (C, C++, C#, Go, Java, JavaScript, Python, Ruby and TypeScript).

CPG in binary PHP. The Control-Flow Graph (CPG) as a data structure was introduced by Yamaguchi et al. [158] and is a combination of multiple static analysis graphs, including the Abstract Syntax Tree, Control Flow Graph, and Data Flow Graph. These graphs are extracted from the source code of a program and then combined into one graph by using the nodes that all graphs share, i.e., the CFG nodes. The resulting graph can be used to describe patterns representing vulnerabilities by formulating traversals across different edges in the graph. The CPG generator for PHP converts a given project folder's individual source files into their bytecode representation, which is then translated into CPG form. This resulting CPG is compatible with the ShiftLeft framework, which allows interaction and storage of CPGs in any language [5].

CPG in binary JAVA. Plume is a code property graph generator for Java projects [21]. It transforms individual source files in a given project folder into their bytecode representation or directly analyzes a Java Archive (JAR) that contains class files in bytecode form. This bytecode is then translated into a code property graph (CPG) format. The resulting CPG is compatible with the ShiftLeft framework, which allows interaction and storage of CPGs for any programming language [5].

Joern. Code property graphs (CPGs) can be generated directly from source code instead of binary code, providing access to syntax trees that more accurately reflect the original program code as viewed by developers. The Joern project [143] offers early versions of CPG generators for Java and JavaScript source code, with plans to include support for Python and PHP source code throughout 2022. One of the key benefits of these frontends

is that they can generate CPGs without the need for a configured build environment, enabling the scanning of a large number of projects without the need for manual setup of build systems.

LGTM. LGTM [33] is a commercial code analysis platform that aims to detect zero-day vulnerabilities in source code through the use of CodeQL queries. The vendor provides a free service for scanning open-source projects hosted on public repositories such as Bitbucket Cloud, GitHub.com, and GitLab.com. LGTM supports programming languages including Java, Python, JavaScript, TypeScript, C#, Go, C, and C++. It utilizes a set of standard CodeQL queries, which have been manually curated to minimize false positive results. If a developer detects and fixes a reported issue, they can also extend the query to identify code patterns similar to the original bug. Additionally, LGTM can be configured to perform automated code review on each pull request.

NodeJsScan. NodeJsScan [35] is a static security code scanner for Node.js applications. It offers a user-friendly interface with various security status dashboards. It can be easily utilized through a Docker image and a web page, and the source code to be scanned can be provided as a zip archive. Alternatively, it can be integrated with GitHub, GitLab, and/or Travis projects. The analysis results are displayed in a report window, organized by priority according to the OWASP ranking. Each finding includes information on the file and line where the issue is located. NodeJsScan claims to be more effective in detecting SQL injection vulnerabilities than cross-site scripting vulnerabilities. It supports JavaScript code written for Node.js and can be integrated with GitHub, GitLab, and/or Travis projects. NodeJsScan is designed for detecting vulnerabilities, particularly injection-related, in Node.js applications.

phpSAFE. phpSAFE [123] is an open-source static code analyzer that can identify vulnerabilities in PHP applications. The tool detects the presence of unsanitized data-flow paths between sources and sinks, which are related to both SQL injection (DB query) and XSS (echo and print statement) vulnerabilities. It is specifically designed for the PHP language and can analyze OOP constructs. phpSAFE can also analyze WordPress plugins without the need for a WordPress installation. It is a web application that accepts a PHP file as input and generates a report listing the potential vulnerabilities in the file.

Progpilot. Progpilot [130] is an open-source static analyzer aimed at identifying security issues in PHP applications. It is an actively maintained tool, with its latest version released in March 2020, making it the most

current open-source PHP analysis tool. Progpilot can be installed using a standalone phar, by building from source code, or through the composer. It supports PHP 7.2.5 and can be configured by editing a configuration file that defines sources, sinks, sanitizers, and validators. Progpilot can be run from the command-line or used as a library within a PHP project and supports OOP constructs.

RIPS. RIPS [65] is a popular static code analysis tool designed to detect vulnerabilities automatically in PHP applications. It works by transforming the application source code into a model, which is later analyzed by a taint engine to identify data flows between sensitive sinks and user-input. The open-source version of RIPS supports only PHP, while the commercial version, branded as SonarSource, supports 27 languages. RIPS is a web application where the analyst specifies the path of the PHP project to be analyzed. The tool provides a structured output of discovered vulnerabilities and also offers an integrated code audit interface for reviewing PHP projects and grouping vulnerable code.

WAP. Web Application Protection [151] (WAP) is an open-source project that performs static analysis of PHP applications to detect input validation vulnerabilities. WAP uses taint analysis to identify possible paths between un-sanitized inputs and sensitive sinks, followed by a data mining phase to confirm the generated alerts and reduce false alarms. The last update to the project was in 2017. WAP supports PHP and can detect 8 types of vulnerabilities, including SQL Injection (SQLI), Cross-Site Scripting (XSS), and others. Written in JAVA, WAP consists of three separate modules: a code analyzer that detects vulnerabilities, a classifier that reduces false positives, and a module that provides source code patches to mitigate discovered bugs.

Selected tools. In this thesis, we address challenges and ways to enhance the coverage of static tools for two programming languages: PHP and Javascript. The tools studied include RIPS, phpSAFE, WAP, Progpilot, LGTM, NodeJsScan, and Shiftleft, as well as four anonymous commercial tools.

2.3 Injection vulnerabilities

Fabian Yamaguchi used the definition of a vulnerability from The Internet Security Glossary (IETF RFC 4949) [136, page 333] in his thesis [157]. According to this definition, a vulnerability is a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy, and goes on to state that a

system can have three types of vulnerabilities: (a) vulnerabilities in design or specification; (b) vulnerabilities in implementation; (c) vulnerabilities in operation and management.

OWASP provides a regular list of the top 10 vulnerabilities that represent a broad consensus on the most critical risks in web applications. This list discusses different types of attacks that systems can face, including authentication, privileges, data encryption, injection, configuration, design, and others. Over the past 20 years, injection vulnerabilities have consistently remained among the top three types of vulnerabilities. Injection vulnerabilities occur when an attacker can inject harmful values into an application, which can lead to unexpected results when interpreted by other parts of the system.

To detect these bugs, SAST tools need to reason about the flow of user-provided information through the program, starting from the points where attackers can inject their input (called "*sources*") until the points in the program where this input is consumed and interpreted (called "*sinks*"). The variables which carry the data between sources and sinks are called "*tainted variables*". Thus, detecting injection vulnerabilities boils down to discovering a data-flow path that connects a source to a sink, along which the data is not properly sanitized.

SAST tools require the ability to analyze the flow of user-provided information through a program in order to detect bugs. This analysis must start from the points where attackers can inject their input, also known as "sources," and continue until the input is consumed and interpreted, known as "sinks." The variables that transport the data between sources and sinks are known as "tainted variables". Detection of injection vulnerabilities boils down to discovering a data-flow path that connects a source to a sink, along which the data is not properly sanitized.

We refer to vulnerabilities that have a direct path from source to sink as first-order vulnerabilities. Second-order vulnerabilities (also known as "stored vulnerabilities") are a type of injection vulnerability that occurs when an application stores user input (such as in a database) and later uses that input in a dangerous way without properly sanitizing or validating it. Unlike a traditional injection vulnerability, where an attacker can immediately exploit the vulnerability, a second-order vulnerability requires the attacker to first input malicious code or data into the application and then wait for it to be retrieved and processed in the future, possibly by a different user or a different part of the system. This delay in the attack can make second-order vulnerabilities more difficult to detect and mitigate.

In this section, we will list the most common injection vulnerabilities and provide code examples in PHP. We will also discuss sanitizers that can be used to protect applications from these vulnerabilities.

2.3.1 SQL Injection

SQL injection is a security vulnerability that arises when an attacker is able to inject malicious SQL code into an application that interacts with a database. This can occur when an application incorporates user input into a SQL statement without proper sanitization or validation.

If a victim user submits the compromised input, the malicious SQL code is executed by the application, enabling the attacker to carry out a range of activities. For example, an attacker may be able to retrieve, modify, or delete data from the database.

In this PHP code snippet, the user inserts their user ID, which is used to fetch data related to that ID from the database. For instance, if the user inputs the value 5 as their ID, the resulting query will be "SELECT * FROM users WHERE id=5". However, if attackers insert input like "5 OR 1=1", they can manipulate the SQL query and retrieve data for all users instead of just one, resulting in a query like "SELECT * FROM users WHERE id=5 OR 1=1".

```
$mysqli = new mysqli("localhost", "user", "password", "database");
$user_id = $_GET["p1"];
$query = "SELECT * FROM users WHERE id=$x";
$result = $mysqli->query($query);
var_dump($result);
```

To prevent SQL injection attacks, applications need to properly sanitize and validate user input (e.g. `mysqli_real_escape_string` in PHP), and use parameterized queries or prepared statements to separate the SQL code from the user input.

2.3.2 Cross Site Scripting (XSS)

Cross-site scripting (XSS) is a type of security vulnerability that arises when an attacker is able to inject malicious code, often in the form of a script, into a web page that will be viewed by other users. This type of attack can occur when an application allows user input to be included in a web page without proper sanitization or encoding.

Once the victim user visits the compromised web page, the malicious script is executed by their browser, enabling the attacker to carry out a variety of actions. For example, an attacker may be able to steal the user's

session tokens, gain access to sensitive information, or manipulate the content of the web page in unauthorized ways.

In this example, we receive a request parameter from the GET method that the user inputs, and then we print this input without sanitizing it. This input may include a script that will be executed on the client side, such as a script that reads the cookie and forwards it to another server.

```
$x = $_GET["p1"];
echo $x;
```

In order to prevent cross-site scripting (XSS) attacks, it is important for applications to sanitize and validate user input appropriately (for example, using functions like `htmlspecialchars` in PHP), and to encode any user-generated content before displaying it on a web page. In addition to these measures, web developers can also implement various security measures, including Content Security Policy (CSP), to further protect against XSS attacks. CSP enables website administrators to define which sources of content can be executed on a website, thus providing an additional layer of defense against potential attacks.

2.3.3 Path Manipulation

A path manipulation vulnerability is a type of security vulnerability that arises when an attacker is able to access or manipulate files on a system in unauthorized ways. This can occur due to flaws in the design or implementation of an application, such as insecure file permissions or insufficient input validation, which can allow an attacker to perform actions on files that they should not be able to access or modify.

This type of vulnerability can enable an attacker to read, write, or delete files on a system, potentially leading to data theft, loss of critical data, or even complete system compromise. Examples of path manipulation vulnerabilities include directory traversal attacks, file inclusion vulnerabilities, and file upload vulnerabilities.

In this PHP code, the user inputs a file name, and the corresponding file in the directory "/var/www/files/" is included. For example, if the user inputs "home.php", the resulting file inclusion will be "/var/www/-files/home.php". However, attackers can insert "../../../etc/passwd" to access the password file on the system, resulting in a directory traversal attack where the resulting file inclusion would be "/var/www/files/../../../etc/passwd".

But the attacker can access others files in the system

```
$x = $_GET["p1"];
```

```
include "/var/www/files/".$x;
```

To prevent file manipulation vulnerabilities, applications should implement appropriate access controls, input validation, and file permissions. In PHP, the built-in function `realpath` can be used to return a canonicalized absolute pathname, which can help prevent path traversal attacks. For example, if the input is "../../etc/passwd", the output of `realpath("./../../etc/passwd")` would be "/etc/passwd".

2.3.4 Command Injection

A command injection vulnerability is a security flaw that occurs when an attacker is able to run arbitrary commands on a system by injecting malicious code into an application that interacts with the operating system. This vulnerability can arise when an application allows user input to be included in a command without properly validating or sanitizing it.

Once a victim user submits the compromised input, the operating system executes the malicious code, enabling the attacker to perform a range of actions, including accessing, modifying, or deleting data on the system. As a result, this vulnerability can result in unauthorized access, data theft, or even a complete system compromise.

In PHP, commands can be executed using the built-in function `exec`. For example, the user can insert the argument "-l" to list the directory contents with a long listing format using the command "ls -l". However, an attacker can inject another command by providing the input "-l; rm -rf *", which would execute the command "ls -l; rm -rf *", resulting in the deletion of all files in the current directory.

```
$x = $_GET["p1"];
$output = exec("ls ".$x);
```

Preventing command injection vulnerabilities requires checking for special characters, such as ';', which can allow multiple commands to be executed on the same line. For better security, it is recommended to use a whitelist approach that checks the input under specific conditions. For example, the parameters for the 'ls' command can be limited to '-c', '-g', '-l', and '–version' to prevent unauthorized execution of other commands.

```
$arr = array("-c", "-g", "-l", "--version");
$x = $_GET["p1"];
if($x in $arr){
   $output = exec("ls ".$x);
}
```

Chapter 3

Testability Tarpits: the Impact of Code Patterns on the Security Testing of Web Applications

Preamble

While static application security testing tools (SAST) have many known limitations, the impact of coding style on their ability to discover vulnerabilities remained largely unexplored. To fill this gap, in this study we experimented with a combination of commercial and open source security scanners, and compiled a list of over 270 different code patterns that, when present, impede the ability of state-of-the-art tools to analyze PHP and JavaScript code. By discovering the presence of these patterns during the software development lifecycle, our approach can provide important feedback to developers about the *testability* of their code. It can also help them to better assess the residual risk that the code could still contain vulnerabilities even when static analyzers report no findings. Finally, our approach can also point to alternative ways to transform the code to increase its testability for SAST.

Our experiments show that testability tarpits are very common. For instance, an average PHP application contains over 21 of them and even the best state of art static analysis tools fail to analyze more than 20 consecutive instructions before encountering one of them. To assess the impact of pattern transformations over static analysis findings, we experimented with both manual and automated code transformations designed to replace a subset of patterns with equivalent, but more testable, code. These transformations allowed existing tools to better understand and analyze the applications, and lead to the detection of 440 new potential vulnerabilities in 48 projects. We responsibly disclosed all these issues: 31 projects already answered confirming 182 vulnerabilities. Out of these confirmed issues– that remained previously unknown due to the poor testability of the applications code– there are 38 impacting popular Github projects (>1k stars), such as PHP Dzzoffice (3.3k), JS Docsify (19k), and JS Apexcharts (11k). 25 CVEs have been already published and we have others in-process.

3.1 Introduction

According to the 2020 Edgescan Security Report, "*Web application security is where the majority of risk still resides*" [69]. This is confirmed by the fact that most of the recent data breaches took advantage of the poor security of web applications. From a defensive point of view, there are two main options to detect vulnerabilities in web applications: static application security testing (SAST) and dynamic application security testing (DAST).

Dynamic approaches are sound, but often treat the application as a black box and are therefore severely limited in the number of vulnerabilities they can detect. Static tools can instead reason about the entire behavior of the application, thus potentially detecting more vulnerabilities. However, in practice, they are neither sound nor complete, and often result in very large amounts of false positives.

To mitigate this problem, a large amount of research has been conducted to improve these two numbers, by either proposing techniques to increase the ability of static analysis to discover more vulnerabilities or to reduce the number of false alarms. Despite the progress done in both directions, it is undeniable that SAST tools still struggle to cope with the complexity of real-world code – which is one of the reasons for the poor security of today's web applications.

In this chapter, we look at the problem from a different angle. In particular, we focus on another limitation of these tools that is often neglected: the fact that it is very difficult (independently from the tool's precision) for an analyst to interpret their results. In the example above, how can we translate the lack of vulnerabilities reported by a SAST tool into an actionable insight on the security of the application?
A key observation that motivates our work is that while the precision of the results depends on the tool, the level of "*confidence*" largely depends on the application. For instance, if zero vulnerabilities are reported in a small application with only a handful of untrusted input, the analyst might be confident that the tool was right and the codebase could not contain many undetected vulnerabilities. In contrast, if the same result is returned on a very large and complex application, that confidence might be much lower, therefore resulting in a higher *residual risk* that the code could still contain vulnerabilities.

Our goal is to find a way to capture this residual risk by proposing a novel approach based on the concept of **testability tarpits**. A tarpit is a specific pattern of code that is known to cause problems for a class of static analysis tools. Other researchers have reported such patterns as a way to point out the current limitations (and possible venues for improvement) of static analysis tools. We propose instead to use them as a metric to capture *how testable* an application is. The intuition is that the residual risk of undiscovered vulnerabilities is lower if the application was easy to analyze, and higher if it presented many challenges for the analysis tool.

By building upon a comprehensive library of testability tarpits, we propose a general framework that can support a more principled understanding

of the results of one (or a composition of) SAST tool(s). Our approach does not only provide a way to assess the confidence of the reported results, but it also points to the precise nature and location of the code that reduces this confidence. As a result, an analyst can decide to add more tools to reduce the impact of the existing tarpits, to perform a manual audit of a poorly testable part of the code, or to refactor part of the application to increase its testability.

While the methodology we present is general and independent of the language of the application and the class of tools used to analyze it, in this paper we focus on static analysis tools for PHP and Javascript (JS, in short) code, the two most common languages for web application development. In particular, we create a *library* of 122 testability tarpits for PHP and 153 tarpits for JS (cf. 3.3.1), covering language features, built-in APIs, security-related functionalities, and static and dynamic operations. We then selected an *arsenal* of 11 commercial and open-source SAST tools (6 for PHP and 5 for JS) and we assessed them against our tarpits' libraries (cf. Section 3.3.2). The best commercial tools were only able to handle 50% of the PHP and 60% of the JS tarpits, thus potentially leaving large parts of an application code unexplored. To measure the impact on those unsupported tarpits, we implemented automated *discovery rules* (cf. Section 3.4) for all our PHP patterns and used them to scan 3341 open-source PHP applications. Our experiments (cf. Section 3.5) demonstrate that these tarpits are very common in the real world: the average project contains 21 different tarpits and even the best SAST tool cannot process more than 20 consecutive instructions without encountering a pattern that prevents it from correctly analyzing the code.

The ability to automatically discover each tarpit brings many benefits. First, it can provide immediate and precise *feedback to the developers* about the tarpits in their code (e.g., by integrating the discovery rules into an IDE). This information can then be used to make an informed decision about which combination of SAST tools are better suited to analyze the code, which parts of the application are *blind spots* for a static analyzer and thus may require a more extensive code review process, and which region of code could be *refactored* into *more testable* alternatives.

We conclude our study by performing two experiments to assess the use of code refactoring as a mean to make an application more testable for SAST tools. In the first (Section 3.6), we *manually investigate* five PHP and five JS applications, for which SAST tools were unable to discover the presence of known vulnerabilities. By transforming the testability tarpits

```php
// FILE: core/gpc_api.php
function gpc_get( $name, ..) {
 if( isset( $_POST[$name] ) ) {
  $r = gpc_strip_slashes( $_POST[$name] );
 }
 ...
 return $r;
}

function gpc_get_string($name, ..) {
 $args = func_get_args();
 $r = call_user_func_array('gpc_get', $args);
 ...
 return $r;
}

// FILE: bug_actiongroup_ext.php
$act = gpc_get_string('action');
$act_file = 'bug_actiongroup_' . $act . '_inc.php';
require_once(.. . $act_file);
```

Listing 3.1: Example of a file injection vulnerability in MantisBT

we enabled the tools to detect the vulnerabilities. Moreover, over 200 additional bugs were reported, leading us to the disclosure of 71 confirmed vulnerabilities, as some of the discovered issues still applied to the latest version of the tested projects. In the second experiment (Section 3.7), we target instead thousands of popular real-world applications (the same we used for the prevalence experiment), to which we apply five pattern transformations in a fully *automated* fashion. Our tool modified 1170 applications, by transforming 32,192 occurrences of the five tarpits. By running SAST tools both before and after the transformations we could observe the improvement in the overall testability, supported by the detection of hundreds of previously unknown vulnerabilities. In particular, we discovered 370 vulnerabilities in 43 different applications, 55 of which affected very popular projects with more than 1000 stars in Github. We responsibly disclosed all issues, and we have received 111 confirmation from the development teams (36 confirmations for the popular projects). These outcomes confirm the added-value of our approach and the impact of removing tarpits to increase testability for SAST tools.

All the testability patterns and the resources of this research (for both PHP and JS) are available in our repository [34].

The source-code excerpt shown in Listing 3.1, simplified for presentation, highlights a file injection vulnerability (CVE-2011-3357) in the popular

Mantis Bug Tracker. The code uses the function `require_once` to include and evaluate code from an external file (line 20). Care must be taken to ensure that users cannot freely choose the name of the file as this would permit them to execute arbitrary code. Unfortunately, in the example, the file name ultimately depends on the value of an unsanitized POST request-parameter, a value that an attacker fully controls. The example illustrates the complex interprocedural assignment chains that a static analyzer must correctly handle in order to identify the vulnerability: the file name depends on the variable `$act` that is initialized via a call to the application-defined utility function `gpc_get_string` (line 18-19). This function internally makes use of the PHP functions `func_get_args` and `call_user_func_array` to dynamically invoke the variadic function `gpc_get` (line 12). Finally, `gpc_get` accesses the attacker-controlled POST parameter (line 3-5).

The majority of SAST tools in our selection (see Section 3.3) are not able to detect this vulnerability. Through a manual investigation, we discovered that the dynamic function invocation (`call_user_func_array`) prevented them to connect the user-provided parameter to the name of the included file. In addition, some tools are also unable to handle the `func_get_args` function, which again affects the data-flow of the application.[1] Our finding is confirmed by the fact that a simple refactoring of line 12 into the equivalent `$r = gpc_get($args)` is sufficient to enable the tools to report the file injection vulnerability.

3.2 Approach Overview

The main goal of our research is to build upon this observation and use the concepts of *testability tarpits* to build a new framework to assess the security of web applications. Our approach, outlined in Figure 3.1, is composed of three phases.

Pattern creation. The first objective of our work is to compile a comprehensive list of testability tarpits, that is, code patterns that impede the ability of SAST tools to reason about the code and identify vulnerabilities. The creation and selection process, which we describe in detail in Section 3.3, involved an extensive manual effort. To show the applicability of our approach to different programming languages, we reviewed the documentation, the internal specifications, and the APIs of both PHP and JS and distilled this information into a number of code snippets that emphasize different functionalities. We then embedded these patterns in small

[1] Note that both functions are not library APIs, but core features of PHP.

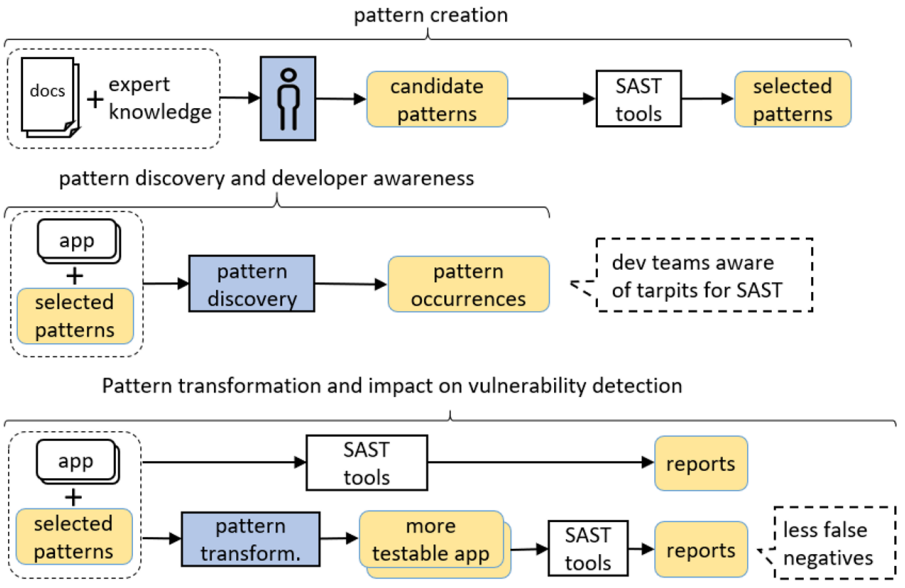

Figure 3.1: Approach outline

test cases, which we tested on a set of commercial and open sources SAST tools (5 for JS and 6 for PHP) to identify the tarpits that could impede the testability of an application.

Pattern discovery and developer awareness. In the second phase of our approach, we implemented a tool to identify instances of our patterns in the source code of a program. As we strongly believe that extending existing production-quality tools trumps re-implementing basic code analysis from scratch, we base our tool on the code analysis platform Joern [143]. The platform allows code patterns to be formulated in a domain-specific query language that provides access to syntax, control flow, and data flow properties. While the platform offers built-in patterns for the discovery of vulnerabilities in C/C++ code, patterns for Web applications are only scarcely available and focus entirely on the discovery of vulnerabilities. Our work performed several improvements to Joern's core analysis engine and PHP support and we developed queries to allow the discovery of testability problems. Both queries and improvements were contributed back to the project, enabling Joern to discover not only vulnerabilities but also testability problems.

Our tool can discover the presence of testability tarpits and make devel-

opers aware of the exact snippets of code that will confuse the SAST tools in their arsenal. While until today developers were only aware that a given SAST tool did not discover any vulnerability in their code, our approach provides additional information that can be integrated into a risk assessment methodology or used, for instance, to select other SAST tools that are more suitable for a specific application. To evaluate our tool and measure the overall prevalence of our library of tarpits in real world applications, in Section 3.4 we present the results of the experiments we conducted on 3341 PHP applications.

Pattern transformation and impact on vulnerability discovery. While awareness is very important, in the final phase of our process we discuss how the identified tarpits can be removed by transforming the corresponding code. Our goal is to show that by transforming the code, the testability improves and SAST tools become capable to uncover more vulnerabilities. We evaluate this idea by applying different types of transformation rules in two separate experiments. In the first (Section 3.6), we manually investigated ten applications (five PHP and five JS) for which SAST tools were unable to discover known vulnerabilities. In this case, we progressively refactored all testability tarpits until the tools were able to detect the bugs. By making the applications more testable, this also allowed the tools to discover new, previously-unknown vulnerabilities. In fact, on the refactored code of the ten projects, the SAST tools in our arsenal reported 503 new alerts, 224 of which corresponded to true-positive vulnerabilities. In the second experiment (Section 3.7), we targeted instead thousands of popular real-world PHP applications, to which we apply five pattern transformations in a fully automated fashion. By running SAST tools both before and after the refactoring we could observe the improvement in the overall testability, supported by the detection of hundreds of previously unknown vulnerabilities. In particular, we discovered 370 security issues in 43 different applications, with 55 of these vulnerabilities affecting popular projects with more than 1000 stars in Github.

3.3 Pattern creation and selection

The first step of our approach consists of identifying those code patterns that prevent SAST tools from properly analyzing an application code. It is important to stress that in this chapter we are only interested in *code-related* patterns, and not in other forms of architectural or deployment aspects that can affect testability. For instance, plugin-based infrastructures

are often difficult to analyze statically, and tools often struggle to deal with application state maintained across requests (which could, for instance, lead to stored vulnerabilities). While these are also very important aspects, and they constitute possible venues to extend our work, in the rest of the chapter we restrict our focus to source code patterns only. Because of this, while the approach we present is completely generic, the actual patterns we identify vary from one programming language to another. To show the generality of our approach, we performed our study on the two most popular programming languages for web application development: JS (ES11) and PHP (v7.4.9).

On top of their specificity for different languages, testability tarpits may also differ from one SAST tool to another. In fact, what constitutes a problem for a testing tool may be handled correctly by a different product, and vice versa. Moreover, developers often use a combination of tools to test their applications, thus making the final selection of patterns specific to each development and testing environment.

Therefore, we first selected a representative set of SAST tools that includes both commercial and open source solutions. In particular, based on the results reported by other studies that compared existing tools [48, 38], for PHP we selected RIPS [65], PHPsafe [123], WAP [151], and Progpilot [130] as representative of open-source solutions, and two leading commercial products referred here as Comm_1 and Comm_2.[2] For JS we selected instead three commercial tools (Comm_1, Comm_2, and Comm_3)[3], and NodeJsScan [35] and LGTM [33] as open source alternatives.[4] While this choice reflects the current state-of-the-art techniques used to analyze PHP and JS applications, our approach can be easily applied as-is to any other set of static analysis tools.

In the rest of this section, we discuss how we built our library of testability patterns (Section 3.3.1) and present the experiments we conducted to validate them on our SAST toolset (Section 3.3.2).

3.3.1 Pattern Creation

Given a programming language, we want to identify code patterns that can affect the ability of SAST tools to detect vulnerabilities. To this end, it is

[2]For legal reasons, we have to anonymize commercial products.

[3]Comm_1 and Comm_2 are the same commercial tools used for PHP.

[4]Notice that LGTM is subject to a commercial license when used on projects that are not open source. This was not the case for our study.

important to understand how static analyzers detect web vulnerabilities in the first place.

While many techniques exist, a common requirement is the ability to identify how user-provided input is propagated and processed by the application. Static taint analysis provides this capability and is employed in particular to uncover injection vulnerabilities such as SQL injections (SQLi), cross-site scripting (XSS), and code injections (CODEi). While the actual mechanism used for vulnerability detection is orthogonal to our approach, this observation provides us with a simple way to identify patterns.

The idea is to use a small fragment of code, hereinafter the *stub*, designed to receive an input value from the user (in the form of a GET parameter) and simply write it back to the output.

```php
$a = $_GET["p1"];
echo $a;
```

```javascript
const parsed = route.parse(req.url);
const query  = querystring.parse(parsed.query);
var c = query.name;
r.writeHead(200,{"Content-Type":"text/html"});
r.write(c);
r.end();
```

These two snippets (respectively for PHP and JS) contain a very simple form of reflected XSS that is correctly identified as vulnerable by every SAST tool on the market. To create our patterns we routinely customized these stubs by adding new operations – which represent our candidate tarpits – as part of the data flow between the source (the GET parameter) and the sink (the `echo` and `write` invocations).

For pattern creation, we systematically inspected all the chapters of the PHP and JS language documentations. We also analyzed comments describing special examples and corner cases and reviewed all the internal language APIs. For PHP, we also went through the entire instruction set of its intermediate language and verified that there was not a single opcode that was not covered by one of our patterns. By following this procedure, we manually created hundreds of test cases, each one dedicated to showcase a different aspect of the language. However, most of these snippets have no impact on the analysis performed by SAST tools. Therefore, we filtered the list by retaining only the problematic examples. For this purpose, we used our selection of SAST tools as oracles. Each tool was used to scan all candidate snippets, and for each test we verified whether the tool was still able to detect the reflected XSS vulnerability. To be conservative,

if *at least one* of the tools failed to report the vulnerability, we saved the corresponding test case in our testability tarpits library. Hereafter we detail the five main dimensions we used to categorize our patterns and to guide our pattern-generation process.

Core language features vs built-in internal APIs

We started our investigation by studying the language documentation and, for PHP, the (often undocumented) list of internal opcodes (i.e., the low-level instructions that are processed by the Zend engine). For example, while *references* are a common concept present in most programming languages, under the hood PHP operates on them by using seven different opcodes (e.g., `ASSIGN_REF` creates a reference to a scalar variable and `RETURN_BY_REF` returns a reference from a function).

For instance, we integrated the `RETURN_BY_REF` opcode [5] in our stub by producing the following snippet:

```
1 class foo {
2  public $v = 42;
3  public function &getV() {return $this->v;}
4 }
5 $a = $_GET["p1"]; $obj = new foo;
6 $myV = &$obj->getV(); $obj->v = $a;
7 echo $myV;
```

In line 7, variable `$myV$` gets bound to the variable `obj->v` returned from `getV`. As a consequence, setting `obj->v` to the source `$a` in line 8 makes also `$myV$` changing, leading to a XSS in line 9.

In total we identified 96 challenging patterns for PHP and 153 for JS. However not all of our patterns are related to features of the language, some are instead associated with the use of core library functions. In fact, internal API functions are typically written in C for better performance, and therefore it is difficult for SAST tools to check their code during the analysis. To mitigate this problem SAST tools often maintain a set of models that describe the relationship (in terms of taint propagation) among the input and output parameters of these functions [65]. For instance, the code snippet below captures the testability pattern created for *extract*, an internal PHP API generating variables dynamically from an array:

```
1 $aaa = $_GET["p1"];
2 $arr = array("A"=>$aaa,"B"=>"BBB");
```

[5] PHP documentation - Return By Reference: `https://www.php.net/manual/en/language.references.return.php`

```
3 extract($arr); echo $A . $B;
```

In this specific case, variables `$A` and `$B` are created and assigned to `$aaa` and to `"BBB"`, respectively. This pattern enables evaluating whether a SAST tool models the extract function and properly propagates the user-controlled input `$aaa` into the variable `$A`. In total, we identified 26 patterns in PHP and 22 patterns in JS that target challenging internal APIs.

Security related

We already mentioned that SAST tools often employ taint-based dataflow analysis to detect vulnerabilities: when a user-controlled input (referred to as *source*) flows into a sensitive operation (referred to as *sink*), without being processed by a *sanitizer*, that dataflow is reported as a vulnerability. We thus created some testability patterns to probe how good the SAST tool is in recognizing sources, sinks, and sanitizers. For instance, the following code snipped captures a pattern instance evaluating whether a SAST tool supports the sink PHP operation `exit`, which terminates the application execution and passes a message to the user:

```
1 $a = $_GET["p1"];
2 exit($a);
```

Similarly to the `echo` sink, `exit` can lead to XSS vulnerabilities. In total, we identified 16 patterns in PHP and 22 in JS in this category.

Static vs Dynamic features

Developers often rely on code constructs that cannot be fully analyzed statically, because their exact behavior can only be determined at runtime. In some cases, this might be required by the application and therefore it might be difficult to rewrite the code in a different way. But in other cases, these are used just for convenience, as the result of cut&paste operations, or to support functionalities that have already been removed from the code long before. For instance, the motivating example we discussed in the previous section uses the dynamic operation `call_user_func_array`, a form of *dynamic dispatching*. However, the MantisBT developers hardcoded a constant parameter, thus making the target function resolvable from a static analysis perspective.

To capture these differences, we define four dynamic categories for our code snippets targeting dynamic operations:

D1: the core parameter of the dynamic operation (e.g., the first parameter in `call_user_func_array`) is an hardcoded constant.

```
1 /* D1 */ call_user_func_array("Func", $b);
```

D2: the parameter is an expression whose value can be univocally computed statically via constant propagation.

```
1 $a = "FuncA";
2 /* D2 */ call_user_func_array($a, $b);
```

D3: the parameter is an expression whose value can only be partially computed statically. E.g., in the following example, only functions starting with `"Func"` can be called. This could be used by SAST to reduce the over-approximation needed to cover all the possible execution paths.

```
1 /* D3 */ call_user_func_array("Func" . $v, $b);
```

D4: the parameter is an expression whose value cannot be computed statically. While in the general case it is not possible to handle code belonging to this category statically, it is still important to measure the prevalence of these patterns to assess the testability of the code.

```
1 /* D4 */ call_user_func_array($f, $b);
```

Evaluating SAST tools against these four dynamic categories of increasing complexity allows us to measure more precisely their behavior against these challenging dynamic operations. 52 of our PHP patterns and 52 of our JS patterns involves dynamic features.

Positive vs Negative Test Cases

To deal with dynamic features that cannot be computed statically, SAST tools need to choose between two possible approaches: over-approximate (e.g., by assuming all elements in an array are tainted when one of them is), or under-approximate (e.g., by ignoring all elements altogether). The first can increase the number of false positives, while the second solution can miss real vulnerabilities. To better distinguish among the two cases, we complemented the tests we developed for dynamic features with special tests that used the same dynamic functionality but without a vulnerability.

For instance, the following JS pattern (related to arithmetic operations on an array index) was reported as vulnerable by Comm_2 but not by Comm_1.

```
1 const parsed = route.parse(req.url);
2 const query  = querystring.parse(parsed.query);
3 var c = query.name;
4 array = ['a', 'b', c, 'd'];
5 index = 3; index = index -1 ;
6 r.writeHead(200,{"Content-Type":"text/html"});
7 r.write(array[index]); // print c variable
8 r.end();
```

This could be due to the fact that Comm_2 can compute the index value statically, or it could simply be the consequence of the fact that the two SAST tools might adopt different strategies to deal with arrays (over-approximating the first and under-approximating the other). To answer this question, we created a negative version of the same pattern, where line 7 is replaced by:

```
7 res.write(array[index-1]); // print 'b' char
```

Since Comm_2 reports a vulnerability also for the negative version of the pattern, we can conclude that it was indeed applying an over-approximation to deal with the array. In total, we retained 7 negative pattern instances for PHP and 20 for JS, for which the negative version was still reported as vulnerable by some of the SAST tools, indicating an excessive over-approximation.

Functional vs Object-Oriented

As we discuss in the related work (Section 5.8) several studies conducted by the software engineering community have discussed the poor testability of object-oriented code. In general, researchers have found that when developers use more object-oriented features, projects become harder to test (even though this in the literature normally refers to dynamic testing). Therefore we included 39 PHP and 40 JS patterns related to classes, methods, static methods, and properties. These patterns cover different OO aspects, such as object constructors, encapsulation, overriding, and inheritance.

3.3.2 Pattern Selection

Figure 3.2 and 3.3 summarize the results of our SAST tools against our libraries of 122 PHP and 153 JS tarpits. Due to space limitation, the complete list of patterns is presented in Appendix and is detailed in our public repository [34].

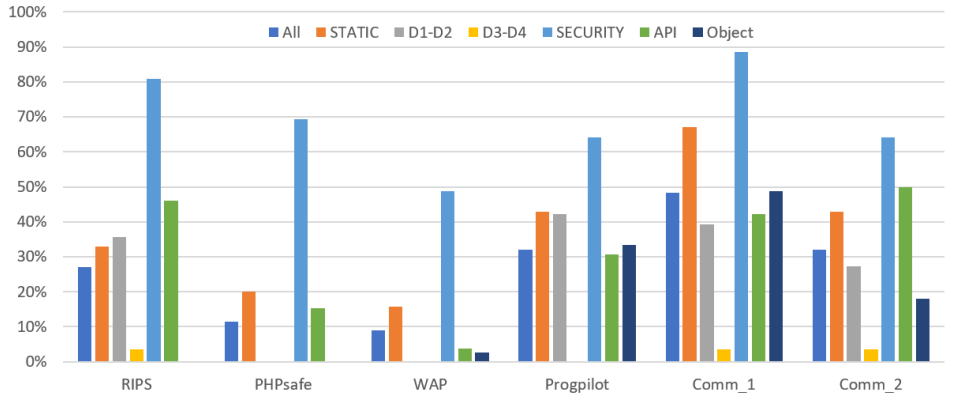

Figure 3.2: SAST measurement over pattern dimensions (PHP)

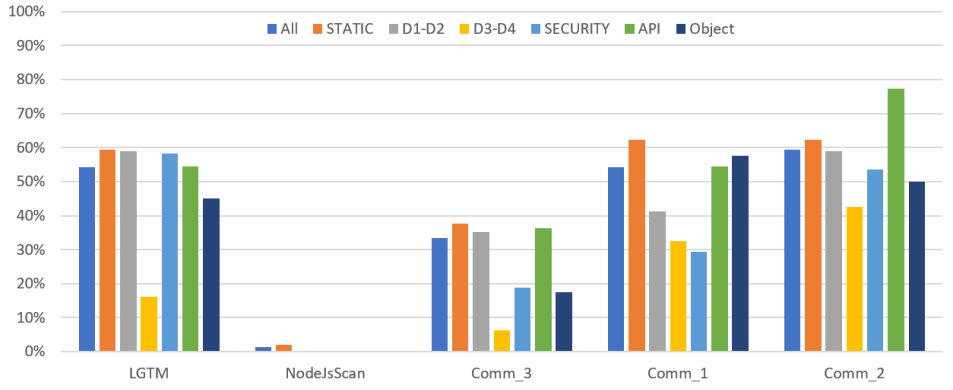

Figure 3.3: SAST measurement over pattern dimensions (JS)

In the graphs, the bars show the percentage of all patterns on which each tool reported the correct answer. First of all, it is interesting to observe that for PHP, none of the tools reaches 50% coverage of our patterns. Comm_1 is the best in terms of overall results, driven by its extensive coverage of PHP static features. However, other tools take the lead in other categories. For instance, Progpilot has the best support for the `D1-D2` tests (where more sophisticated static analysis algorithms might be required to propagate constant and determines the values of program variables), and Comm_2 has the highest coverage of the PHP APIs. RIPS is the best tool among the research tools for the supported APIs, confirmed by the fact that its authors mentioned in the corresponding paper [65] that they performed an extensive

work to model the built-in function of PHP. On the other hand, Progpilot is the only research tool that supports object-oriented code, and it achieves the best results on static features between the open-source tools.

Results are a bit better for JS, where three tools are able to cover more than 50% of the patterns. However, in this case, the higher success rate is in part due to the extremely poor performance of NodeJsScan. Indeed, 24 patterns are supported by all tools but NodeJsScan, and therefore they would have been discarded if this tool was not part of our arsenal. As for PHP, commercial tools are still dominating, featuring similar performances and emerging as the best tool for the security dimension. If Comm_1 performs better on static features, Comm_2 outperforms it on the internal API support. With respect to dynamic patterns, we observe the same behavior as for PHP: tools supports some D1-D2 patterns, but have more troubles against D3-D4 patterns. Only Comm_2 is able to correctly analyze beyond 40% of those D3-D4 instances, though it seems to apply over-approximation in most of those cases.

Another interesting aspect of our experiments is the fact that the tarpits are very different among the different tools. In fact, while tools taken individually have many limitations, their combination is able to handle around 66% of the PHP and even 85% of the JS patterns in our library. Moreover, none of the tools is a superset of any other in terms of tarpits. Thus, using a combination of SAST tools to test a web application is, from a testability point of view, always better than relying on a single product.

3.4 Pattern discovery

In the previous section, we described how we built our library containing hundreds of testability tarpits. While this list can already be used to compare SAST tools and find a suitable combination that minimizes their limitations, this is not the main goal of our paper. Our objective is to support developers to assess which parts of their code can be effectively tested by SAST tools and which are not, and provide guidelines to improve the overall testability by avoiding particular patterns.

For this purpose, we decided to extend each tarpit in our library with a corresponding *discovery rule* that can be executed to discover its presence in the code of a Web application. While `grep`-like regular expressions can be sufficient to identify simple patterns, others require a more sophisticated taint-based analysis that takes into account the interplay of multiple program statements, e.g., to identify where a variable that is used in an inter-

esting operation is previously defined. Designing a complete static analysis framework is beyond the scope of this Chapter, and therefore we decided to build our solution by extending the Joern code querying framework [143]. Joern constructs code property graphs (CPG, [158]) that combine the program's syntax tree, control flow, data dependencies, and calling relations in a joint representation – thus already providing most of the information needed for our analysis. While initially developed for the analysis of C/C++ code, the framework has recently been extended to handle PHP opcodes.[6] However, at the time of writing there was no public CPG generator for Node.js compatible with Joern, and this prevented us from performing a complete analysis of the prevalence of our JS patterns. Nevertheless, we scripted some ad-hoc routines to discover 54 of our JS patterns from the Abstract Syntax Tree (AST) of a JS application. Though these routines are not as precise as CPG queries and suffer from false positives, they have proved to be very helpful in pursuing initial experiments on real applications (e.g., to identify patterns before the manual transformation in Section 3.6).

Our Joern-based discovery rules have different complexities, ranging from a simple search for a given instruction to complex queries where we use the control, data, and call graph dependency. For instance, we use the following query to count the occurrences of the feature *simple reference* (`ASSIGN_REF`) in the CPG of a target application.

```
cpg.call(".*ASSIGN_REF.*").size
```

To find instead the use of objects in which the developer redefines the `__set` function, we can use Joern to search for the `NEW` opcode used on a class that has the method `__set` defined:

```
def hasSet = cpg.typeDecl
  .filter(_.method.name.contains("__set"))
  .name.l
cpg.call("NEW")
 .argument
 .filter{ x => hasSet.contains(x.code.toLowerCase)}.size
```

The ability to automatically discover each tarpit brings many benefits. For instance, these rules can be integrated into an IDE to provide immediate and precise feedback to the developers about the impact of the code they are writing on the static testability of the application. This information can be used to make an informed decision about which parts of the application

[6] The extension is in an early development phase and yet to be released publicly. It was kindly made available to us by its developer—name removed for anonymity.

are *blind spots* for a static analyzer and thus may require a more extensive code review process, and which code patterns could be refactored into more analyzable alternatives. For instance, for security-critical services, developers may decide to limit the use of patterns that are not supported by SAST tools (e.g., those in the category D3-D4) to minimize the risk of undiscovered vulnerabilities.

3.5 Prevalence

We now estimate how prevalent our tarpits are in real-world applications. For this purpose, we use our CPG queries for PHP against four different datasets. The first three are composed of PHP projects hosted on GitHub, chosen according to their popularity (measured by the number of stars they received). In particular, we cloned 1000 applications of low popularity (between 20 and 70 stars), 1000 of medium popularity (between 200 and 700 stars), and 1000 with high popularity (more than 1000 stars).[7] We refer to the three datasets as G_L, G_M, G_H respectively. The detailed list of the cloned projects is available in our repository [34]. Finally, the fourth dataset consists of all applications from the Sourcecodester website (SC in brief), which hosts open-source PHP projects [153] that serve as references to other developers that want to implement their websites.

Due to space limitation, the complete results of our prevalence measurement are reported in Appendix (Table 1). The whole experiment required 1 week on a 16-cores machine with 64 GB of memory. On average, for 1000 lines of code (LOC), generating the CPG took 4.03 seconds, while traversing the CPG with our pattern discovery queries took 7.82 seconds.

The scatter plots in Figures 3.4 and 3.5 compactly present our results. In these plots, each dot represents a PHP application and its coordinates show the number of unique tarpits it contains (Y-axis) and the cumulative number of instances of such patterns normalized by the number of opcodes in the application (X-axis, plotted in log scale). We use the number of opcodes instead of the number of lines of code because it is more accurate as a line of code can contain multiple instructions and because opcodes represent only PHP code without considering other embedded elements such as HTML and JS. It is also important to note that Figure 3.4 does not show four tarpits: simple object (P21), simple array (P58), conditional assignment (P4) and combined operator (P5). In fact, these patterns occur with a very high frequency but only affects a few of our SAST tools.

[7] We used Github-clone-all tool [11] to clone all these projects.

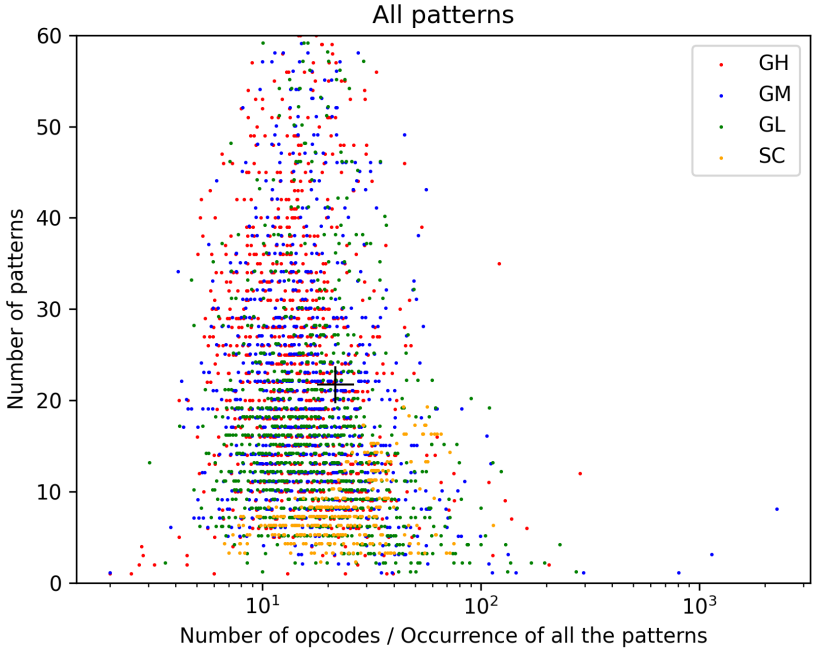

Figure 3.4: Patterns distribution considering all patterns

Figure 3.4 shows the breakdown of the results by datasets. Here we can notice two important points. First, the prevalence of our patterns is very high: the average project contains 21.15 unique tarpits with an aggregated frequency of one tarpit every 8-to-50 opcode (the + sign in the figure represents the average point). Second, the more popular a project is, the more tarpits it tends to contain. The main reason is the size of the project that is higher for more popular projects: by counting the average number of opcodes in G_H, G_M, G_L, and, SC we have $62,303$, $43,782$, $21,066$, and $17,141$, respectively.

If we restrict our analysis to only those patterns that affect a given SAST tool, we observe marginal improvements. For instance, Figure 3.5 shows the results restricted to patterns that affect Comm_2 and Comm_1 for the three Github datasets. It is interesting to observe how the dots clearly follow different distributions, with blue dots closer to the upper-left corner and red dots closer to the bottom-right. This shows that real-world PHP applications are more difficult to test with the first tool than with the

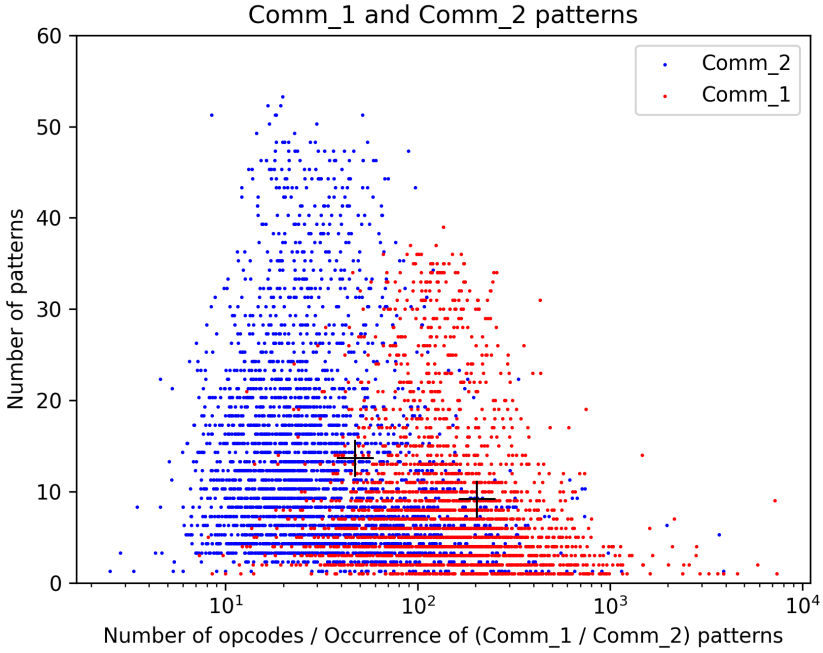

Figure 3.5: Patterns distribution considering only patterns hard for Comm_1 and Comm_2

second: the average application contains 13.3 unique tarpits for Comm_2 (one every 47 opcodes) and 8.5 unique tarpits for Comm_1 (one every 203 opcodes).

Another interesting point is related to the scale of the Y-axis. For over 83% of the applications, the number of tarpits falls between one every 10 and one every 1000 opcodes – with an average of only 21. In other words, on real-world PHP applications even state-of-the-art SAST tools cannot process more than 20 consecutive opcodes without encountering a pattern that prevents them from correctly analyzing the code.

In the rest of this section we will discuss the prevalence of different types of patterns. To streamline the discussion, we only report numbers for the G_H dataset, but all values are available in the appendix for further analysis.

Object Oriented Code - 97% of the projects in the G_H dataset use objects. However, only one out of four of the open source SAST tools we tested supports object oriented code – showing a clear disconnection

between research prototypes and real-world applications. By looking at OOP features that are not supported also by commercial tools, the magic methods `__set` (P32), `__get` (P33), and `__call` (P36) are used respectively by 6.1%, 8.1%, and 6.8% of the projects. This confirms that object oriented code still presents challenges also for the best SAST tools on the market.

PHP Features and APIs - If we exclude OOP, among all our tarpits the most prevalent language feature is the use of combined operators, which is present in 93.4% of the projects. However, this only causes problems to PHPSafe and is correctly handled by all other tools. If we look instead at those features that are causing problems for all our SAST tools, the most common are the use of static variables (71.2% of the projects) and raised exceptions (81%).

Regarding the built-in API functions, `array_map`, which is not supported by any of our SAST tools, is present in 28% of the projects. Among the security critical sources and sinks, the use of superglobals (P66) and the `exit` function (P56) are the most common, with a prevalence of respectively 41.3% and 22.6% projects.

Dynamic Features - the dynamic features of a language are one of the biggest challenges for static analysis tools but also one of the most commonly used by developers. The simple case of storing and retrieving variables from dictionaries (P83), which often requires over-approximation from a static perspective, is used in 88.3% of the projects with a median of 32 occurrences each.

Features related to dynamic functions invocation are also among the most frequently used in our datasets. For instance, storing the function name in a variable (P82) is used in 448 projects, dynamic callbacks (P80) in 269 projects, and dynamic function call (P76) in 602 projects. All these examples belong to the D4 category – and therefore there is little a static tool can do to properly resolve these calls statically.

However, some of the dynamic patterns can be trivially handled by a SAST tool. For instance, the D1 instance of a dynamic call (P80), which hard-codes the name of the target to invoke, is present in 20.8% of the projects with a median of 2.5 instances each. Another simple example comes from file inclusion. In PHP, when file A includes file B, it can access the variables of B without defining them as global variables. The corresponding D1 pattern (P79) causes problems for three of our SAST tools (phpSAFE, WAP, and Comm_2) and it is present in 63.6% of the projects. The related and more complicated case where the file name is stored in a variable (D4), is even more common, with a prevalence of over 75%.

Table 3.1: In-depth analysis on PHP real cases

Project	CVE	Vuln.	Patterns	SAST After (TP/FP) XSS	SQLi	FILEi	CODEi	SUM
MantisBT	CVE-2011-3357	FILEi	callback_functions D1 (P80, T1)	28/88	0/0	18/14	3/0	49/102
Osclass	CVE-2012-0974	XSS	dynamic include D2 (P79, T1), array variable key D2 (P83, T2), static instance of a class (P49, T2)	14/0	0/0	0/0	0/0	14/0
Bakeshop Ordering	CVE-2020-35272	XSS	JS redirect (P57, T2)	15/0	0/0	0/0	0/0	15/0
Bus Booking	CVE-2020-25273	SQLi	Extract function (P70, T3)	0/0	49/0	0/0	0/0	49/0
Domainmod	CVE-2018-11404	XSS	Dirname function (P74, T1), Dynamic include D1 (P79, T1), buffer (P75, T1)	77/118	0/0	0/0	0/0	77/118
			9 patterns	134/206	49/0	18/14	3/0	204/220

3.6 Experiment: Manual pattern transformation

In this section, we will demonstrate the usage of our patterns on five PHP and five JS open-source projects selected according to these criteria: (i) the project is mainly based on PHP or JS, (ii) an injection vulnerability has been reported in the past for the project as a CVE precisely pointing to the latest vulnerable codebase version (*VulCode*, in short), (iii) some testability patterns occur in the path connecting the source and the sink of the vulnerability in *VulCode*, and (iv) one of the SAST commercial tools fails to report the CVE vulnerability on *VulCode*.

We gathered five projects for each language satisfying these criteria from the CVE Mitre website [117]. For each project, we carefully review the CVE and the testability patterns preventing the SAST tool from detecting the expected vulnerability. We then manually transformed these patterns and run again the tool to check if the vulnerability was detected on the transformed project code. Table 3.1 and Table 3.2 present the results of our experiments. For each project, we specify the CVE, the vulnerability class, the SAST tool used[8], the testability patterns identified and preventing the SAST tool from reporting the vulnerability, and the new injections (alerts) from the tool on the transformed code.

After refactoring the testability patterns, all the CVEs' vulnerabilities were detected and many more true positives were reported by the tools. Some of these true positives still applied to the latest versions of the projects and our responsible disclosure lead to three new CVEs[9].

In the rest of this section, we present some of the transformations that

[8]For PHP projects, Comm_1 is the only tool that we used in this experiment and thus we avoid adding the Tool column.

[9]CVE-2021-33557, CVE-2021-33300, and CVE-2021-23342

Table 3.2: In-depth analysis on JS real cases

Project	CVE	Vuln.	Tool	Patterns	SAST After (TP/FP) XSS	CODEi	SUM
Docsify	CVE-2020-7680	XSS	Comm_1	(P87, T1), (P49, T1), (P55, T1), (P79, T1), (P7, T1, T3), (P99, T2), (P78, T2), (P101, T2), (P21, T1)	5/24	0	5/24
Apexcharts	CVE-2021-23327	XSS	Comm_2	(P87, T1), (P101, T2)	6/3	0	6/3
Hello.JS	CVE-2020-7741	XSS	Comm_1	(P24, T1), (P82, T2), (P21, T1)	3/24	0	0
Lazysizes	CVE-2020-7642	XSS/ CODEi	Comm_2	(P14, T1), (P7, T1), (P83, T1), (P78, T2)	4/0	1/8	5/8
Angular Exp.	CVE-2021-21277	CODEi	Comm_2	(P87, T1), (P36, T2), (P86, T3), (P90, T2), (P75, T2), (P21, T1)	1/0	0	1/0
				17 patterns	19/51	1/8	20/59

we applied and we discuss the new alerts reported by SAST tools. More details about the case study projects and their results are available in our repository [34].

3.6.1 Transformations

During these experiments, we encountered a combination of 9 unique PHP and 17 unique JS tarpits that prevented our tools (Comm_1 and/or Comm_2) from discovering the known vulnerabilities. To refactor the corresponding code patterns, we applied three types of transformations. We present hereafter one example from each type. The details for all transformations are available in our repository.

T1 - Semantic-preserving Transformations. This type of transformation can be applied automatically while preserving the semantic of the code. An example of this category is the refactoring of the D1 variant of `callback_functions` pattern (P78 in Table 1) presented in Section 3.1 for the MantisBT PHP project. By replacing line 12 in Listing 3.1 with the equivalent `$r = gpc_get($args)`, the tarpit disappears, the code semantic is preserved, and the SAST tool becomes able to understand the code and detect the vulnerability. Overall, five out of nine (for PHP) and nine out of 17 (for JS) of our transformations belong to this category.

T2 - Over-approximations. Transformations in this category aim at reducing FNs by increasing the amount of code that can be analyzed by the SAST tool. However, they achieve this goal at the price of breaking the semantic of the program, as the removal of the tarpit requires to introduce an over-approximation. We experience this case in the Bakeshop Online Ordering PHP project where the pattern JS redirect (P57) is used to redirect the user to `index.php`:

```
1 function redirect($lc=Null){
2 echo"<script>window.location='{$lc}'</script>";
3 } redirect("index.php");
```

This pattern is not supported by any of the SAST PHP tools in our arsenal. To enable the SAST tool to understand the redirection we rewrote line 2 as `include($lc)`, thus removing the JS code altogether. However, this modification changes the semantics of the code as the redirection via `window.location` provides to the new page access *only* to the *session* variables of the previous page, while `include` provides access to all variables. For example, if there is a variable `v` controlled by the attacker in page A, `v` will be fully accessible to page B included in the context of page A. It is interesting to observe that this transformation introduces a new tarpit in the code: the `dynamic include D2` (P79) is occurring after the transformation as the location of the PHP file to be included is captured in a variable (`$lc`). However, this pattern instance is easily refactored via a T1 transformation. This shows that the refactoring cannot be performed in one single shot, but it needs to be repeated until no pattern instances are left in the code.

T3 - Developer-Assisted Transformations. While both T1 and T2 can be refactored in a fully-automated fashion, in some cases we noticed that we could not remove the tarpit without some form of human assistance. This support can be in the form of code annotations that specify the propagation of data within complex code areas. For instance, we encountered this situation in the Online Bus Booking PHP project. Here the developers use the extract function—which we explained in Section 3.3.1—to generate variables dynamically from an array. SAST tools have trouble computing these variables statically. A simple annotation, added before the extract operation, is sufficient to help:

```
//@sast:propagate($_POST,[$username,$password])
```

Notice that this annotation can be added by the developers once our pattern discovery rules make them aware that SAST tools will not be able to interpret that pattern in their code and that some annotations would be helpful to overcome such SAST limitation. From that annotation is simple to execute a transformation that simply add to the code the following instructions to make the extract explicit:

```
$username = $_POST["username"];
$password= $_POST["password"];
```

Some SAST tools accept in input special annotations to help them in the analysis – and therefore this type of transformations could be implemented

by using those annotations. However, we did not explore this direction in the chapter as this approach would make the transformation tool-specific.

3.6.2 Results upon transformations

After running our SAST tools on the refactored code, we confirmed that they were able to correctly report all ten known vulnerabilities. Moreover, since the transformations allowed them to better analyze the application code, they also additionally reported 503 new potential injection vulnerabilities. After manual verification, we confirmed 224 of them to be true positives. In some cases, the new vulnerabilities were similar to the ones already reported in the CVEs. In other cases, after removing the tarpits the SAST tools were able to detect new vulnerabilities of different types. Motivated by the high number of new true positives, we moved our attention to the latest versions of the applications (*LatestCode*, in short). First, we used our discovery methodology (see Section 3.4) to confirm that the testability patterns preventing the detection of the old CVE were still present in the current code. Second, we run the testing tool on the latest versions, both before and after applying our pattern transformation to the whole project. This allowed us to discover several previously-unknown vulnerabilities, which received new CVEs. These include one in the MantisBT project (CVE-2021-33557), four in the latest version of Domainmod (CVE-2021-33300 in process), one in Docsify (CVE-2021-23342) and one in Apexcharts (validated by the developers and rewarded with a bug bounty, while a CVE will be released soon). Finally, for both Bakeshop Online Ordering and Online Bus Booking, all true positives reported in the old versions were still applicable as we directly worked on their latest version for the known CVEs experiment. However, in this case, no new CVE was assigned because the new findings were considered as variations of the main CVE.

3.7 Experiment: Automated pattern transformation

In the previous section, we investigated whether known vulnerabilities could be discovered by SAST tools once the obstacles related to our testability patterns are removed from the corresponding applications. After their code was manually transformed, we discovered that SAST tools were also able to discover a number of new, previously-unknown vulnerabilities – thus

Table 3.3: Large-scale transformation experiment

	SC		G_L		G_M		G_H	
	occ.	prj.	occ.	prj.	occ.	prj.	occ.	prj.
R_1	14	9	927	128	2029	160	2116	210
R_2	21	3	89	24	155	39	202	53
R_3	130	9	1287	125	1929	179	2582	254
R_4	130	12	3613	173	8724	263	7043	340
R_5	21	7	258	89	524	113	488	149
Total	316	21	6174	281	13361	382	12341	486
Alerts	18	3	7086	19	1019	26	1297	24
TP	18	3	224	13	73	12	55	15

confirming the negative impact that our patterns have on static analysis.

Motivated by these results, we decided to develop a number of fully automated routines to transform the code of five simple PHP patterns.[10] We then applied these routines to all the applications in our four datasets (G_L, G_M, G_H, and SC, introduced in Section 3.4) and run both the original and the transformed code through Comm_1 (the top-performer tool in our assessment for PHP). Finally, we manually compared the alerts reported by the tool before and after the transformations to distinguish between false positives and real vulnerabilities.

The five testability tarpits we selected to transform for this large-scale experiments are:

[R1] Callback functions (D1), P80 – as we described in Section 3.1, `call_user_func` can be used to invoke a function or an object's method by passing its name as parameter. For the simple case in which the target function is constant, we implemented three refactoring rules:

```
//Before
call_user_func("F",$x);
call_user_func($obj,"method1");
call_user_func_array("F",$args);
//After
F($x); $obj->method1(); F(...$args);
```

[R2] Callback functions (D3), P80 – This routine applies to a variation of the previous pattern in which the name of the function to invoke is obtained by concatenating a prefix string to a variable. In this case, our routine transforms the code by first retrieving all functions whose name starts with the prefix and then by invoking them inside `if` statements:

[10] For this experiment to be done for JS, we need to wait for a CPG generator to automatically and precisely discover patterns, cf. Section 3.4.

```
/*Before*/ call_user_func("Func_" . $x);
/* After */
if($x == "F1") {Func_F1();}
else if($x == "F2") {Func_F2();}
else if ...
```

[R3] Get arguments, P17: In PHP it is possible to define a function to receive an arbitrary number of arguments and then retrieve them from the code by invoking the `func_get_args` function. We transform this pattern to use the PHP variadic function instead (less problematic for SAST tools):

```
/*Before*/function F(){$y=func_get_args();}
/*After*/ function F(...$x){$y = x;}
```

[R4] Foreach with array, P58: PHP provides an internal function called `array_keys` which returns the keys of an associative array. Our routine transforms the use of this function in *foreach* loops as follow:

```
/*Before*/foreach(array_keys($arr) as $key){}
/*After*/ foreach($arr as $key => $value){}
```

[R5] Exit, P55: the `exit` function terminates the program and prints to the user the message it receives as a parameter. Therefore, if the message is controlled by the user and not properly sanitized, this can result into a XSS vulnerability. SAST tools do not take this potential sync into account. Our routine makes this explicit by performing the following change:

```
/*Before*/exit($value);
/*After*/ echo($value); exit();
```

Table 3.3 reports the results of our experiment. For each of the five tarpits, we counted the number of occurrences (which were transformed by our automated routines) and the number of projects that were affected. In total, our tool modified 1170 applications, by refactoring 32,192 occurrences of the five tarpits. As a result of these transformations, Comm_1 raised 9420 additional alerts for 72 of these applications (an average of 130.8 alerts per application). For 17 of these applications the new alerts related to more than one vulnerability class. Specifically, the new alerts applied to 103 pairs of application and vulnerability class, referred to as (app, vuln).

The verification of the new alerts required an intensive manual effort. If the pair (app, vuln) featured less than 50 new alerts (in total, 82 pairs), then all were inspected and classified as either true positive or false positive

(587 alerts were classified in this phase). Otherwise, we sampled 20% of the new alerts related to the pair (app, vuln), ensuring that alerts with different source-sync combinations were included. This resulted in the manual verification of 21 pairs and their 8833 new alerts. Finally, if any true positive was identified in this step, then the entire set of new alerts of the corresponding pair was inspected (this resulted in three pairs for which we inspected 425 new alerts). In total, we manually inspected 2700 new alerts.

Overall, this process allowed us to confirm 370 vulnerabilities in 43 applications, all of which were responsibly disclosed to the respective developers. However, not all maintainers answered our messages. For instance, out of the 55 vulnerabilities discovered in 15 popular GitHub projects, 36 (from 10 projects) were confirmed. In the G_M dataset, developers acknowledged while we received only two answers from the low popular projects G_L after we disclose 224 vulnerabilities in 13 applications. Based on these discoveries, three CVEs (CVE-2021-43673, CVE-2021-43682, and CVE-2021-43687) have already been assigned and many more have been reserved and will be published soon.

We also discovered 18 unique vulnerabilities (CVE-2021-44280) in demo code applications (SC dataset) used to showcase functionalities for other developers and, thus, suitable to be copied&pasted to speed up code implementation, with the subtle risk of porting the vulnerabilities in other applications.

False Positives Discussion

Table 3.3 shows that a large fraction of the new alerts, (approximately 2300 over 2700 alerts we manually inspected) are false positives. However, it is important to understand that this does not mean that our *transformations are the cause for those false positives.* Our transformations increase the amount of code that can be analyzed and tested by SAST tools. The more code is analyzed, the more likely the tool is to report findings, most of which are unfortunately false positives. Without our refactoring, the SAST tools are simply blind to those code areas.

Those blind code areas may be many and their impact on false-negatives significant. For instance, by inspecting the MantisBT project—part of the G_H dataset and presented in Listing 3.1—we identified other 9 functions that, as `gpc_get_string`, suffer from the callback functions (P80, D1) tarpit and invoke `gpc_get` in a (useless) dynamic fashion. These functions are called 769 times in 182 files.

A second important point is that all the five automated transformations

Table 3.4: False positives experiments

Project	Before (TP/FP)	Transf.	After (TP/FP)
MantisBT (1.4k)	0/90	73	1/207
Cloudflare-CNAME-Setup (1.3k)	42/46	16	10/10
Librenms (2.4k)	49/135	144	1/2

applied in our experiment are semantic-preserving (T1) and as such they do not add any over-approximation. Thus, the large numbers of false-positives emerging in that experiment is essentially due to the ability of SAST tools to better understand the application code. To confirm this observation, we performed an additional experiment in which we manually inspected a few popular projects (in G_H) for which we received answers (and confirmed vulnerabilities) from developers. Our goal was to demonstrate that there is no direct relation between our transformations and the false-positives rate. Table 3.4 shows the results. For each project, we indicate the ratio TP/FP before transformations, the number of transformations, and the ratio TP/FP of the new alerts raised on the refactored code. We can see that the ratio between the number of transformations and the new alerts is very diversified. Note that these results were validated with the development teams to improve the projects' quality. For instance, we submitted a pull request for Librenms (2.4k stars) that was promptly accepted to fix the detected vulnerability.

While the above discussion is true for the transformations we used in our large-scale experiments, not all possible transformations have no impact on false-positives. For instance, T2 transformations could, as a side effect, increase the number of false-positives and as such they should be used with parsimony. We foresee the developers playing a key role in our approach deciding whether those transformations should be applied or not.

3.8 Limitations

Pattern Discovery. In Section 3.4 we explained how our discovery rules first perform a static analysis of the application code. However, since the patterns we want to discover are by definition those that are problematic for static analyzers, this might seem a contradiction. In reality, the fact that our rules can discover the presence of a tarpit (e.g., the use of a given instruction in a certain context) does not necessarily imply that the underlying static analysis engine is able to correctly handle the pattern. For instance, our rules can detect that a piece of code performs a string operation even though

the static analysis framework the rule is built upon cannot reconstruct the actual value of the string.

It is important to note that, while implementing sufficiently precise static taint analysis is simpler on an intermediate language, such as PHP opcode, some information (such as class inheritance) is lost in the compilation process or optimized out by the JIT compiler. In case our rules need access to this information (five patterns in total) we had to implement a simple ad-hoc text processing script to detect the tarpits at a syntactic level.

In other cases, our queries might under-count the number of instances of a pattern due to the difficulties of static taint analysis, in particular when tracking taint that is not passed from function to function via a method call. Similarly, patterns in the D3 category (where only part of a value is constant) could be implemented in many possible ways, but we only count when they rely on string concatenation and not, for instance, on sub-strings substitutions.

Because of these limitations, it is important to understand that the number of times a pattern is reported by our rules is a lower bound over the actual number of times it can be present in the code.

New Patterns. We tried to be comprehensive in our pattern catalogs by systematically inspecting all the Chapters of the language documentation (for both PHP and JavaScript) and by including development community comments about language corner cases. However, we reckon that new features (e.g., from new widely used libraries) may emerge and result in new patterns. For this reason, we designed our approach to be extensible towards the addition of new patterns and we are developing an open-source framework where the community can add patterns by following a well-defined format. The procedure is described in detail in our repository [34].[11]

Once a developer has identified a new challenging code fragment, adding the corresponding pattern is a matter of a few hours of work. The most challenging part is coding the pattern discovery rule, which requires some knowledge in Scala and Joern. However, the developer can count on many examples in our catalog as well as on a broad and active community [143], which is another advantage of building our system on top of a popular framework.

For example, if a user suspects that the use of the PHP API function `substr` could be a potential tarpit for SAST tools, she will first adapt the pattern stub (cf. Section 3.3.1) by adding a call to `substr` between the

[11] `https://github.com/enferas/TestabilityTarpits/tree/main/Docs/AddingPatternProcess.md`

source and the sink:

```
$a = $_GET["p1"];
$b = substr($a,0,15);
echo $b; // XSS
```

Second, she would initiate the SAST tools validation scan (an operation fully supported by our framework) to collect the impact on the new pattern. Third, if one tool fails to discover the vulnerability on the stub, the pattern will be added to our collection and a discovery rule will need to be created. This can be easily done by tuning any of the discovery rule created for other API patterns (e.g., P59-P65 in Table 1) via a simple replacement of the API function name with `substr`:[12]

```
cpg.call(".*INIT_FCALL.*").argument.order(2).code("substr").size
```

Beyond Injection Vulnerabilities. While we focused on injection vulnerabilities (XSS, SQLi, Code Injection, File injection, Command Injection, and Path Manipulation), other types of vulnerabilities can be covered as a future work (e.g., Information leakage and/or improper error-handling attacks [146]). In fact, even though our tarpits focus on data flow-related challenges, many of them are quite general and also impact the analysis of the control flow of the application. A complete set of tarpits related to control flow can be an interesting follow-up for our work.

3.9 Related work

Over the past two decades, researchers have developed many tools and techniques see e.g., [126, 120, 75, 107, 108] to statically identify vulnerabilities in source code. Our research does not introduce a new vulnerability discovery techniques but rather focuses on the difficulties these tools face. The three research areas most closely related to our work are: software testability, studies of the limitations of static analysis, and comparisons of web SAST tools. In the following, we discuss related work in each of the three areas.

Software Testability. While *testability* of software artifacts has been the subject of a large body of research in the software engineering community, its definition remains unclear. In a recent survey, Garousi et al. [74] collected 33 different definitions of testability from different sources, finding the most common to be that given by ISO, which defines testability as the

[12] `INIT_FCALL` opcode is used to call internal functions and the name of the function is the second argument.

"*attributes of software that bear on the effort needed to validate the software product*". This captures a common interpretation accepted by software engineers, which sees testability as a measure of the number of test cases needed to test a program and/or of the difficulty of generating those cases.

In this chapter, we instead use the term testability to measure the ability of static analysis tools to "understand" the code, with the goal of discovering security vulnerabilities. As such, our definition captures not the effort required to generate test cases, but the challenge of analyzing the code.

Despite the difference in scope, our work shares similarities with other studies in which the authors focused on either *improving* or *measuring* testability. In the first category, researchers have mainly studied source code transformations to improve automated test data generation [85, 84, 56, 95, 98, 46, 88].

On the measurement side, Garousi et al. [74] discuss 35 papers that present metrics to deal with testability. For instance, Gupta et al. [82] propose three fuzzy metrics for object-oriented software testability: Depth of Inheritance Tree, Coupling Between Objects, and Response For a Class. More recently (in 2020), Oluwatosin et al. [124] list 20 publications that measure testability of software design and categorize them based on whether they were related to Encapsulation, Coupling, Cohesion, Inheritance, Polymorphism, or Complexity.

While no previous study has systematically looked at patterns that prevent static tools from discovering vulnerabilities in web applications, previous work has already covered some of the aspects that can affect testability of programs written in dynamic languages. For example, Alshahwan et al. [44] list three categories that affect the testability of web applications (1) Forms, (2) Client-side validation, and (3) Server-side manipulation. Bures [60] instead define two types of patterns that affect testability, the first one represents the anti-patterns, and the second represents the functional features of the front-end application. The same author also introduced a semi-automated framework to collect metrics for automated testability [59].

Challenges for Static Analysis. Our work also shares similarities with research on the limitations of static code analysis. For example, Landman et al. [101] analyzed the main challenges to perform static analysis on programs that use Java reflection. The authors' experiments show that for 78% of the projects in their dataset, reflection could not be resolved statically. They also provided suggestions for developers to analyze reflection code as well as improvements for static tool builders. Sui et al. [142] compared three tools that had difficulties in discovering vulnerabilities in JAVA applications

because of the presence of dynamic features and reflection. Other papers discussed the challenges in analyzing dynamic proxy API [72].

In general, the use of dynamic features is the main challenge for static analyzers. Kyriakakis et al. [100] defined a number of patterns of PHP dynamic features and they counted their frequency in 10 projects, while Hills et al. [90] proposed a categorization of the PHP features and counted them in 19 projects. The authors also counted the prevalence of dynamic features and how many times they could be resolved statically, finding that 78% of the *dynamic includes* and 61% of the *variable variables* are statically computable.

Medeiros and Neves [115] recently published the first work that looked at the impact of *coding styles* on the accuracy of SAST tools (RIPS, WAP, and phpSAFE) applied to web applications. The authors define six scenarios of coding styles, with three vulnerabilities each. In all cases, they found that the tools identify true positives when the query source is defined closer to the sink and false negatives when it is defined farther from the sink.

Comparison of SAST Tools. Several studies have compared the accuracy and effectiveness of SAST tools. Nunes et al. [122] proposed a benchmark to compare five popular tools on their ability to discover SQL injections and XSS vulnerabilities in 149 WordPress plugins. Kupschs and Miller [99] studied the ability of two popular commercial applications, Fortify and Coverity, to discover 15 vulnerabilities in the Condor project. Each vulnerability was validated by hand, and classified by the authors according to its difficulty to discover (8 Difficult, 1 Hard, 5 Easy). Finally, Spoto et al. [138] uses object-sensitive taint analysis to build their static taint analysis for web applications in JAVA and compare their results with ten static analyzing tools.

Novelty. While other researchers have performed similar studies (but limited to only a handful of patterns) to identify challenges for crawlers and dynamic testing, to the best of our knowledge this work is the first to evaluate web applications based on how easy they are to analyze for SAST tools. By using our approach developers can get a precise indication of the fraction of the application that a SAST tool will not be able to cover. This helps to put the results of static testing into context and to suggest which SAST tool is better suited to analyze the application. Finally, our study is the first to show the impact of code refactoring on vulnerability discovery for web applications.

Previous works (Kyriakakis et al. [100] and Medeiros and Neves [115]) provided inspiration for some of our tarpits, in particular for the dynamic

patterns. On the other hand, our tarpits provide new metrics to compare between static security tools, and provide a number of novel findings which were never discussed in the state of the art. Finally, previous papers have compared SAST tools based on their accuracy of discovering vulnerabilities in real-world applications, while in our work we compare them based on the types of code patterns they are able to handle correctly.

3.10 Conclusion and future work

In this chapter, we demonstrated that specific code patterns, which we call testability tarpits, are a major impediment for static analysis of real world web applications. In particular, we assembled a library of testability tarpits for the two most used web programming languages (PHP and JS) and we validated them by using a mix of state of the art open-source and commercial SAST tools. By defining discovery rules for these tarpits and applying them on thousands of open-source applications, we showed that these tarpits are widely used, indicating that nowadays static analysis has plenty of *blind-spots*. Finally, we performed two sets of experiments to show that refactoring the code to remove tarpits has a significant impact on the alerts reported by SAST tools, leading to the discovery of many previously-unknown vulnerabilities.

We believe the framework discussed in the Chapter introduces a novel way to think about security testing. By shifting the focus from the testing tools to the code of the application, our solution allows to better assess the residual risk that a vulnerability is still present in the code after static testing. We contributed all our discovery rules to the popular Joern community, in the hope that other researchers and developers will extend our library of tarpits and adopt to assess the security testing of web applications. All other results and resources we developed about our research are available in our public repository [34].

Chapter 4

WHIP: Improving Static Vulnerability Detection in Web Application by Forcing tools to Collaborate

Preamble

Improving the accuracy of static application security testing (SAST) is key to fight critical vulnerabilities and increase the security of the Web. However, as we saw in Chapter 3, even state-of-the-art commercial tools have many blind spots that limit their ability to properly analyze modern code and therefore to discover complex inter-procedural vulnerabilities.

In this chapter, we present WHIP, the first approach that enables SAST tools to '*collaborate*' by sharing information that can help them to overcome each other's limitations. Our technique only operates on the application source code by using different tools as oracle to search for signs of interrupted data flows. When we discover such obstacles we inject alternative paths that circumvent the piece of code that SAST tools were not able to handle correctly.

We conducted extensive experiments by analyzing over 100 popular PHP projects with more than 1,000 stars on Github. Our experiments show that our approach enables two popular SAST tools to increase their coverage of the applications' source code, resulting in an increase of up to 25% in the number of high-severity alerts. We manually inspected 30% of the novel 9,226 new alerts obtained by WHIP and responsibly disclosed 35 zero days injection vulnerabilities over 14 applications.

4.1 Introduction

According to a survey published as part of the *OWASP Code Review Guide* [125], the most common approach adopted by developers to identify injection vulnerabilities in Web applications is through Source Code Scanning Tools. While these tools (also called Static Application Security Testing tools, or SAST, in the industry) are invaluable instruments for vulnerability detection, their accuracy is still fairly limited. For instance, several comparative studies [103, 138, 99] have found that even commercial tools struggle to cope with the complexity of real-world applications. Chapter 3 studied one of the reasons behind these limitations, by assembling a library of hundreds of PHP and Javascript code snippets (called *testability tarpits* by the authors) whose presence prevented SAST tools from inferring the data-flow link among two elements of a target program.

One of the main findings of Chapter 3 was that tarpits affect different tools in different ways: what poses a problem from one tool may be analyzed correctly by another and vice versa. Moreover, the authors noticed

that these code patterns are very common in today's applications, with the average project on Github containing 21 different tarpits, each present multiple times. This translates to the fact that even the most advanced commercial SAST tools on the market were unable to analyze applications in depth, without encountering a pattern that prevented them from correctly modeling the code [36].

These limitations are well known by practitioners, who try to mitigate the risk of false negatives by analyzing their application with multiple static analysis tools, in the hope that what a product misses, another can find. For instance, the NIST organization published a document on static code analysis [114] where they explicitly suggest the best practice of combining the results of two or more tools.

This idea of combining the alarms generated by different static analysis tools is also often supported by researchers. For example, Nunes et al. [121] performed an empirical study of combining the results of static tools. Muske et al. [118] published instead a survey about research directions on handling static analysis alarms. The authors cite many papers that discuss the concept of alarms ranking, where the severity of an alarm is chosen based on how many tools raise the same alert.

Unfortunately, combining the alarms of different tools can reduce the risk of false negatives only to a certain extent. In fact, any sufficiently complex application would contain enough different tarpits to impede the analysis of all SAST tools. Thus, even if for different reasons, it is very likely that each tool would encounter a snippet of code it cannot handle correctly. For this reason, in this chapter, we argue that it would be more beneficial to somehow combine the *internal model* reconstructed by different tools, and not just the vulnerabilities they discover. However, the collaboration between the tools – where one tool uses its strength to overcome the weaknesses of another – has been rarely explored by researchers. NAVEX [40] only mentions the collaboration between static and dynamic analysis solutions, when a crawler (dynamic) can be used to retrieve the flow between files that are later analyzed by a static tool.

In this chapter, we present the first approach **to enable the collaboration between SAST tools**. Our novel technique only operates on the application source code, thus allowing our approach to be applied also to commercial tools, without the need to access their internal data structures. Our idea is to search for signs of interrupted data flows, by using the tools as oracles, and then inject another path to circumvent the piece of code that tools were not able to handle correctly.

Our approach is general and can be applied to any programming language. As an example, we implemented a fully-automated prototype, called Whip, targeting the PHP language, which today is still by far the most common language to develop Web applications (78% market share in 2022 [8]).

We conducted a number of experiments to show that Whip can increase the amount of source code processed by two research tools (WAP and Progpilot) and two commercial SAST tools (Comm_1 and Comm_2) and in turn lead to a higher number of security alerts. Over 114 popular projects (all with more than 1,000 stars on Github) Comm_2 reported 25% and Comm_1 reported 10% more alerts, corresponding to 9,226 new high-severity alerts that none of the tools was able to discover in isolation. By sampling and manually investigating 2,732 (30%) of these new alerts, we confirmed the discovery of 35 zero-day vulnerabilities across 14 applications, 24 of which have already been confirmed by the respective developers. However, research tools did not demonstrate clear benefits in comparison to commercial tools in our approach.

Finally, we compared the complexity of the new vulnerabilities discovered by Whip with a dataset of 100 CVEs. Our analysis shows that by using a tool to overcome the limitations of another, both tools are able to explore deeper into the target dataflow. For instance, while the average vulnerability in previous CVE contained a path (between a source and a sink) of only 7.8 lines of code (LOC), the shortest path among our 35 new discoveries is 12 LOC long, and the average is 25.

The rest of the chapter is organized as follows. Section 5.2 presents background information on the static analysis tools and their efficiency in detecting injection vulnerabilities. We then present a motivational example (Section 5.3) inspired by one of our discoveries. Section 5.5 illustrates our approach and the algorithm that we created to apply the changes to the source code and force different tools to collaborate. We discuss the impact of false positives and false negatives in Section 4.5. Finally, we present the design and results of our experiment in Section 4.6 and 4.7.

4.2 Background

Static analysis tools scan applications without the need of deploying the project, by analyzing their code for signs of security issues [104]. Researchers have proposed different models to capture both the syntax and the semantics of source code. *Code property graphs* [158] (CPGs) became one of the most popular by merging in a single model the abstract syntax tree with three

other graph-based representations: the control flow graph to represent the order of execution of the statements, the program dependency graph to capture the dependency between two statements, and the call graph to represent functions and methods invocations.

This graph-like representation is particularly suited to detect one of the most prevalent classes of vulnerabilities, called *injection vulnerabilities*. Injection vulnerabilities occur when an attacker can inject harmful values into an application that lead to unexpected results when interpreted by other parts of the system. For example, an attacker-controlled snippet included in an HTML page can lead to an XSS vulnerability, which can cause the victim browser to execute attack-provided code.

To detect these bugs, SAST tools need to reason about the flow of user-provided information through the program, starting from the points where attackers can inject their input (called "*sources*") until the points in the program where this input is consumed and interpreted (called "*sinks*"). The variables which carry the data between sources and sinks are called "*tainted variables*". Thus, detecting injection vulnerabilities boils down to discovering a data-flow path that connects a source to a sink, along which the data is not properly sanitized.

This process presents two main challenges. First, the static analysis tool needs to be able to construct the path in the first place, by understanding how data can propagate among different variables and different parts of the code (a process normally called taint propagation). Second, the tool needs to correctly analyze the resulting path to detect whether the user-provided input is properly sanitized to protect against the specific type of injection vulnerability that has been considered. Errors in these two steps can cause the tool to miss vulnerabilities, but also to raise false alarms.

To capture the reason behind these errors, Chapter 3 proposed the concept of the *testability tarpits*, i.e., specific code patterns that can prevent a static tool to properly analyze its code (and therefore build a correct internal model). The authors found hundreds of these tarpits and showed that even the best SAST tools are strongly impacted and cannot fully analyze real-world applications.

Finally, even if the internal graph representation of a web application is built correctly, it is often very large and very time-consuming to explore exhaustively. Thus, static analysis tools often employ several thresholds to limit their analysis and produce results in a reasonable time. For example, in our experiments, we noticed that commercial tools apply thresholds (which are often outside the control of the user) to limit the depth of the call graph,

```
1  <?php
2  function func1($vars){
3      $res = "";
4      foreach ($vars as $var => $val) {
5          $res = $res . $var;
6      }
7      return $res;
8  }
9  function func2(){
10     $args = func_get_args();
11     $ret = call_user_func_array('sprintf', $args);
12     return $ret;
13 }
14
15 $vars = $_POST;
16 $x = func1($vars);
17 $y = func2($x);
18 echo $y;
```

Listing 4.1: Example of an XSS vulnerability

as well as the length of the data and control flow paths they analyze.

Some testability tarpits could be mitigated by modifying the tool and improving its code analysis engine. This can be challenging in the case of commercial tools or when research tools are no longer supported. On top of that, not all testability tarpits can be resolved. For instance, to handle dynamic features (such as reflection and dynamic function invocation) static tools can only offer solutions through over- and under-approximations: the first increasing source code coverage at the price of higher false positives, the second ignoring certain features at the price of increased false negatives. Each commercial static tool is optimized to find the right balance between accuracy, the number of alerts provided to developers to manually review, and the time and resources required to scan projects. This tuning requires static tools to carefully choose the type of code analysis they implement and the threshold they use to control their operation.

As a result of all these limitations, even state of the art SAST tools are often limited to the discovery of *shallow* vulnerabilities.

4.3 Motivation

Listing 5.1 shows a snippet of PHP code inspired by a real XSS vulnerability we discovered in the Cacti fault management framework. The vulnerability exists because the attacker controls the `$_POST` variable at line 15, whose

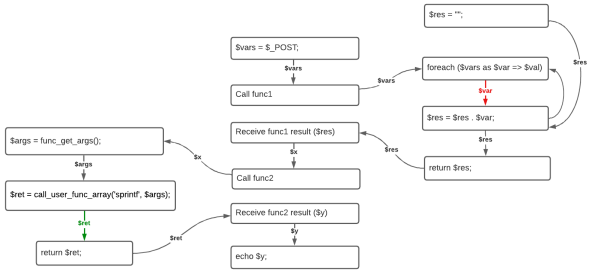

Figure 4.1: The data flow of the motivational example

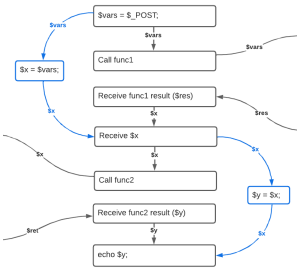

Figure 4.2: The motivational example with the solution

value can reach, without being properly sanitized, the `echo` statement at line 18. Despite the fact that the code is very simple, none of the popular commercial SAST tools we used in our experiment can detect this vulnerability.

To understand the reason we can look at Figure 4.1, which shows the data flow graph of our motivational example. In the figure, there are three sets of blocks, associated respectively to the main function (in the middle) to `func1` (on the right), and to `func2` (on the left). Finally, the edges illustrate the data flow between the different lines of code.

In general, SAST tools identify vulnerabilities by detecting a path between a source and a sink [48, 158]. However, in our example, both tools we tested (Comm_1 and Comm_2) encountered a specific testability tarpit that prevented them from detecting the data-flow that connects the source to the sink. We can detect the testability tarpit that the tool couldn't "understand" by transforming the corresponding line of code and test again if the tool can detect the vulnerability. If it detects it in the second case but not in the original we can conclude that it is unable to infer that particular edge. In one case, `func1` uses a `foreach` loop to concatenate the keys of an array, an operation that is not handled correctly by Comm_1. As a result, the red edge in Figure 4.1 would be missing from Comm_1 internal representation, thus breaking the path associated with the vulnerability. The second case is due to `func2`, which receives its arguments through the built-in function `func_get_args` and then calls the built-in function `sprintf` dynamically through the PHP function `call_user_func`. This code poses problems to Comm_2, resulting again in a missing edge (the green one in the figure) and thus in a fragmented path that prevents the tool to detect

the vulnerability.

This example clearly shows two important aspects, which served as motivation for our research. First, the fact that the reasons why SAST tools fail to discover vulnerabilities are different [72, 101, 142]. Second, the fact that existing tools cannot be *combined* to overcome each other's limitations. Today, all an analyst can do is run both tools in isolation, in both cases failing to discover the aforementioned vulnerability.

Thus, the main goal of our research is to propose a new way to allow a tool to help another. Our intuition is that if we could transfer somehow the part of the dataflow graph of `func1` from Comm_2 to Comm_1, then Comm_1 would have a complete picture of the program and could detect the unsanitized data-flow path associated to the XSS vulnerability. Similarly, transferring the missing red edge from Comm_1 to Comm_2 would achieve the same result, this time helping Comm_2 to detect the vulnerability.

In other words, while different tools use different strategies and are affected by different limitations, their *combined model of the program is more complete, and therefore more effective at finding bugs, than their two models in isolation.*

4.4 Approach

As we explained in the previous section, our goal is to share information about the internal models of two or more tools. However, many popular SAST tools are commercial applications that do not provide access to their code or data structures. Therefore, we need to find a general solution that considers each tool as a black box, which poses a serious constraint to the design of our system.

Our solution is to operate only on the source code of the target application, by using an approach based on two main operations: *infer* and *stitch*. The *infer* operation extracts security-relevant data-flow paths (i.e., those that originate from a source that can contain user-controlled input values) from one SAST tool. To achieve this, we first inject fake sink instructions related to one type of vulnerability (e.g., `echo` for XSS) into the target application. Then, we scan the modified application with each SAST tool and we process all the reported alerts related to the fake sinks. If the path between a source and the fake sink is reported as vulnerable by tool A, it means that the tool was able to build an uninterrupted data-flow path between the two statements. Then, if the same path is NOT reported as vulnerable by another SAST tool B, we can deduce that its code likely con-

tained some testability tarpits that prevented B to reconstruct the same data-flow. Thus, we can use what we learned from A to "*stitch*" (the second building-block of our approach) the two ends of the data-flow together, by creating a new edge in the data-flow graph that can help B to conduct its analysis.

It is important to note that we do not permanently modify the application. Instead, our technique only creates a temporary variant for the purpose of SAST testing. While the variant does not preserve the original semantic of the code, it does not need to be executed and its only purpose is to increase the code coverage of SAST tools and their ability to discover vulnerabilities.

In summary, our technique modifies the application by using a set of SAST tools as oracles to infer which variables are connected to a given source without being properly sanitized. If a tool detects these connections, we forcefully add new data-flow edges to the application (through new variable assignments) to help other tools discover the same connections. The implementation of our prototype is available in our repository [29].

In order to implement this idea, we first need to decide at which granularity we want to perform our "infer and stitch" operations. For instance, one could implement this approach at the variable assignment level, adding a fake sink every time a variable assignment takes place. However, such a fine-grained solution would require the introduction of a huge number of fake sinks and, as a consequence, a very long post-processing phase to analyze the SAST findings.

Therefore, we decided to implement our solution at the function level, where information can only flow between parameters or from a parameter to a return value, thus limiting the number of fake sinks we need to introduce and the required SAST processing time.

The complete approach is presented in Algorithm 1. It takes as input a web application A, a class of vulnerabilities V, and a set of SAST tools Ts. It then outputs any new alert (of the selected vulnerability class) generated from the SAST tools after our transformation. We now explain our approach in more detail, by using again our motivational example presented in Listing 5.1 and two commercial tools (Comm_2 and Comm_1).

4.4.1 Phase I: Prepare

In this phase (cf. lines 7-10 of Algorithm 1), we simply init two variables (i.e., the iteration step i is set to 0, the initial set of stitches ST^0 is set to

Algorithm 1 Approach

```
1: Input
2:     A   web application
3:     Ts  set of SAST tools
4:     V   vulnerability type
5: Output
6:     Ns  new findings of type V
7: Prepare
8:     i ← 0
9:     ST^i ← ∅
10:    Fs^i ← scan(Ts, A, V)
11: InferStitch
12:    A^i ← inject(A, V)
13:    repeat
14:        i ← i + 1
15:        Fs^i ← scan(Ts, A^{i−1}, V)
16:        ST^i ← infer(A^{i−1}, Fs^i)
17:        A^i ← stitch(ST^i, A^{i−1})
18:    until ST^i ≡ ∅
19: Evaluate
20:    A^i ← clean(A^{i−1})
21:    Fs^{i+1} ← scan(Ts, A^i, V)
22:    Ns ← diff(Fs^{i+1}, Fs^0)
```

the empty set) and we run the SAST tools against the original application to collect the set of findings Fs^0 of type V (e.g., XSS). These findings will serve as a baseline to evaluate the effectiveness of our approach in detecting novel findings. When running Comm_2 and Comm_1 on our motivational vulnerable example, Fs^0 is empty as both tools are unable to discover the XSS vulnerability when used in isolation.

4.4.2 Phase II: Infer and Stitch

In this phase (cf. lines 11-18 of Algorithm 1), our first objective is to use a set of SAST tools to determine if there is propagation of tainted values between the input and output of functions. To do this, we modify the application source code by "injecting" (cf. line 12) fake sinks that are related to the chosen vulnerability type after each function call. These sinks should be recognizable by all the static tools in the approach. For example, we choose to use the `echo` statement to write a code snippet that directly prints the user-provided input. We then scan this code using our tools and if they all detect the presence of an XSS vulnerability in the code, we consider the sink to be suitable for our approach.

Our tool uses this technique to print both the inputs and output values of each function (No fake sinks are added for functions that neither return a value nor get input parameters). Since printing a variable can lead to an XSS vulnerability we expect SAST tools to raise an alert if the variable is tainted (i.e., it can contain unsanitized user input).

For instance, in our motivational example our approach would modify the call to `func1`, by adding two fake sinks, as follows:

```
// ...
$x = func1($vars);
/*E1-16*/ echo $vars;
/*E2-16*/ echo $x;
// ...
```

These added instructions are labeled with `E1-16` and `E2-16` to indicate they are fake sinks generated for the function call at line 16 of listing 5.1.

The approach is now ready to iterate over the infer and stitch operations (cf. `repeat-until` loop). Once the iteration step is increased (cf. line 14), the SAST tools are run against the modified application (cf. line 15) and the `infer` operation (cf. line 16) is then used to process the SAST findings. In particular, the SAST findings related to the injected fake sinks are automatically inspected and the following conditions evaluated:
[C1] at least one SAST tool (say A) detects that tainted data can flow from one function parameter (say `$in`) to the return value of the function (say `$out`); and
[C2] another SAST tool (say B) reports that tainted data can only flow into `$in`, but not to `$out`.

If both **[C1]** and **[C2]** are true, our approach uses the findings of tool A to help tool B by introducing a new "stitch" in the dataflow to enforce the data-flow connection between `$in` and `$out`. The inferred stitches are concretized in our approach again as a modification at the source code of the application, this time through the use of a conditional assignment for each stitch. This is done by the `stitch` operation at line 17 of our algorithm. For instance, for the stitch `ST1` capturing the data-flow connection between `$in` and `$out`, the conditional assignment hereafter would be added just after the function:

```
// ...
$out = func($in);
/* STITCH_BEGIN: ST1 */
if(round(rand(0,1))){
  $out = $in;
}
```

```
/* STITCH_END */
// ...
```

Listing 4.2: Simple stitch

We wrap the assignment inside an `if` statement to create an alternative edge in the data-flow, without completely replacing the path through the function. This, as we will explain in Section 4.5, is important to prevent our transformation to introduce new false negatives. As a condition we chose an expression that is randomly computed as true or false at runtime.

In the general case in which a function has more than one parameter, they are all individually tested and, if more than one argument is part of taint propagation, multiple edges (stitches) will be introduced in separate conditional blocks – like in the following example:

```
// ...
$out = func($in1,$in2);
/* STITCH_BEGIN: ST1 */
if(round(rand(0,1))){
  $out = $in1;
}
/* STITCH_END */
/* STITCH_BEGIN: ST2 */
else if(round(rand(0,1))){
  $out = $in2;
}
/* STITCH_END */
// ...
```

Note that while adding stitches, the fake sinks of the input parameters that were used to infer these stitches are removed, as they are not needed anymore. If the fake sinks of all the input parameters of a function are removed, then also the fake sink of the function return variable is removed.

We now describe the whole iterative approach against our motivational example. In the first iteration, Comm_2 raised an alert for both `E1-16` and `E2-16`, a sign that its static analysis algorithm correctly concluded that `func1` propagated tainted information from its parameter to its return value. However, while Comm_2 succeeded, other tools might miss this connection. In fact, in this case Comm_1 raised an alert for `E1-16` but NOT for `E2-16`, due to its inability to process correctly the `foreach` loop in the function body.

To sum up, from this first iteration our approach learned that, through the function `func1`, tainted data propagates from the `$vars` variable to the `$x` variable. Since not all SAST tools in our set detected it, our approach

forcefully add this dependency in the program.

To make the relationship between the `$vars` and `$x` variables explicit, our approach modifies again the source code of the application, this time by adding a simple conditional assignment as in Listing 4.2 where `$vars` and `$x` replace `$in` and `$out`, respectively.

In summary, with the `stitch` operation of our approach, we modify the application to add new instructions that explicitly connect two variables, when at least one tool detects that tainted data can flow from one to the other.

It shall be noted that one iteration of the infer and stitch operations is not sufficient to discover the vulnerability of our motivational example. In fact, if we consider the whole code of our example, during the first iteration Comm_2 raises alert for both the input and the output of the first function (`func1`), while Comm_1 only raises an alert about its input, as it is unable to properly process the function (because of the missing red edge in the first function, cf. Figure 4.1). On the other hand, Comm_2 reports only the input for the second function (`func2`), because of the missing green edge in its model (cf. Figure 4.1), while Comm_1 does not raise any alert, as its analysis is still blocked by the first function.

In other words, the interplay between different tarpits that affect different tools result in the fact that none are able to process the entire chain during the first iteration. Therefore, our infer and stitch operations need to be repeated in an iterative fashion until an equilibrium is reached, i.e., until no new edges (new stitches) are discovered in the graph.

This way, the first iteration our approach helps Comm_1 to understand that `$x` is tainted, and thanks to this information during the second iteration Comm_1 detects that `$y` is tainted as well – a piece of information that helps Comm_2, which previously missed this connection.

Figure 4.2 shows the data-flow graph of the modified application after two iterations of our approach. The new edges, marked in blue, are the stitches introduced by our approach. In the third iteration no new stitches are inferred and our approach moves to the evaluation phase.

4.4.3 Phase III: Evaluate

In this final phase (cf. lines 19-22 of Algorithm 1), our approach first cleans the application code from any remaining fake sinks (cf. `clean` instruction at line 20) and then scans it with all SAST tools in the arsenal. By removing from these SAST findings for vulnerability class V those already reported on the original application (cf. `diff` at line 22), our approach can output

the novel SAST findings emerging because of the stitches added in the previous phase. In order to remove already reported SAST findings, our `diff` instruction compares findings as follows. Two findings $F1$ and $F2$ are considered identical if and only if the sink line of $F1$ is identical to the sink line of $F2$.

For our motivational example, the comparison is trivial as there were no findings reported on the original application. By running our infer and stitch operations, new stitches were added in the first two iterations and none were uncovered during the third iteration. When scanning the final code, both SAST tools in our arsenal were correctly reporting the XSS vulnerability, indicating that two new findings emerged as a direct consequence of our approach that forced the SAST tools to collaborate.

4.5 False Positives and False Negatives

SAST tools employ various techniques to analyze the source code of an application. In particular, to handle dynamic code constructs that cannot be resolved statically, all SAST tools use some form of over- and under-approximation. The two are often combined to find a balance between the amount of code that can be analyzed and the number of false alerts generated [32]. Additionally, SAST tools also incorporate heuristics to halt the analysis of a particular path when specific thresholds are reached (e.g. if the path involves over 500 variables or more than 5 nested functions). The aim is to improve performance and keep the running time in a reasonable range, but this approach can decrease the ability of the tool to detect vulnerabilities.

Our approach reduces false negatives by allowing all tools to access data-flow connections that emerge by combining and complementing the models of each individual tool. As an additional benefit, our approach also reduces the impact of performance-related thresholds since, by adding stitches, we introduce shortcuts that bypass functions and make data-flow paths shorter. This again helps in reducing false negatives. It is also important to note that, by construction, our approach cannot increase false negatives as it cannot miss what individual tools would already detect in isolation. This is also confirmed experimentally in our results, where we never encountered a target application for which our approach was not detecting a finding that was previously detected by one of the SAST tools.

In the majority of cases (around 80% of the applications analyzed in our experiments), our approach is returning novel SAST findings. Although

the majority of these findings are false positives, which is a common occurrence with most SAST alerts, they all result from the examination of novel security-related data-flow paths. In fact, our approach only requires SAST tools to analyze a path if it can be decomposed into a sequence of sub-paths, each of which was already analyzed by at least one SAST tool because the tool believed it may transfer unsanitized user input. Therefore, our algorithm does not introduce more false positives *unless* one of the tools already introduced them because of an over-approximation. In this case, the infer-and-stitch approach enforces the same over-approximation on the other tools, causing all of them to consider the corresponding, potentially erroneous, path.

4.6 Experiments: methodology

We implemented our approach in a prototype tool named Whip. Whip takes as input a web application and one or more types of sinks (which depend on the class of vulnerabilities the analyst wants to discover) and then orchestrates the insertion of sinks, the execution of SAST tools and the collection of the corresponding alerts, and the injection of conditional assignments instruction to create the new edges in the dataflow graph. The tool automatically performs multiple iterations until the data-flow graph converges and no new edges are inserted.

The tools currently support the PHP language, chosen because it is still by far the most common language to develop Web applications, with a 78% market share in 2022[8]. On the other hand, since we are building our stitches at the function call level and Whip does not require any static analysis, our approach can be applied to any programming language supported by SAST tools. To support a new language, the analyst just needs to list the sinks statements used by the language and the syntax required to assign variables. In the rest of this section, we discuss the selection of SAST tools we used in our experiments, their integration into Whip, and the dataset of Web applications we tested with Whip. The results obtained are presented in Section 4.7.

4.6.1 SAST Tools Selection

While our solution is generic and it can be applied to any SAST tool, it is particularly useful in the presence of commercial, closed-source tools whose source code cannot be modified to take into account other complemen-

tary solutions. For this reason, for our experiments we selected two of the most widely used commercial SAST tools that support PHP: Comm_2 and Comm_1. We have acquired full licenses for both tools, allowing us to run our tests without the restriction of and limited analysis time available in the free trial options.

On top of them, we also decided to include in our tests a few selected open source tools to verify whether their presence could benefit the results by helping commercial tools to overcome their limitations. In fact, while much more limited in terms of complexity and supported features, it would be enough for an OS tool to discover a data-flow relationship missed by the more mature commercial alternatives to provide a valuable contribution to the overall *collaborative effort*.

Over the years, the research community has proposed several static vulnerability detection tools for PHP to choose from (including RIPS [65], phpSAFE [123], WAP [151], Progpilot [130], WeVerca [86] and Pixy [94]). For our experiments we selected Progpilot v1.0.2 and WAP v2.1 because they both support scanning entire projects instead of individual files, they both provide a CLI implementation, and they both support object-oriented code. In addition, Chapter 3 found Progpilot to be the best OS tool in terms of its ability to handle the authors' testability tarpits library.

4.6.2 SAST Tools Integration

Since Whip needs to orchestrate the execution of SAST tools, it requires a dedicated module to support the interaction with each tool and the parsing of the generated alerts. So the integration of a SAST tool within Whip requires the implementation of a small interface that handles its operation and the collection and inspection of its alerts.

In order to add a static tool to Whip, two conditions must be met: (1) the static tool needs to have a CLI interface or some form of API to control its operation, and (2) the output (alerts) of the need to be stored in a way that allows for an automated extraction and parsing. These requirements come from the fact that Whip is fully automated and requires the ability to orchestrate the process. For instance, Progpilot and WAP offer CLI commands to scan a project and produce results. To use them with Whip, we run the scan command and redirect their output to a text file. Then, we wrote a script to parse the text file and extract the data required by Whip. The simple parser we developed is available in our repository [29].

It is typical for commercial SAST tools to provide in their reports a list of findings along with the *type* and *severity* of each entry. The sever-

ity was not important in our experiments, since Whip focuses on injection vulnerabilities [20], which are considered by all SAST tools as high severity risks. In addition, for each alert, SAST tools may also report the full path between the corresponding source and sink (i.e., the attacker-controlled input and the point in which the vulnerability is located in the code) to help developers to understand the issue and provide a patch.

4.6.3 Dataset

To measure the benefits of combining multiple SAST tools, we tested our tool on a set of modern and popular web applications. We cloned the latest version of all PHP projects from Github with more than 1,000 stars, resulting in an initial set of 1,183 projects.

We then extracted the number of sources of user-provided inputs in each project, by grepping for the corresponding predefined variables in PHP (e.g., $_GET and $_POST). Roughly half of the projects (602 projects) do not have any source. This is due to the fact that these are often libraries used by other projects and not standalone applications. Among the remaining projects, 127 contained more than 100 sources – thus making them a perfect target for our vulnerability analysis. Thus, we selected these 127 applications as a dataset for our experiments.

The analysis was performed on a machine with 16 cores and 64 GB of RAM. Since each SAST tool needed to be invoked multiple times for each project, we excluded those for which a single analysis did not complete within 6 hours. This was the case for 13 projects, reducing our final dataset to 114 projects. The complete list of applications with their corresponding statistics is reported in Appendix in Table 3. The projects range from small (with less than 10K LoC) to big (with more than 1M LoC). Altogether, they account for 21.4M LoC, 1.9 million functions, and 85K sources of user-provided input.

4.7 Experiments: Results

We break down the results of our experiments in six different parts. First, we will examine the poor performance and lack of contribution of open source tools. Then, by using XSS as an example, we will analyze different statistics that show how Whip performed on the 114 projects in our dataset, including the number of iterations required to converge and the number of additional edges that Whip introduced in the PHP code.

In the third part, we will evaluate the impact of Whip by measuring the number of extra alerts raised by each SAST tool for three types of injection vulnerabilities (XSS, SQLI, and File Manipulation). An increase in the number of alerts indicates that the tools were able to analyze the application code more deeply and process more paths that were previously blocked by the presence of testability tarpits.

In the fourth part, we will discuss the advantages and disadvantages for companies that will use Whip. On the one hand, the tool will increase the coverage of the source code and de-duplicate the alerts from multiple tools. On the other hand, it will require more time and resources to scan the project using multiple SAST tools over a few iterations.

In the fifth part of this section, we show that the model of the application built by SAST tools thanks to Whip is *more* than the sum of the individual models. In other words, it is not just that one tool can help the other to overcome its limitations, but that each tool can now discover vulnerabilities that could not be previously discovered by any tool in isolation.

Finally, in the sixth part, we compared the vulnerabilities we discovered with a dataset of 100 past CVE reports, by analyzing the length and complexity of the data-flow paths associated with each bug.

4.7.1 Research Tools

In our experiments, Progpilot produced results only for 26 out of 114 projects (23%) and crashed in the remaining cases. WAP did better, successfully scanning 90 projects (79%). In both cases, the research tools did not report any additional alerts (i.e., no new edges) on top of those reported by commercial tools, leading to the conclusion that these tools could not be used to enhance the analysis of commercial tools. This poor result is not completely unexpected. In fact, in 2017 Nunes et al.[121] already noticed that none of the static research tools for PHP (RIPS, phpSAFE, WAP, Pixy, and WeVerca) were able to successfully analyze a complex web application in its entirety. Similarly, Chapter 3 found that only commercial SAST tools were up-to-date with recent PHP language features and capable of scanning modern web applications and that the two leading commercial tools supersedes all other open source alternatives in terms of supported testability tarpits.

In conclusion, adding small tools to the arsenal has a cost (in terms of analysis time) but may not bring any clear benefits to security testing. On the other hand, in the next sections we will see how the combination of state of the art solutions can instead lead to a large increase both in terms

Iteration	0	1	2	3	4	5	6	Total
Projects	9	31	33	23	10	3	5	114

Table 4.1: Number of Converged projects over iterations

Iteration	1	2	3	4	5	6	Total
Comm_1	5,475	1,847	365	131	68	46	7,932
Comm_2	6,976	1,450	462	240	144	104	9,376
Combined	12,451	3,297	827	371	212	150	17,308

Table 4.2: Inserted data-flow edges with XSS fake sinks

of code coverage and number of alerts and discovered vulnerabilities.

4.7.2 General Statistics

Table 4.1 shows the number of iterations required by WHIP to converge (i.e., until no more data-flow edges were discovered) for different projects when we run the experiment for XSS vulnerabilities. At each iteration, each SAST tool was able to explore the application deeper, i.e., to test the security of longer and longer inter-procedural data-flow paths that were invisible (or better, fragmented) without our approach. One iteration was sufficient to converge for 31 of our 114 applications, two could cover roughly half of the dataset, and the rest required three or more iterations – with a maximum of six. This is very important, as it means that some applications contained paths between a source and a sink that involved at least six different functions, all of which contained snippets of code that both our tools were not able to process correctly (different tools had problems with different code blocks in alternation).

Table 4.2 shows the number of edges added by WHIP to the applications code, for each iteration and each SAST tool. By far, the largest number of edges (12.4K and 3.2K respectively) were added over the first two passes. In total, our approach added 17,308 new edges: 7.9K to increase the coverage of Comm_1 and 9.4K to increase the coverage of Comm_2.

It is also interesting to observe that not all applications were impacted the same way. For instance, the Dolibarr application contained the maximum number of sources in our dataset (8,609 sources) with more than 8 million lines of code. This project was also the one for which our tool had the largest effect, introducing 108 new stitches for Comm_2 and 206 for Comm_1 after four iterations. At the other end of the spectrum we find the Valet-plus project, which has the minimum number of sources in our

dataset (100), and for which our tool only added two stitches for Comm_2 in the first iteration. Overall, WHIP added 151.82 stitches per project, with a median of 35. More details are presented in Table 3, where for every project we show the number of stitches added and the number of iterations run by WHIP.

4.7.3 New Alerts

To test for new alerts we run three separate experiments, using WHIP to test high severity injection vulnerabilities such as XSS, SQLi, and file manipulation vulnerabilities. As a reminder, the stitches we introduce are always specific for a class of sinks as mixing two types can lead to errors in the alerts. For instance, if a function propagates information unsanitized for XSS but sanitized for SQLi, adding a stitch would result in a 'shortcut' that can make tools erroneously report SQLi vulnerabilities (since the alternative path we insert would bypass the sanitization). Therefore, when we add stitches for a given type of sink, we need to later test only for vulnerabilities that involve the same sink type.

For the three experiments we collected all the corresponding alerts (XSS, SQLi, and file manipulation, depending on the experiment) raised by the SAST tools when scanning (i) the original version of each application as well as (ii) the modified versions generated by WHIP.

Overall, Comm_2 went from 49,231 alerts on the original applications to 61,583 (+25.1%) on the stitched versions. These were divided in 52,843 (versus 42,040) XSS, 1,789 (vs 1,744) SQLi, and 6,861 (vs 5,447) File Manipulation. Comm_1 alerts increased instead from 50,217 to 54,985 (+9.49%), divided into 33,028 XSS (vs 30,062), 10,284 SQLi (vs 9,236), and 11,673 File Manipulation (vs 10,919).

The complete breakdown of the discoveries for each project is presented in Table 3. Even if we expect the majority of these alerts to be false alarms (as we will discuss in more details in Section 4.7.5), these numbers show that both SAST tools were able to improve the number of potentially vulnerable data flows by roughly 25% for Comm_2 and 10% for Comm_1.

We can further divide these sets of new alerts into two different categories: Known and Unknown. The first contains new alerts that are generated by one tool (with the help of the other), but that the other tool was already able to discover by itself. These alerts are associated with source-sink paths in the data-flow graph that only contain new edges for one tool. The second category (Unknown alerts) contains instead the alert that one tool generated (with the help of the other) but that none of the tools were

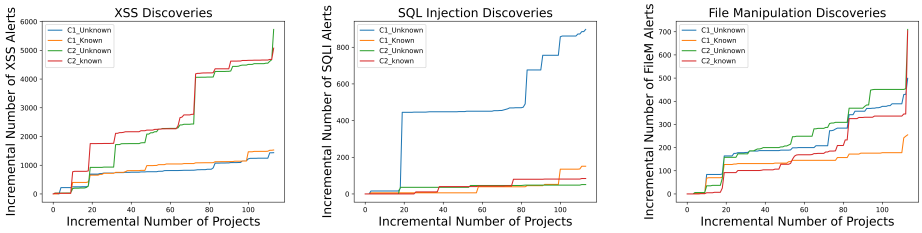

Figure 4.3: The incremental number of alerts regarding the number of projects

able to discover alone. In this case, the path associated with the alert passes through stitches for both tools.

Figure 4.7.3 shows the breakdown of the Known and Unknown alerts in the two categories for the three types. We can see that, over the 114 projects, Comm_1 was missing around 1,936 known alerts (1530, 151, and 255 for XSS, SQLI and File manipulation), which were already reported by Comm_2. On the other hand, Comm_2 was missing 5,864 known alerts (5076, 84, and 704 for XSS, SQLI and File manipulation) that were detected by Comm_1. But it is much more important to focus on the curves of the alerts that were previously unknown. In this case, we can notice that a stunning 9,226 alerts with the highest priority (respectively (i) 2,742 reported by Comm_1: 1436 XSS, 897 SQLI, and 409 File manipulation and (ii) 6,484 by Comm_2: 5727 XSS, 51 SQLI, and 706 File manipulation) were raised for the first time thanks to our tool.

If we look at the different applications in our dataset, 65 out of 114 show an increasing number of high severity alerts for Comm_2, with an average number of new alerts of 108.35. Comm_1 reported instead new alerts in 61 projects, with an average of 41.82. At a closer look, we can observe a clear relationship between the number of sources, the number of stitches, and the number of alerts. For instance, `Dolibarr`, `phpipam` and `testlink-code`, the three projects with the highest number of new alerts, all have more than 900 sources each, and required more than three iterations for WHIP to converge. On the other hand, out of the 35 projects that have less than 200 sources, 19 did not have any new discovery by Comm_2 and 21 had zero new alerts for Comm_1. This seems to suggest again that the more complex the application is, the more likely it is to benefit from our approach, and the higher is the number of new alerts generated by SAST tools.

4.7.4 Overhead

Our approach offers many benefits for enterprises. First of all, our solution increases the coverage of the source code and allows existing tools to report new alerts and detect new vulnerabilities. In addition, WHIP provides a solution for companies that have invested in multiple SAST tools but are facing difficulties in effectively integrating them. Our approach enables the optimization of existing tools and can also save time for testers. In fact, in our experiments we identified 44,000 duplicate alerts that were separately reported by Comm_1 and Comm_2, which our tool automatically de-duplicates and reports only once.

The downside of our approach is the increased scanning time and resource consumption, as static tools must be re-executed over multiple iterations. In our experiment, running Comm_2 alone for one iteration on all projects took 11 hours, 23 minutes, and 21 seconds (6 minutes per project on average). Comm_1 required 36 hours, 6 minutes, and 57 seconds (with an average time of 19 minutes per project). Due to the additional iterations, WHIP took a total of 164 hours, 36 minutes, and 11 seconds to run all experiments until all projects converged. Thus, on average WHIP required 86 minutes per project, versus 25 minutes required by the two commercial tools in isolation (corresponding to a slowdown factor of 3.4X).

A company can also decide to limit our approach to three iterations (reducing the slowdown to 2.6X), a good compromise if we think that only 18/144 projects converged after three iterations, and that 83% of the new alerts were reported over the first three iterations.

4.7.5 New Discoveries

As we previously explained, the use of our technique allowed the two tools to report 9,226 completely new high severity alerts, discovered after adding at least two stitches. To conclude this section, we now look at those alerts and, through a process of manual validation, try to separate false positives from zero-days vulnerabilities.

Since the number is too high for a complete and thorough manual investigation, we started by randomly sampling 10 alerts for each project, for a total of 640 alerts. If at least one of the ten alerts was confirmed as true positive, we proceeded to verify all the other alerts for the same project, to check how many alerts will be fixed when we fix this real vulnerability. In total, this resulted in a set of 2,732 alerts we manually investigated. Each time we confirmed that an alert was a true positive, we contacted the devel-

	Type	Project	Stars	Discoverd_By	Vuls	Status	CVE	FUNC	LEN	Stitch
1	XSS	Vesta	2700	Comm_1	1	Confirmed	CVE-2022-36305	6	23	2
2	XSS	Jukebox-RFID	1000	Comm_2	4	Confirmed	CVE-2022-36749	2	12	2
3	XSS	Cacti	1200	Comm_2	3	Confirmed	Requested	8	33	2
4	File M	ICEcoder	1400	Comm_2	2	Confirmed	CVE-2022-34026	4	14	2
5	XSS	Dokuwiki	3500	Comm_1	1	Confirmed	CVE-2022-28919	4	13	2
6	XSS	PicUploader	1000	Comm_1	4	Confirmed	CVE-2022-41442	4	16	2
7	XSS	Phoronix	1700	Both*	7	Confirmed	CVE-2022-40704	14	43	2
8	XSS	Librenms	2800	Comm_2	1	Confirmed	CVE-2022-36746	7	32	2
9	XSS	Phpipam	1700	Comm_2	1	Pending	CVE-2022-41443	5	17	2
10	File M	Dzzoffice	3500	Comm_2	7	Pending	-	4	11	2
11	XSS	Razor	1100	Comm_2	1	Pending	CVE-2022-36747	7	18	2
12	XSS	Pfsense	3900	Comm_2	1	Confirmed	CVE-2022-42247	4	16	3
13	XSS	Carbon-Forum	1800	Comm_2	1	Pending	-	8	47	4
14	XSS	SuiteCRM	3100	Comm_2	1	Pending	-	11	53	4
	SUM	14			35					

Both*: In Phoronix project, five discoveries deteced by Comm_2 and two detected by Comm_1

Table 4.3: New Vulnerabilities Detected with Our Approach

opers to initiate a process of responsible disclosure. In each communication we described the issue and provided feedback on how the vulnerability could be fixed, and in some cases even submitted ourselves pull requests on Github containing the patch.

At the time of submission, we identified 35 zero-day vulnerabilities in 14 projects. Developers have confirmed **24** of these vulnerabilities (from 9 different projects), as shown in Table 5.2. The remaining 2697 alerts we investigated turned out to be false alarms. The fact that 98% of the validated alerts were false positives should not be a surprise as SAST tools are known, unfortunately, for their very high false positive rates. With our approach, the total number of generated alerts increased by roughly 10%. In fact, without WHIP Comm_2 and Comm_1 already reported a stunning 99,448 alerts. If we consider the fact that we only scanned very popular projects that are regularly analyzed with SAST tools for security purposes, we can expect the very vast majority (if not all) of these alerts to be false alarms.

Table 5.2 reports the number of vulnerabilities aggregated in groups, based on how they were handled by the developers. For instance, in the `Phoronix` application (line 7 in the table) our system found several ways to bypass the sanitize function the authors used in their project. While each way is an independent discovery and therefore a true positive alert, the developers were able to fix all of them by changing the sanitizer, and thus we only requested one CVE covering all the corresponding cases. On top of those already accepted, we also reported 11 other potential vulnerabilities in five projects, for which we did not yet receive an acknowledgment from the developers. Table 5.2 also shows the number of stitches required to discover

each vulnerability. Among our findings, 32 vulnerabilities were discovered with two stitches, one with 3 stitches, and 2 after the insertion of 4 stitches.

At first, one might think that missing data-flow edges (the main contribution WHIP helps to mitigate) is only a tiny factor among many other limitations that affect today's SAST tools. However, it is important to stress that missing edges is *NOT* a limitation per se, but only the consequence of a multitude of other actual limitations. In other words, many problems – from the inability to support certain language features, to missing models of API functions, to the inability to correctly reconstruct inter-procedural control flows, to under-approximation in resolving dynamic behaviors – ultimately result in the inability of a tool to detect the data-flow link among two parts of a program. Given the nature of injection vulnerabilities, these missing edges (independently from the reason why they are missing) are the main cause of undiscovered vulnerabilities in complex real-world applications.

Ethical Risk Assessment. In this study, we responsibly disclosed 35 zero-day vulnerabilities that we detected using our approach. We validated these vulnerabilities by manually checking 2,732 out of 9,226 alerts generated by WHIP. Due to the large number of alerts, we were unable to check them all. If the ratio remains the same, we could expect another 83 vulnerabilities to be present in the remaining 6,494 unvalidated alerts. The names of the static tools used to detect the alerts will be kept anonymous, and we will not publish any information about the non-reported alerts. Finally, we promise to delete all alerts from our servers after the paper has been reviewed and accepted.

4.7.6 Vulnerabilities Complexity

Our approach not only helps tools to discover more vulnerabilities, but also to discover vulnerabilities associated with long data-flow paths, which can be difficult for analysts to discover even through manual inspection. To support this hypothesis we built a dataset of 100 vulnerabilities (CVEs) from Vuln-code DB [28], for which there was enough information to reconstruct the vulnerable source-to-sink path. This requires the corresponding patch to clearly distinguish between the security fix and other changes in the source code, and the CVE to contain an example of input to reproduce the bug. By using this information we manually reconstructed, for each vulnerability, the path between the source and the sink through a manual inspection of the source code. Over the 100 vulnerabilities in our dataset, the average number of functions traversed by these paths is 3.4 (with a median of 3), while the average number of lines of code is 7.8 (with a median of 5).

If we compare these values with the ones associated with the vulnerabilities discovered by Whip (as reported in Table 5.2) we can notice a clear difference. For instance, the length of the paths among the new vulnerabilities we discovered span from 12 to 53 lines of code (with a mean of 24.9 and a median of 17.5). Thus, if we take the length of the vulnerable path as a possible measure of the complexity of a vulnerability, our approach results in vulnerabilities that are, on average, three times more complex than those regularly reported by other means.

This is due to the fact that our solution helps SAST tools to overcome their limitations and therefore to explore deeper in the applications code and detect vulnerabilities associated with long inter-procedural paths.

4.8 Related work

We can identify three research directions related to our work. First, researchers have conducted extensive tests of different SAST tools and they concluded that none of them outperforms the other in all situations. Second, researchers have tried to mitigate the shortcomings of a single tool by either 1) combining static and dynamic techniques, 2) resorting to human experts to help the tool perform better, or 3) combining multiple tools and joining the results. However, our approach is the first that, by combining the internal knowledge of different tools, allows a set of tools to discover more than the sum of the individual components.

Tools Comparison. Many research papers analyze static tools to demonstrate that there is no tool that supersedes all others. Nunes et al. [122] introduce a benchmark for comparing static analysis tools and their effectiveness in detecting security vulnerabilities. The authors chose five static tools for PHP and they found that the best tool varies from one scenario to another, depending on the vulnerability class. Algaith et al. [39] compare the same five static tools as well as their combinations. The authors pointed out that none of the tools (or combinations thereof) can discover all the vulnerabilities in their dataset, but a set of three gave the maximum number of discoveries.

While most of the research papers compare open-source tools, few papers also include commercial tools. For instance, Chapter 3 include three tools, Spoto et al. [138] six and Kupsch et al. [99] compare Fortify and Coverity on the analysis of a single project.

Tools Collaboration. In this work, we present a new direction to enable

static tools to collaborate and discover more vulnerabilities. To the best of our knowledge, there are no other works in this area, previous studies have looked at other forms of collaborations – either between static and dynamic tools, or between static tools and human developers.

[I] Static + Dynamic. Static tools have obvious limitations when it comes to handling dynamic features [90, 89, 100], such as indirect function calls. NAVEX [40] tries to overcome these limitations by proposing the collaboration between static and dynamic approaches. In this paper, the authors proposed to use a crawler to capture the relationship between different web pages, and then use this information to complement a static analysis performed on the application's source code. We can distinguish two cases. If the name of the file is defined statically, then there is no need for the dynamic approach because SAST tools can already detect this file. If, on the other hand, the file is included dynamically then none of the static tools can detect this file, and therefore the solution presented in this paper does not help. Thus, we believe the two techniques to be orthogonal with no intersections in their findings.

Other studies used dynamic techniques to *verify* the discoveries of static tools. For example, Csallner et al. [64] used a hybrid analysis approach to automate bug findings. The proposed approach includes three steps: dynamic inference, static analysis, and dynamic verification. The same authors [63] also presented a different approach in which a constraint solver was used to generate concrete test cases to verify the static tools alerts. There are also other types of collaborations, like the one proposed by Hough et al. [91], in which the authors employ human developers' test suites to support automated dynamic analysis.

[II] Static + Humans. A different form of collaboration that has been explored by researchers to overcome some of the limitations of SAST tools is based on human-in-the-loop approaches. For instance, Al Kassar et al. [36] discuss the collaboration between SAST tools and developers when they provide manual transformation at the source code level to improve the discoveries of the static tools. Other authors study how to change the rules automatically depending on the users' preferences (e.g., in Mangal et al. [111]), or how to provide feedback to the tool's developers to improve the results (e.g., in Sadowski et al. [133]).

Combining the results of multiple tools. Many papers have proposed to combine the output of different static tools, as suggested by NIST "CAS Static Analysis Tool Study Methodology" [114]. Meng et al. [116] show that static tools for JAVA can report different alerts for the same source code

depending on the tool performance in that specific class of bugs. So they ask the analyst to provide the source code and an example of the set of bugs she is looking for. The system then chooses the right tools that are most likely to provide the best results for that type of bugs and returns a merged list of their discoveries. Rutar et al. [132] suggest a bug-finding meta-tool for joining the results of different static tools together. Wang et al. [149] introduce a web service where the user can choose the type of bugs and upload the source code. The system then scans the code with multiple tools and returns the merged results to the user. Finally, Nunes et al. [121] run an empirical experiment on combining the results of five static tools for web applications, reporting the increased percentage of the true-positive and false-positive after this combination.

4.9 Conclusion

In this chapter, we proposed a novel idea to 'force' different SAST tools to collaborate to find more vulnerabilities. Whereas each static tool has its own strengths and weaknesses, our solution allows them to help each other to overcome their respective challenges. Our approach is completely automated and considers all tools as black boxes (thus supporting commercial tools for which we have no visibility on their internal data structures).

By routinely modifying the source code of the application under test, our system can inject fake sinks to infer how tainted values propagate through the different program functions. Whenever one of the tools is unable to 'understand' these connections, we help it by stitching the data-flow with additional edges that bypass the problematic function.

Our experiments performed only on very large and popular PHP applications, show that our approach can successfully improve the amount of source-to-sink paths that each tool is able to analyze. This leads to a total increase in the number of critical alerts between 10-12%. By manually validating a subset of these new alerts, we discovered and reported 35 zero-day vulnerabilities in 14 projects with more than 1,000 stars on Github.

Chapter 5

Mitigating the impact of the MVC design pattern on Web Application Static Security Testing

Preamble

We discussed in Chapter 3 how static tools have many testability tarpits that block them from detecting vulnerabilities, and we discussed in Chapter 4 how static tools can help each other through WHIP. In this Chapter, we show how even the best commercial tools struggle to scan applications developed using frameworks with specific design patterns such as Model View Controller (MVC). In this Chapter, we investigate MVC frameworks and their interaction with the application source code. We identify the challenges that SAST face in understanding this interaction, and we provide a solution to "disconnect" the application from the framework. Our innovative approach operates solely on the application source code, transforming the highly dynamic interaction challenges into static code that is testable for SAST. This enables the SAST of MVC based web applications without requiring any modifications on the SAST tools.

Our experiments demonstrate the effectiveness of our approach. Besides validating our approach on vulnerable applications, we also applied it against the latest version of 20 modern web applications based on the CodeIgniter MVC framework (overall used by more than 1.5 million websites). More than 2,000 alerts were reported by SAST after our transformations: 826 stored XSS and 103 reflected XSS vulnerabilities were confirmed as true positives impacting 18 of the 20 applications. Upon responsible disclosure, CVEs were released to acknowledge our findings.

5.1 Introduction

According to the Acunetix Web Application Vulnerability Report 2020 [2], 25% of web applications are vulnerable to XSS injections. Developers commonly use static analysis tools, also referred to as Static Application Security Testing (SAST, in short) tools, to fight these vulnerabilities [125]. SAST gets in input the source code of an application and try to reason about the entire behavior of that application, to spot dangerous data-flows and other potential issues. However, many limitations have been identified for SAST that, in practice, make this approach neither sound nor complete. SAST is thus subject to both false positives and false negatives, whose prevalence strongly relate to the amount of dynamic features used by the application under testing.

A trend that makes the SAST task even more difficult is the wide usage of model-view-controller (MVC) frameworks in modern web applications.

For instance, only considering PHP, over 23.5% of the Top 1 Million websites [10] use MVC frameworks. If the MVC paradigm eases the development and maintenance of the application, it also add a lot of dynamicity in the interaction between the application code and the MVC framework code. Using SAST in this situation becomes just unfeasible.

To mitigate this problem, some studies have been conducted to enhance the testability of these MVC-based applications by developing specifications and vocabularies to capture application behavior that SAST often misses. In these studies, SAST tools are either updated to adapt to this behavior or new tools are created from scratch (e.g., [45, 139, 81]). Unfortunately, these progresses do not seem to be adopted by advanced SAST tools that are widely used by developers (mainly commercial SAST tools). This puts at risk the security of modern web applications strongly based on MVC framework.

In this chapter, we introduce a novel approach to transform an application's source code and disconnect it from the MVC framework. Our approach focuses solely on the application source code, making it applicable to both research and commercial SAST tools without requiring access to their internal implementations. The core idea is based on a two-phases process. First, our approach identifies problematic flows between the application and the MVC framework. Second, our approach transforms those flows from highly dynamic application-to-framework flows to static application-to-application flows, so to make them "understable" for SAST tools.

In our work, we focus on the PHP scripting language: unlike previous works, that have focused on JAVA and ASP.NET, PHP is still by far the most common language to develop Web applications (78% market share in 2022 [8]), and its frameworks are widely adopted by top websites.

We conducted two experiments to demonstrate the effectiveness of our approach in improving the source code coverage processed by SAST tools. Indeed, in both the experiments the SAST tools reported a significant higher number of security alerts. In the first experiment, we applied our approach against 10 known-to-be-vulnerable applications built on popular MVC frameworks for PHP (Laravel and CodeIgniter). Our transformations, performed manually by us, enable SAST to detect the expected vulnerabilities. In the second experiment, we automated the transformation process for 20 CodeIgniter projects from Github and Sourcecodester. SAST reported over 2000 new findings, including 826 previously unknown stored XSS vulnerabilities and 103 reflected XSS vulnerabilities affecting 18 out of the 20 projects analyzed. All affected projects acknowledged our findings following

responsible disclosure and CVEs were released.

The rest of the chapter is organized as follows. Section 5.2 presents background information on the MVC design pattern, static analysis tools and their efficiency in detecting injection vulnerabilities. A motivational example is presented in Section 5.3, highlighting one of the previously reported vulnerabilities. Section 5.4 discusses the motivation behind PHP frameworks and their prevalence in open-source projects. Our approach and the steps taken to disconnect the application from the framework are explained in Section 5.5. Finally, our experiments and results are presented in Section 5.6 and Section 5.7.

5.2 Background

5.2.1 MVC Pattern

The Model-View-Controller [58] (MVC) is a popular design pattern widely used to implement modern web applications. The pattern separates the user interface from the core application logic by using three main elements. The *model* (dedicated to the business logic) manages the data and the relation with the database. The *view* (dedicated to the UI logic) displays the data from the model and encapsulates the presentation by showing only the required attributes. Finally, the *controller* (which is the processing unit) responds to the user's requests and organizes the communication between the model and the view.

For example, in a typical e-commerce website, the model would define the product's properties while the view would be in charge of how the website should display the products. When a user wants to browse a specific item, the controller would receive the request with the product's ID, ask the model for the corresponding product (which will be retrieved from the database), and forward its data to the view to be formatted and displayed to the user.

This pattern is often adopted for its ability to easily organize large-size applications. In fact, the MVC pattern offers extremely loose coupling and higher cohesion between the three layers, which facilitate parallel and fast development [87]. Hao et al.[83] discusses in more detail the advantage of the separation between the three layers, and Majeed et al. [110] shows how this separation simplifies the development process, and creates scalable applications. Hapsari et al. [97] shows the advantages of MVC also to improve the reusability of code. In addition to that, the flexibility of the MVC design

makes the project easy to plan, maintain, and modify by the development teams.

On the negative side, Singh et al. [137] list some disadvantages of MVC. For example, the authors found that it is hard for programmers to understand the MVC architecture and that MVC can make the code deployment more difficult. To solve the usability problems and simplify the adoption of the pattern, many Web development frameworks encapsulate the MVC approach in an easier-to-use bundle (as we will describe in more details in Section 5.4).

From a testing perspective, things get more interesting. On the one hand, researchers (such as Wu et al.[155] and Freeman [73]) have shown how the modularity of MVC can facilitate unit testing. On the other hand, other studies noted the difficulty of understanding the data flow on the MVC pattern [137], which can introduce problems for testing tools that need to reason over large portions of an application (which is often the case for security testing). For instance Spring and Struts are popular MVC frameworks for JAVA, and different publications (e.g., JackEE in Antoniadis et al. [45], TAJ in Tripp et al. [147], and F4F in Sridharan el al. [139]) tried to address the testability of enterprise applications that use these frameworks by providing new framework modeling approaches that can help to better understand their data flow.

To the best of our knowledge, no research has been conducted to measure the impact of MVC on the testability of an application, nor the consequence of the adoption of this popular pattern on security testing.

5.2.2 SAST Tools and Testability Tarpits

Over the past 20 years, static tools have been largely used in industry to detect injection vulnerabilities. These occur when an attacker can input harmful data that misuses the system, often because input validation is lacking. For example, special characters can manipulate SQL queries (SQL Injection) [18] or inputs can include scripting tags that execute JavaScript on the client side (XSS) [17]. Static tools detect these vulnerabilities by tracing the flow of user input (source) to the endpoint (sink) without any form of sanitization.

Static analysis also comes with many limitations. Chapter 3 compiled a list of hundreds of code patterns that hinder the static analysis of web applications. These patterns capture specific code instructions, referred to as *testability tarpits* by the authors, that prevent SAST tools from reconstructing the correct data-flow of the target applications, thus impacting

the detection of severe injection vulnerabilities.

Among the different classes of tarpits presented in [36], one is related to the use of dynamic features that result in values that can only be computed at runtime (or that are difficult to compute otherwise). While it is not surprising that SAST tools need some form of approximation to deal with, for instance, indirect function calls, our goal is to show that these tarpits are at the core of how MVC is operating. In other words, while MVC might ease software development and maintenance, it introduces unavoidable roadblocks for full-program static analysis.

As an example, we will now briefly discuss four dynamic features in PHP and their related testability tarpits which are commonly associated with the MVC implementations that targeted with three commercial SAST tools Comm_1, Comm_2, and Comm_3. For each dynamic feature we illustrate some of the testability tarpits with their code snippets taken from the open source catalog [23] made available by [36]: each code snippet comprises a trivial XSS vulnerability (occurring at the line with the `echo` statement), useful to test whether a SAST tool is blocked by the tarpit or not.

[DF1] Function and Methods Variables. In PHP, it is possible to call a function whose name is stored in a variable (Pattern 76 in [23]). As seen in the code snippet, the variable (`$var`) holds the name of the function (in this case, "func"). Comm_1, Comm_3, and Comm_2 are unable to detect the vulnerability because they cannot identify the function in this code. Similarly, when the method name is stored in a variable (Pattern 82 in [23]), the same issue arises.

```
function func($x){echo $x;}
$var = "func";
$var($_GET['p1']);
```

[DF2] Class Name Variables. In this dynamic feature, a static tool is unable to determine which class will be used (Pattern 81 in [23]), so it cannot identify the properties or methods of the class. Comm_1 and Comm_3 cannot detect the vulnerability, but Comm_2 uses a heuristic. In this example, if the "func" method is defined only in one class, Checkmarx maps the method call with the method body. However, if multiple classes have the same name, Comm_2 ignores the method calls and is unable to detect the vulnerability.

```
class myclass{ function func($x){echo $x;}}
$var = "myclass";
$obj = new $var();
```

```
$obj->func($_GET['p1']);
```

[DF3] File Variables. In PHP when one file includes another file, all the variables from the external file are accessible from the internal file without defining the variables as globals. In this tarpit, Comm_1, Comm_3 and Comm_2 cannot detect the relation between File1 and File2 (Pattern 79 in [23]) and cannot discover the XSS vulnerability with the variable `$x`.

```
// File1 content
$x = $_GET['p1'];
$var = "File2.php";
include($var);
// File2 content
echo $x;
```

[DF4] Variable Variables. In differ from other programming languages, PHP has a special feature called variable-variable, when the name of the variable is in another variable. This feature gives a lot of dynamicity for PHP but it is not possible for Comm_1, Comm_3 and Comm_2 to follow the variable (Pattern 84 in [23]). Similarly, the property names can also be stored in a variable (e.g. `$obj->$var`).

```
$x = $_GET['p1'];
$var = "x";
echo $$var;
```

5.3 Motivation

While the MVC design pattern can be implemented in different ways, each implementation requires the use of many dynamic features to provide to the application enough flexibility to handle the relation between the models, views, and controllers. This dynamicity clashes with the operation of SAST tools, which need to be able to statically identify and analyze data-flow paths between sources (where users can inject information) and sinks (where that same information is used in a sensitive operation). Intuitively, we can therefore expect SAST tools to encounter many difficulties when tasked to analyze an application that uses the MVC pattern.

To show the impact on a practical example we investigated several CVEs (e.g., CVE-2014-1944, CVE-2014-6280, CVE-2015-2796, and CVE-2016-10509) reported for PHP projects that implement the MVC pattern. For instance, Listing 5.1 shows a simplified version of the XSS vulnerability

```
1  Env::executeAction(request_controller(), request_action());
2  static function executeAction($controller_name, $action) {
3    $controller = new $controller_name();
4    return $controller->execute($action);
5  }
6  function request_controller() {
7   return $_GET['c'];
8  }
9  function request_action() {
10  return $_GET['a'];
11 }
12 function execute($action) {
13  // call the method search
14  $this->$action();
15  $this->render();
16 }
17 function search() {
18  $search_for = array_var($_GET, 'search_for');
19  assign('search_string', $search_for);
20 }
21 function assign($name, $value) {
22  $this->vars[$name] = $value;
23 }
24 function render() {
25  $template_path = $this->getTemplatePath();
26  includeTemplate($template_path);
27 }
28 function includeTemplate($template) {
29  extract($this->vars, EXTR_SKIP);
30  // include the file search.php
31  include $template;
32 }
33 //search.php
34 echo $search_string;
```

Listing 5.1: CVE-2015-2796 - Example of an XSS vulnerability

from CVE-2015-2796. The user input is provided in the parameters of this request:

```
index.php?c=project&a=search&search_for=INPUT
```

where `c` represents the name of the controller and `a` the name of the action (the method in the controller). The vulnerable path traverses the connection between a controller and a view. The user chooses the controller and the method which will be called, then the view is loaded by using the search string provided by the user. Since this string is not sanitized, it leads to a XSS vulnerability.

In more detail, the vulnerability in Listing 5.1 begins with the call to the static method `executeAction` (line 1). This method retrieves the controller

and action names from the GET request (in this example, the controller name is "project" and the action name is "search"). It then creates a new instance of the controller and invokes the `execute` method, passing the action name. Within `executeAction`, we encounter the first dynamic feature: [DF2], previously described in Section 5.2.2. The `execute` function calls the action `search` (line 14), revealing the second dynamic feature [DF1].

The "search" method retrieves the "search_for" string from the GET request and stores it in the `vars` array using the `assign` method. The `execute` method then calls the `render`, which retrieves the template name (line 25) and invokes the `includeTemplate` method to include the template. This method uses the built-in `extract` function to load variables from the array, whose keys are used as the names of the generated variables. This results in the variable `$search_string`, which holds the value of the "search_for" input. The use of a variable to hold the name of another variable (the key of the array, in this case) introduces a third dynamic feature of type [DF4]. Finally, after loading the variables, `includeTemplate` includes the view file, which prints the input and triggers the XSS vulnerability. The file inclusion is performed using a filename stored in a variable, which represents the fourth dynamic feature [DF3].

This example shows how even a simple custom implementation of the MVC design pattern, which does not involve any complex MVC framework, requires many dynamic features that impede SAST testability. Just along the path of the aforementioned XSS we identify four tarpits which are very difficult for SAST tools to handle.

It is also interesting to notice how this vulnerability emphasizes the difference between static and dynamic testing solutions. In fact, while in the source code the path traverses 15 different functions invoked dynamically, the vulnerability could be easily discovered by using a black-box approach, just inserting a script in the search box.

5.4 MVC Frameworks

As emphasized by Singh et al. [137], the MVC design requires a good knowledge of software engineering and its development and deployment are difficult and time consuming. Thus, many Web Application Frameworks have been developed to aid programmers in creating and maintaining their applications.

Each framework provides a set of resources and tools, together with a standardized way to develop an application that programmers need to fol-

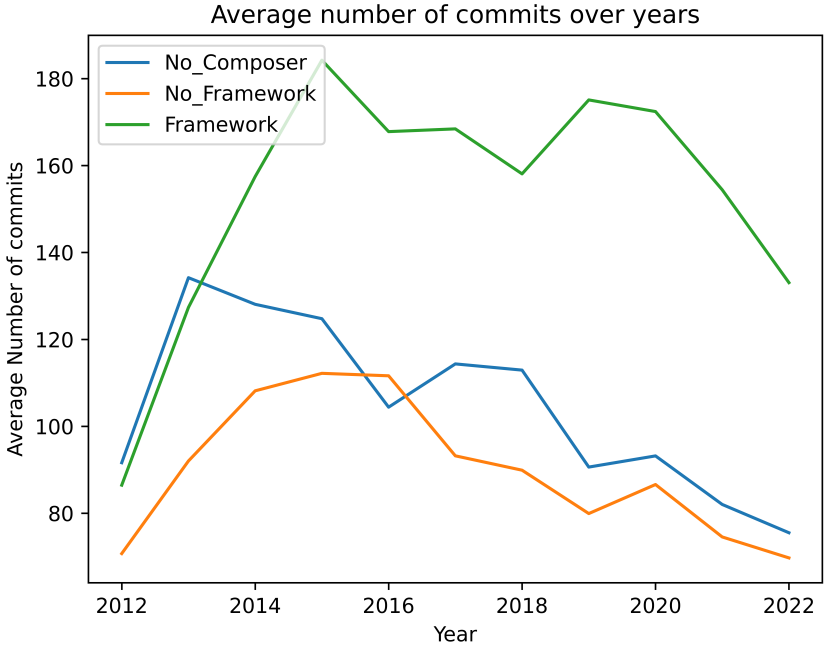

Figure 5.1: Number of commits over 22 years

low in their coding practices (like the name of the classes and the file/directory structure). In addition to that, they provide ready built-in features to handle common tasks, such as authentication, session management, transactional business logic, web application security, object relational mapping, and others.

PHP is one of the preferred languages for these frameworks because of its flexibility and its ability to support reflection and other dynamic features, as explained in [100] and [90]. For example, PHP can invoke functions and create new class instances from names stored inside a variable, as we saw in the tarpits presented in Section 5.2.2. As a result, PHP is the first language in the list of Framework Usage Distribution among the Top 1 Million websites [10], with its frameworks adopted by over 23.5% of the sites.

Most PHP Frameworks guide the development of web applications by following the MVC design pattern to provide rapid development, reduce time, create stable applications and reduce the amount of repetitive code. In

the annual Developer Ecosystem Survey conducted by Jetbrains in 2021 [24] on 31,743 developers from 183 countries, only 7% of PHP developers declared that they develop applications without any framework.

We identified 14 frameworks that are popular and used by PHP developers: CakePHP, CodeIgniter, Drupal, FatFree, FuelPHP, Joomla, Laminas, Laravel, PhalconPHP, YII, Phpixie, PopPHP, Slim, and Symfony. *All* these frameworks use the MVC design pattern.

5.4.1 Framework Prevalence

To study the prevalence of these frameworks we cloned all PHP projects on Github with more than 200 stars. Out of the resulting 5,270 projects, 3,853 included a `composer.json` file[1]. We searched in the composer files for the 14 frameworks, and found that 2,316 projects have at least one framework in the dependencies. Thus, we can conclude that more than 60% of the popular open source PHP projects that use a composer file are developed with one of the aforementioned 14 frameworks.

We also counted the number of commits on PHP projects over the last decade. Figure 5.1 shows the average number of commits between 2012 (composer released year) and 2022. The green and orange lines represent projects which have a composer file (with and without a framework dependency), and the blue line those without. The graph shows that after 2015 the average number of commits on projects developed by using frameworks is almost double compared with projects without framework.

5.4.2 Implementation

Figure 5.2 shows the different entities involved in a typical MVC implementation. The user sends requests to the controller. The controller then manages the communication between the view and the model to insert, update, fetch or delete the requested data. Finally, it displays the result back to the user. In the Figure we can also see the drivers that provide the implementation for different Database Management Systems (DBMS).

SAST tools struggle to handle this complexity as they are unable to reconstruct the relation between the different entities of the pattern. To understand the reasons, we analyzed the implementation of the MVC pattern in the latest version of four popular PHP frameworks: Laravel (v10),

[1] The PHP Composer `https://getcomposer.org/` is a tool for dependency management in PHP. It helps the developers to define the dependencies in their project and update them by running one command

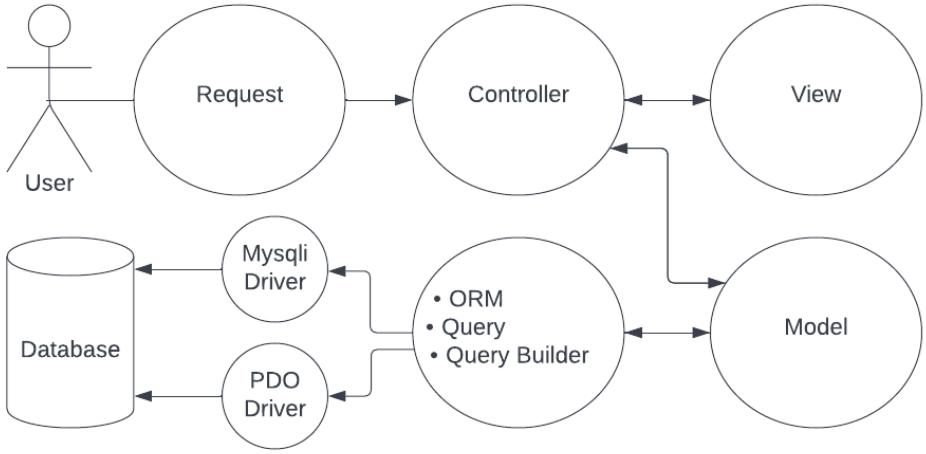

Figure 5.2: MVC Design Pattern

Symfony (v6), CodeIgniter (v4), and CakePHP (v4). As a guiding example, we are going to use a vulnerability we discovered in our experiments on the Kalkun project (an open source web-based SMS manager), which is built by using the CodeIgniter framework.

In the example, summarized in Listing 5.2, the attacker controls the `username`, which is stored un-sanitized in the DB. When the user logs in (lines 12-18), the username is retrieved from the DB and saved in the session. Then the system redirects the user to the `index` action in the `Kalkun` controller (line 17), which loads the `layout` view (line 22). Next, the `layout` view loads another view (`dock`, see line 25) where the username is extracted from the session and displayed in the page (line 27). This leads to a second order XSS vulnerability.

None of the SAST tools in our arsenal is able to detect that vulnerability. This is only partially related to the fact that the vulnerability is a *stored* XSS. Even when we decompose the stored XSS into (1) detecting that an attacker-controlled value is ending in a database element and (2) that element is retrieved from the database and printed, the SAST tools fail already in the first decomposed step, indicating that the data-flows of attacker-controlled inputs are very challenging for applications using MVC framework.

We will now detail the challenges for SAST emerging from the integration between the application and the MVC implementations. For each challenge we enumerate the dynamic features (cf. Section 5.2) that make

```
1  <?php
2  // Usermodel model
3  function addUser(){
4   $this->db->set('username',trim($this->input->post('username')));
5  }
6  // login controller
7  function index(){
8   $this->load->model('Kalkun_model');
9   $this->Kalkun_model->login();
10 }
11 // Kalkun_model model
12 function login(){
13  $username = $this->input->post('username');
14  $query = $this->db->from('user')->where('username', $username)->get();
15  if ($query->num_rows() === 1){
16   $this->session->set_userdata('username', $query->row('username'));
17   redirect('kalkun');
18  }
19 }
20 // kalkun controller
21 function index(){
22  $this->load->view('main/layout');
23 }
24 // main/layout view
25 $this->load->view('main/dock');
26 // main/dock view
27 echo $this->session->userdata('username');
```

Listing 5.2: XSS vulnerability in Kalkun project

SAST failing over some MVC implementations. More details, including a table summarizing how each challenge impacts the most popular MVC frameworks, are available in our repository [16].

[CM] Controller ↔ Model. The controller communicates with the model every time it needs to interact with the database. For instance, in Listing 5.2, the `login` controller loads the `Kalkun_model` and calls the `login` function to check the username (lines 8-9).

The loading of the model is implemented in different ways across the frameworks we analyzed. In our example, the CodeIgniter backend creates a new instance of the class name and assigns it to a property with the same name of the class:

```
public function model($model, $name){
 $CI =& get_instance();
 $model = new $model();
 $CI->$name = $model;
}
```

The CodeIgniter backend code is thus impacted by two dynamic features: the class-name-variable [DF2] and the property-variable [DF4]. The same PHP dynamic features impact also Symfony and CakePHP. The only exception is Laravel, where the model loading relies on static methods only, and therefore it can be handled correctly by SAST tools.

[MDB] Model ↔ Database. The model communicates with the database whenever it needs to fetch, add, update or delete information. In our example of Listing 5.2, the `Usermodel` model contacts the database to set the username (line 4) and the `Kalkun_model` model retrieves the user data from the database (line 14).

An advantage provided by all frameworks is the transparent support for multiple Database Management Systems (DBMSs), which is handled by a separate driver for each database (e.g., PDO, CUBRID, and Mysqli). The model communicates with the drivers by running a query, query builder, or object-relational mapper (ORM). The name of the driver is set in the configuration file for the whole project, and a new instance of this driver is created accordingly. This is implemented by keeping the name of the driver in a variable, which leads to a class-name-variable dynamic feature [DF2], hereafter illustrated for the CodeIgniter backend:

```
// Instantiate the DB adapter
$driver = 'CI_DB_'.$params['dbdriver'].'_driver';
$DB = new $driver($params);
```

[CC] Controller ↔ Controller. A controller can redirect the request to a different action in the same controller or to another controller. In our example, after a successful login a user is redirected to the `index` method in the `kalkun` controller (line 17).

All frameworks store the name of the class and the method to invoke as strings, which lead to the dynamic features method-variable [DF1], class-name-variable [DF2], and variable-variable [DF4]. For instance, in CodeIgniter, we observe the redirect function utilizing the controller class name and method name.

```
redirect("ClassName","MethodName");
```

[CV] Controller ↔ View. After the controller receives the user request and communicates with the model, it loads the view and passes the requested data to this view. In our example, the code loads the page `main/layout.php` (line 22).

All four frameworks use a similar approach to load a view, delegating

the operation to a function that receives the view name as parameter, leading to a File-Variables dynamic feature [DF3]. In Laravel, Symfony, and CodeIgniter the function also receives an array of variables that is later passed to the view, while CakePHP adds each variable separately through the `set` method. Either way, the name of the variable to pass will be stored into another variable, which leads to variable-variables [DF4]. For example, if we look at the code of CodeIgniter, we can see that the method `view` receives the name of the file and the data, then passes them in an array to the function `_ci_load`. This function generates the names in new variables, introducing a variable-variables [DF4]:

```
protected function _ci_load($_ci_data){
 foreach (array('_ci_view', '_ci_vars') as $_ci_val){
  $$_ci_val = $_ci_data[$_ci_val];
 }
}
public function view($view, $vars){
 return $this->_ci_load(array('_ci_view' => $view, '_ci_vars' => $vars));
}
```

[VV] View ↔ View. Usually, website pages share a common style and design. To avoid replicating code among the different views, frameworks allow the developer to create subviews, which can be reused and included into others. For example, the `main/dock.php` is loaded in `main/layout.php` (line 25 in Listing 5.2).

Symfony uses Twig templates to handle views [25] and Laravel uses Blade templates [3]. These templates define a specific language for the view files. This requires SAST tools to interpret those specific languages that is mainly an engineering effort. This is out-of-scope in our work that instead targets the PHP language.

In CodeIgniter, similar to the controller-view relation, the controller can load multiple views or load the content of one and then pass it to another view inside a variable. As such, this [VV] in CodeIgniter has to deal with the File-Variables [DF3] and variable-variables [DF4] dynamic features. Instead, CakePHP divide views into templates, elements, layouts, helpers, and cells. All the relations between these entities in CakePHP use dynamic constructs. Thus, extending the view with another view adds both a file-variable [DF3] and a variable-variable [DF4]. While passing the name of the method as a string in the cell will experience the method-variable [DF1], and loading the helper face the class-name-variable [DF2].

[I] Input Management. Frameworks provide classes to deal with requests and responses and to get information from `POST` and `GET` parameters, files,

environment variables, external servers, and cookies. For instance, in our example (Listing 5.2) the username is sent over a `POST` request (line 4 and 13).

Both Symfony and Laravel provide a `Request` class that uses `InputBags` containers to store the user inputs. The `InputBags` implementation, in turn, introduces a function-variable [DF1]:

```
static function createRequestFromFactory($args){
  $request = (self::$requestFactory)($args);
  return $request;
}
$request = self::createRequestFromFactory($_GET, $_POST, [], $_COOKIE, $_FILES,
      $_SERVER);
```

CakePHP defines instead request classes as components, whose loading procedure introduces a property-variable [DF4] and a class-name-variable [DF2]. On the other hand, in CodeIgniter, the input property is loaded in the constructor of the controllers' parent (`CI_Controller`), whose code contains a property-variable [DF4]:

```
foreach (is_loaded() as $var => $class){
 $this->$var =& load_class($class);
}
// then we can access the POST through
$this->input->post('param_name');
```

[S] Session Management. Frameworks provide an implementation of sessions. The session is part of the request in Laravel and Symfony, while a system library must be loaded in CodeIgniter and a component in CakePHP, resulting in a class-name-variable [DF2] and property-variable [DF4]. We can see the usage of the session in our example when we save and retrieve the username (line 16 and 27).

5.5 Our Approach

In the previous section, we showed that it is impossible for static tools to detect connections between an application's source code and MVC frameworks. Using these tools to scan an application's code relying on a MVC framework is ineffective and a waste of resources as we will further demonstrate with experimental evidence in Sections 5.6 and 5.7. In this paper, we present a novel approach to enable the SAST tool to capture the flow between the application and the MVC frameworks by transforming the application's code (only for the purpose and timeframe of the SAST analysis) and without modifying the SAST tool. Our approach differs from previous

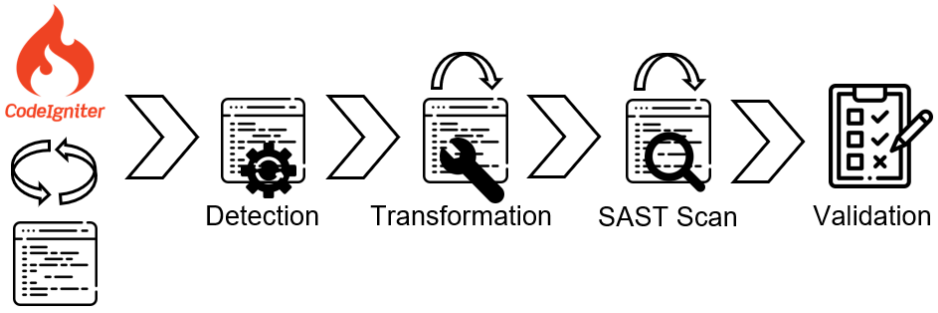

Figure 5.3: Our Approach

works that require changes to the static tool (Sridharan et al.[139]) or the creation of a new ad-hoc SAST tool (Antoniadis et al.[45]).

Chapter 3 proposed a one-to-one solution for their testability tarpits. However some of these tarpits must be resolved manually by the developers. Transforming the highly dynamic MVC framework's code therefore would require a comprehensive understanding and rewriting of the entire framework and its dependencies. In this chapter, we propose a new transformation called "MVC design transformation" that involves understanding the main tasks of an MVC framework and providing alternative code to achieve the same task without using the framework's code.

Our approach is illustrated in Figure 5.3. It takes the application source code as input, discovers the MVC challenges in the application, applies the transformations to disconnect the application from the MVC framework, scans the transformed code using a static tool, and presents the findings to the user for (manual) validation (e.g., classify true positives vs or false ones). Cutting the connection between the application and framework reduces the number of testability tarpits and improves the coverage of the source code. However, there may still be testability tarpits, as part of the application itself and not related to the MVC framework, that developers can handle by applying the approach proposed in [36].

In the rest of this section, we illustrate our approach by showing concrete transformations for the latest version (i.e., v4) of the CodeIgniter framework. However, our approach can be implemented for any other MVC implementations. In our repository, we detail the implementation of our approach also for CodeIgniter v3 and Laravel v10 (latest). Implementing the transformations for other MVC frameworks follows a similar process and it just a matter of allocating development time. From our own experience, we

estimate this development time to one week overall for one intermediate-level developer to comprehend the challenges in the framework, create the detection rules, perform the transformation, and test their implementation.

5.5.1 Detection

Before applying any transformation, our approach detects in the application the code areas that need to be transformed, i.e., those areas impacted by the MVC challenges discussed in the previous section. We use the PHP Opcode, the low-level instructions that are processed by the PHP Zend engine [15], to detect the areas to transform and the important values for the transformations.

For example, the following presents the PHP and the corresponding opcode for loading the model in CodeIgniter:

```
/* PHP Code */
$this->load->model("product","prod");
/* PHP Opcode */
T0 = FETCH_OBJ_R THIS string("load")
INIT_METHOD_CALL 2 T0 string("model")
SEND_VAL_EX string("product") 1
SEND_VAL_EX string("prod") 2
DO_FCALL
```

To detect this operation, our approach searches in the opcode for the operation that fetchs an object to read (`FETCH_OBJ_R`) through the method `load`. Then, it determines the method call `model` from this object (`INIT_METHOD_CALL`). The names of the model class and of the property to load are then extracted from the `SEND_VAL_EX` operations and used to provide static values to the related trasformation.

5.5.2 Transformation

Once the code areas with MVC challenges are identified, our approach transforms them to remove their dynamic features and to make the application code more testable for SAST.

Hereafter we present the transformations for the MVC challenges impacting CodeIgniter. (For each transformation we specify which MVC challenge(s) the tranformation remediates by means of the following notation *transformation -> challenge(s).*)

Load Model -> [CM]. Our transformation creates a new instance of the model directly and assigns it to the related property name.

```
/* Before */
$this->load->model("product","prod");
/* After */
$this->prod = new Product();
```

Load View -> [CV] and [VV]. We change the variables that will pass to the view to be global variables and include the related PHP file.

```
/* Before */
$data["var1"] = "val1";
$data["var2"] = "val2";
$this>load->view("display",$data);
/* After */
$GLOBALS["var1"] = "val1";
$GLOBALS["var2"] = "val2";
include "display.php";
```

Database -> [MDB]. We provide one database class instead of all the classes and drivers that deal with the model. This class contains all the methods that models need to communicate with the database. Then, we categorize the methods into two categories. (1) Methods insert data in the database, we change the body of the method to print the inserted values. (2) methods get data from the database, we change the body of the method to return a GET input (source). This implementation helps us detect second-order vulnerabilities, as we will explain in the next section.

```
/* Before */
$this->db->insert("table_name",$object);
$object = $this->db->get($id);
/* After */
$this->db = new database();
$this->db->insert("table_name",$object);
$object = $this->db->get($id);

class database{
    public function insert($table,$object){
        echo $object;
    }
    public function get($id){
        return $_GET['p1'];
    }
}
```

Redirect to a Controller -> [CC]. In this transformation, we create a new instance of the controller then we call the method of this controller.

```
/* Before */
```

```
redirect("ProductsController","choose");
redirect("ProductsController");
/* After */
$product = new ProductsController();
$product->choose();
$product = new ProductsController();
$product->index();
```

Input Management -> [I]. We transfer the inputs (Post, Get, Cookie, and Server) for direct inputs in PHP. For example to transform form the GET parameter.

```
/* Before */
$var = $this->input->get('p1');
/* After */
$var = $_GET['p1'];
```

Session Management -> [S]. Replacing the session methods with global variables helps static tools solve the difficulties in understanding the sessions in CodeIgniter framework.

```
/* Before */
$this->session->set_userdata('var_name', 'value');
$var = $this->session->userdata('var_name');
/* After */
$GLOBALS['var_name'] = 'value';
$var = $GLOBALS['var_name'];
```

5.5.3 SAST scan

After separating the application source code from the framework, our approach runs the SAST tool to scan the transformed source code. It shall be noted that the transformation that our approach performs is only for the purpose of having a better static analysis and it is not intended to be persisted on the project code as it would contrast the MVC benefits.

The SAST tool reports any unsanitized input flows from source to sink that it identifies. In doing so, it often produces many false alerts due to over-approximation strategies. The alerts reported by SAST need thus to be carefully validated.

5.5.4 Validation

Finally, we validate our approach by manually examining the alerts generated in the previous step. A larger number of alerts indicates that the

tool can cover a larger space and identify better data-flow connections in the source code. Detecting more exploitable vulnerabilities (true positives) proves the efficacy of our approach and demonstrates that commercial static tools may fail to identify high-severity vulnerabilities when the application relies on a framework.

In evaluating vulnerabilities, we concentrate on both first-order (direct path from source to sink) and second-order (two paths, from source to database and from database to sink) vulnerabilities, as outlined by Dahse et al. [66]. The second-order vulnerabilities assess the impact of controller-model and model-database challenges, while both types measure the impact of other MVC challenges.

None of the static tools in our study are capable of detecting second-order vulnerabilities, as they require parsing database queries and linking stored and retrieved data. To overcome this, we modify the database class by inserting a sink and retrieving a source, as described in the [MDB] transformation. This allows us to manually validate the alert from the database to the sink. If the alert is a true positive, we then match and validate the path from the source to the database.

Our approach does not result in an increase in false positive rate for first-order vulnerabilities despite the SAST over-approximations. However, increased false positives may occur in the second-order when the source is sanitized prior to being stored in the database.

On the other hand, developers are advised to apply sanitizers at the sink as discussed in [30]. This is because the input may change along the path, potentially rendering the sanitizer useless, or because the same input may reach different sinks which require different sanitizers. Although CodeIgniter previously offered a feature to sanitize all project inputs, it was removed in version 4 as it was deemed a bad idea [31].

5.6 Experiment: Manual MVC transformations

In this section, we will demonstrate the usage of our SAST-enabling transformations against 10 known-to-be-vulnerable applications based on two popular MVC frameworks, Laravel and CodeIgniter. The goal is to show that our transformations, manually applied on the vulnerable paths of the known-to-be-vulnerable applications, enable SAST tools to detect the expected vulnerabilities.

MVC frameworks. We selected Laravel and CodeIgniter because of their popularity. Each of them is used in over 1.5 million of enterprise applica-

tions [13, 7].

Static Tool. We selected the commercial SAST tool Comm_1 for our study. Our decision is based on two elements. First, the recent results in [36, 121] demonstrate that commercial tools have significantly higher support than open-source tools over SAST challenges for PHP. Second, by running three commercial static tools in our arsenal (Comm_1, Comm_2, and Comm_3) against the publicly available benchmark of [23], Comm_1 reported the better results.

Vulnerability Type. We chose XSS for our experiment as it is the most prevalent injection vulnerability and ranks second on the top 25 most dangerous software weaknesses [1]. We are taking into account both reflected and stored XSS.

Dataset. We target known-to-be-vulnerable projects by selecting 5 CVEs related to Laravel and 5 to CodeIgniter. We first look at projects developed for those MVC frameworks [12, 6], and we then search for reported XSS vulnerabilities in those projects in the CVEs database `https://cve.mitre.org/`.

Methodology. For each CVE in our dataset, we first fetch the source code associated with the CVE report. Second, we inspect the XSS vulnerability of the CVE and we identify the XSS vulnerability path in the code, so to clearly extract from that path both (i) the MVC challenges and (ii) the traditional tarpits (see [36]) in the project unrelated to MVC that may block SAST. Third, we run the SAST tool (Comm_1) against the code to confirm that the expected vulnerability is not reported. (Actually, no XSS findings are reported at all by the SAST tool, confirming the complexity of the MVC-based code). Fourth, we apply our MVC related transformations over the XSS vulnerability path, so to remediate the MVC challenges previously extracted and when necessary we also perform remediations for the additional application tarpits unrelated to MVC. Finally, we run the SAST tool against the transformed code and check whether the expected XSS vulnerability is now properly reported.

Results. Table 5.1 shows the ten CVEs in our dataset. For each CVE, we present whether it refers to first or second order XSS, the MVC framework and version, and the MVC challenges and the extra application tarpits (not related to MVC) that we remediated to detect the expected XSS vulnerability. More details, including the vulnerability path of these CVEs, are in our repository [16].

All the vulnerabilities have an MVC-related input management challenge, and all the stored XSS involve controller-model and model-database

CVE	MVC	CM	CV	VV	MDB	I	AT
CVE-2022-41938$^{\bowtie}$	LA v8	✓	✓		✓	✓	2
CVE-2021-27371$^{\bowtie}$	LA v7	✓	✓		✓	✓	
CVE-2019-15489$^{\triangleright}$	LA v5.7					✓	
CVE-2018-20962$^{\bowtie}$	LA v5	✓	✓	✓	✓	✓	
CVE-2018-19917$^{\triangleright}$	LA v5		✓			✓	
CVE-2020-26043$^{\bowtie}$	CI v3	✓	✓	✓	✓	✓	
CVE-2022-28586$^{\bowtie}$	CI v3	✓	✓		✓	✓	
CVE-2018-16772$^{\bowtie}$	CI v3	✓			✓	✓	
CVE-2019-7223$^{\bowtie}$	CI v3	✓	✓		✓	✓	2
CVE-2018-12255$^{\bowtie}$	CI v3	✓	✓		✓	✓	2

$^{\triangleright}$: 1st order XSS, $^{\bowtie}$: 2nd order XSS
LA: Laravel, CI: CodeIgniter, AT: extra application tarpits
note: [CC] and [S] are not reported as no CVE included that.

Table 5.1: Previous Reported Vulnerabilities

relations. All the discoveries end in view files where we can observe controller-view relations, except for CVE-2018-16772 where the sink is in a controller file. Only 2 discoveries (CVE-2018-20962 and CVE-2020-26043) have view-view relations, and no controller-controller relation nor session management were observed.

By applying our MVC transformations (see Section 5.5.2), Comm_1 detects the expected vulnerabilities for 7 out of 10 CVEs. Despite remediating MVC challenges, the remaining 3 CVEs still have application-level testability tarpits unrelated to MVC that block Comm_1 from detecting the bugs. Once we address two extra tarpits (Patterns 38 and 81 in [23]) in CVE-2022-41938, and two different extra tarpits (Patterns 17 and 79 in [23]) in both CVE-2019-7223 and CVE-2018-12255, the static tool can detect all the expected vulnerabilities.

5.7 Experiment: Automated MVC transformations

In the prior section, we demonstrated the effectiveness of our approach in helping SAST tools to discover known vulnerabilities in 10 projects run by two popular MVC frameworks. In this section, we assess our approach on a dataset of 20 MVC-based applications where the ground truth is unknown, i.e., we do not know if these applications are vulnerable or not. The goal is to determine if by applying in an automated manner the transformations of our approach, we can still help SAST tools to improve the coverage of their testing analysis. The main metrics we can use in this respect are (i) the

number of new findings reported by the SAST tools and (ii) the detection of any previously unknown vulnerability.

Summary. More than 2000 new findings were reported by SAST after our approach transformed the code. Among these findings there are 826 previously unknown stored XSS and 103 new reflected XSS. These severe issues are impacting 18 of the 20 projects we analyzed. Upon responsible disclosure of our vulnerabilities, all the affected projects acknowledged our findings and CVEs were (or are about to be) released.

Settings. In this experiment, we apply our automated approach to CodeIgniter framework v3 and v4, using Comm_1 as a static tool and XSS as the vulnerability type. We selected the top 10 most popular projects based on CodeIgniter from its Github library [6] (excluding libraries and plugins), and 10 projects based on CodeIgniter from Sourcecodester, a website where developers share source code for various projects and services in PHP. The projects from Sourcecodester include websites that provide services like a bookstore website, expense management system, hotel management system, and a blog site. We used these 20 projects to assess our approach.

Results. Table 5.2 More than $2,000$ alerts were reported at the view files: upon validation this resulted in 774 true positive stored XSS (from the database to the sink) and 100 true positive reflected XSS (from source to sink); $1,151$ alerts were false positives. 93 alerts were reported for the controller files: 52 true positive stored XSS and 3 true positive reflected XSS; 38 alerts were false positives. We provide additional information regarding our discoveries and their vulnerable paths in our repository [16]. On the other hand, we didn't find any vulnerabilities in two projects. The Online Exam System project had all inputs sanitized, and the Pagination Tutorial website, being a simple tutorial site, only had one source that didn't reach a sink.

Responsible disclosure. We reported the new discoveries to the developers and created pull requests to fix some of them. Our findings were acknowledged and we obtained CVEs for all the vulnerable projects, but Mapos for which the developers approved our pull request to fix the vulnerability and we are in the process to get the CVE. Our findings reveal hidden bugs by the MVC components, which pose a challenge for static tools. For example, as seen in the Kalkun project 5.2, the relationship between the MVC components, such as controller-controller, input management, controller-model, controller-view, model-database, view-view, and the session management in CodeIgniter, result in at least seven testability tarpits that make very difficult for SAST to explore the data-flow that could

	Github Project	Stars	Views_FP	Views_TP	Ctr_FP	Ctr_TP	CVE
1	Wscats-cms	251	3	12	0	8+(1)	CVE-2023-23016
2	Corn-manager	406	0	0+(2)	0	0	CVE-2023-23017
3	Kalkun	182	15	1	2	0	CVE-2023-23015
4	InventorySystem	172	0	0	0	27	CVE-2023-23014
5	hr-payroll	137	462	73	36	14	CVE-2023-23013
6	classroombookings	127	0	4	0	0	CVE-2023-23012
7	InvoicePlane	2.1K	7	5+(13)	0	0+(2)	CVE-2023-23011
8	Ecommerce Bootstrap	1K	68	57+(25)	0	0	CVE-2023-23010
9	Mapos	615	300	261	0	0	Confirmed
10	Sales Management System	375	81	30+(22)	0	0	CVE-2023-23018
	Sourcecodester Project	Views					
1	Book Store	5.2K	3	47	0	3	CVE-2023-23024
2	Expense Management System	4.3K	45	9	0	0	CVE-2023-23027
3	Sales Management System	4.8K	38	18	0	0	CVE-2023-23026
4	Hotel System	5K	31	22	0	0	CVE-2023-23025
5	Laundry System	2.7K	6	163	0	0	CVE-2023-23023
6	Employees Payroll	5.8K	36	59+(23)	0	0	CVE-2023-23022
7	Point Of Sale	4.5K	28	7+(6)	0	0	CVE-2023-23021
8	Online Exam System	5.9K	65	0	0	0	-
9	Blog Site	2.8K	36	6+(9)	0	0	CVE-2023-23019
10	Pagination Tutorial	776	0	0	0	0	-
	SUM		1151	774+(100)	38	52+(3)	18

Views_FP, Views_FP: Alerts (true or false positive) at view file.
Ctr_FP, Ctr_TP: Alerts (true or false positive) at controller file.
XX+(YY): XX the number of second order vulnerabilities, YY the number of first order vulnerabilities.

Table 5.2: New Vulnerabilities Detected with Our Approach

lead to a vulnerability.

5.8 Related work

Challenges for Static Tools. Al Kassar et al. [36] is a recent paper that discusses in detail the difficulties for static tools in handling some code practices in PHP and JS, and they suggest some types of transformations that can increase the coverage of the source code. While Landman et al. [101], Yue et al. [102], Fourtounis et al. [72], and Sui el al. [142] discussed the challenges for static tools with reflection and dynamic features in JAVA. Other papers work on improving static tools like Livshits et al. [105] presents an automatic probabilistic approach for inferring explicit information flow specifications from the source code. Alhuzali et al. [40] and Chandra et al. [61] combines the static and dynamic approaches to capture the explicit data and control flows. In addition to that, Allamanis et al. [43] present in their survey different papers in machine learning for structured prediction in different programming languages like Gu et al. [77] who use deep learning to predict the sequence of API calls and Raychev et al. [131] who predict program properties from massive codebases.

In our research, we complement this line of research by providing an

alternative method to address the challenges of SAST by offering a design transformation that eliminates a series of MVC challenges instead of addressing each one individually.

Static Tools with frameworks. Different papers discussed the relations between the application source code and frameworks in several programming languages. Researchers suggest creating a specification like Jaspan el al. [93] which presents Fusion language for the specification, and Bastani el al. [52] presents ATLAS that infers points-to specifications. Sridharan et al. [139] present a standardized language for the specification (WAFL), then they add the WAFL specification support for ACTARUS static analyzer [81]. While writing the WAFL generator is far from a trivial task as they mention in Antoniadis et al. [45]. In addition to that, Ball et al.[50] present an approach of creating a boolean API specification from several programs that use this API, and Aldrich et al. [37] use the PLAID language to purpose the typestate-oriented programming where classes are modeled in terms of their changed states.

Other works build static tools for enterprise applications that are developed by different frameworks as we can see in JakeEE and TAJ in Antoniadis et al. [45] and Tripp et al. [147]. Both of the static analyzers use framework modeling for JAVA frameworks (like Spring and Struts). Concerto, in Toman et al. [144], is a framework-based applications which combines concrete and abstract interpretation. Arzt et al. [47] presents FlowDroid which is a tainting analyzer for android applications that need to handle callbacks invoked by the Android framework. And Paramitha et al [127] built a static tool for Laravel framework in PHP. In this tool, they use tainting analysis on the AST that is generated from the application's source code. Unfortunately, this static tool is not published and they evaluated their approach only on 7 old reported CVEs. Another commercial static tool for Ruby and Rails is Brakeman Pro [4] which became part of Synopsys in 2018. This work delves into the frameworks and highlights the dynamic features, which have been briefly discussed in previous studies [139, 102].

Our solution is implemented at the application level and does not require any modifications to static tools, giving developers the flexibility to use any static tool of their choice.

5.9 Conclusion

In this chapter, we propose a novel approach to transform the source code of an MVC-based application so to "disconnect" it from the MVC framework.

Unlike previous works that aim to adapt SAST tools to understand the dynamic features occuring in this MVC setting, our solution operates at the source code level and enables to perform the analysis without changing the SAST tools. As proof of concept, we implemented our approach for popular PHP MVC frameworks (CodeIgniter and Laravel), used by millions of web applications.

Our experiments were conducted on 10 known-to-be-vulnerable applications and against the latest version of 20 applications served by the popular CodeIgniter MVC framework. The results demonstrate that our transformations enables SAST tools to detect the known vulnerabilities. In addition, 929 new high-severity zero-day vulnerabilities were uncovered by our approach, including 826 stored XSS and 103 reflected XSS. We responsibly disclosed all of our findings, that were properly confirmed and acknowledged with CVEs.

5.10 Data Availability

In our Github repository [16], we provide all resources related to our work. This includes 5 main elements. First, a table that details the MVC challenges in all the MVC implementations of Laravel, Symfony, CodeIgniter, and CakePHP (an overview of this was discussed in Section 5.4.2). Second, an explanation of the transformations for the latest version of Laravel (similar to the CodeIgniter transformations we described in Section 5.5.2). Third, our implementation to detect and transform the MVC challenges in CodeIgniter (v3 and v4) and Laravel (v10). Fourth, the details (vulnerable paths) of the old reported CVEs that we discussed in Section 5.6. Fifth, the details (e.g., vulnerable paths) of the novel vulnerabilities that we discovered with our approach and that we overview in Section 5.7.

Chapter 6

Conclusion and Future Work

6.1 Future Works

While we have defined a list of testability patterns that we checked systematically in Chapter 3, developers and researchers may encounter new patterns when working with different programming languages. To have a greater impact in the research community, a framework that standardizes testability patterns and their discovery rules could be created. This framework could be integrated into IDEs such as Eclipse and NetBeans, which would allow developers to identify the lines of code that selected static tools cannot comprehend. This approach could encourage security testing during the development process, reducing testability tarpits and improving the coverage of static tools, thus mitigating the risk of undetected vulnerabilities.

Chapter 4 introduces WHIP, a groundbreaking approach that facilitates collaboration among SAST tools by sharing information to surmount their limitations. By exclusively operating on the application source code, our technique employs different tools as an oracle to detect signs of disrupted data flows. This technique is applicable to other forms of collaborations, such as dynamic tools with static tools or static tools combined with human interaction. Such solutions can resolve new testability tarpits that static tools are incapable of detecting and address challenges that are beyond the scope of dynamic approaches or human interaction, but can be resolved by static tools.

Finally, Chapter 5 is devoted to the analysis of the MVC frameworks and their interaction with the application source code. We explore the challenges that SAST tools encounter when attempting to comprehend this interaction, and we propose a solution to disconnect the application from the framework. This solution is generalizable and can be applied to cover all the frameworks and third-party libraries that are typically employed in applications. By severing the connections between the application and its dependencies, we can significantly reduce scanning time, enhance the detection of vulnerabilities, and minimize the risk of encountering testability tarpits.

6.2 Conclusion

This thesis provides evidence that certain code patterns, referred to as testability tarpits in Chapter 3, present a significant challenge for static analysis of real-world web applications. Our research involved assembling a library

of testability tarpits for the two most commonly used web programming languages (PHP and JS), which we evaluated using a combination of state-of-the-art open-source and commercial SAST tools. This library resulted in the development of a new OWASP project aimed at creating a community focused on testability tarpits [19]. Then, we established discovery rules for these tarpits and applied them to thousands of open-source applications, revealing that these tarpits are widespread, indicating that static analysis has numerous blind spots. Furthermore, we conducted two sets of experiments that demonstrated that refactoring the code to remove tarpits had a substantial impact on the alerts generated by SAST tools, resulting in the identification of numerous previously unknown vulnerabilities.

In Chapter 4, we proposed a novel approach to address the high prevalence of tarpits in web applications, which involves "forcing" different SAST tools to collaborate in order to detect more vulnerabilities. While each static tool has its own strengths and weaknesses, our solution enables them to work together to overcome their respective limitations. Our fully automated approach treats all tools as black boxes, making it compatible with commercial tools for which we have no visibility into their internal data structures. By periodically modifying the source code of the application under test, our system inserts mock sinks to trace how tainted values spread across various program functions. When one of the tools is unable to interpret these connections, we assist it by linking the data flow with additional edges that circumvent the problematic function.

Finally, in Chapter 5, we present a novel approach to transform the source code of an MVC-based application to "disconnect" it from the MVC framework. Unlike previous works that focus on adapting SAST tools to understand the dynamic features present in this MVC setting, our solution operates at the source code level and enables analysis without modifying the SAST tools. To demonstrate the effectiveness of our approach, we implemented it for popular PHP MVC frameworks (CodeIgniter and Laravel) that are used by millions of web applications.

We believe that the findings and solutions presented in this thesis can be a valuable foundation for future research in this field and provide guidance for both academics and practitioners to address the limitations and the solutions of SAST tools.

Appendices

.1 Appendix A

.1.1 Testability patterns for PHP

The testability patterns for PHP are presented in Table 1. This table groups together (by using horizontal lines) pattern instances which address similar aspects of the language and have the same response from SAST tools. For each instance (tarpit), the table reports its name, its properties with respect to the dimensions introduced in Section 3.3.1, and the tools that are affected by it (by using a sequence of letters, R for RIPS, S for PHPsafe, W for WAP, P for Progpilot, X for Comm_1 and Y for Comm_2). When a tool handles the pattern by means of an over-approximation, we mark its name with an overline. For instance, the string $----X\overline{Y}$ means that a tarpit is handled correctly only by Comm_1 and via over-approximation by Comm_2. The last fours sets of columns report the prevalence of each pattern instance in our four datasets – as expressed by the number of affected projects (*prj* column) and by the median number of occurrences of the pattern (*med* column).

Finally, when the same pattern has multiple instances (e.g., to describe tests belonging to different dynamic categories) that lead to the same result, we group them and report their number in the number of instances ($\#i$) column.

Table 1: Patterns

ID	Pattern	#i	API	SEC	Dyn	OOP	Neg	Tools	SC		G_L		G_M		G_H	
									prj	med	prj	med	prj	med	prj	med
1	static_variables	1			S			------	50	13	443	4	635	7	712	14.0
2	global_variables	1			S			−S−−XY	89	10	203	7	213	10	210	12.0
3	global_array	1			S			−−WP−Y	33	6	138	4.0	162	5.0	179	7
4	conditional_assignment	1			S			R−WPXY	221	74	795	18	890	31.5	908	59.0
5	combined_operator	1			S			R−WPXY	335	170	919	33	942	64.0	934	97.5
6	coalesce	1			S			RS−−XY	0	0	0	0	280	6.0	433	11
7	string_arithmetic_operations	1		✓	S			RSW−XY	277	10	523	6	636	9.0	707	11
8	simple_reference	1			S			----X−	42	39	163	9	231	5	292	6.0
9	reference_argument	1			S			----XY	208	14	387	7	399	10	486	9.0

10	return_by_reference	1			S			-------	19	11	83	4	95	4	132	4.0
11	foreach_with_reference	1			S			-------	41	7	182	3.0	238	4.0	321	4
12	make_ref	2			S		✓	---P-$\overline{Y}$	25	6	116	6.0	134	4.0	180	4.0
13	assign_static_prop_ref	1			S			---PX-	9	1.0	19	1	10	1.0	17	1
14	object_assigned_by_reference	1			S			----X-	22	21	100	6.5	107	7	155	5
15	nested_function	1			S			-SWPX-	66	3.5	166	4.0	222	4.0	283	5
16	variadic_functions	1			S			----XY	1	12	59	3	143	3	239	3
17	get_arguments	1			S			R----Y	23	5.0	106	4.0	137	4	191	4
18	send_unpack	1			S			RS--X-	1	31	73	3	146	3.0	264	4.0
19	closures	2			D2			----XY	36	1	543	7	733	11	782	25.5
20	use_with_closures	2			D2		✓	----X$\overline{Y}$	25	1	321	4	524	5.0	614	12.0
21	simple_object	1			S	✓		---PXY	336	199	968	350.5	977	863	974	1536.5
22	assign_object	1			S	✓		--W-X-	30	3	138	4.0	212	4.0	325	4
23	object_argument	1			S	✓		----X-	119	30	591	23	718	53.5	804	79.5
24	new_self	1			S	✓		----X-	41	6	162	2.0	249	3	351	3
25	clone	1			S	✓		---PX-	41	6	147	3	238	5.0	338	5.0
26	late_static_binding	2			D2	✓	✓	----$\overline{X}$-	13	1.0	165	4	279	5	386	8.0
27	get_called_class	1			D2	✓		-------	0	0	16	1.0	28	1.0	34	2.0
28	static_methods	1			S	✓		---PX-	119	17	792	29.0	865	61	898	126.5
29	static_properties	1			S	✓		---PX-	93	48	406	12.0	498	14.0	615	20
30	anonymous_classes	1			S	✓		----XY	1	6	30	2.0	81	3	174	3.0
31	static_method_variable	2			D2,D4	✓		-------	1	1	23	1	51	2	68	2.0
32	set_overloading	1			S	✓		----X-	0	0	44	6.0	50	7.0	61	8
33	get_overloading	1			S	✓		-------	1	1	56	6.5	70	7.0	81	8
34	isset_overloading	1			S	✓		-------	1	1	32	6.5	34	7.0	49	7
35	unset_overloading	1			S	✓		----X-	0	0	24	5.5	23	8	29	7
36	call_overloading	1			S	✓		-------	6	1.0	47	4	59	7	68	7.5
37	callstatic_overloading	1			S	✓		-------	3	22	12	35.0	20	141.0	26	41.0

38	invoke	1			S	✓		----X-	1	21	6	1.5	9	3	11	4
39	serialize_unserialize	1			S	✓		---P--	42	1.5	135	5	143	8	188	6.0
40	trait	1			S	✓		----XC	87	6	800	13.0	881	26	907	43
41	self_methods	1			S	✓		---PX-	68	35	360	12.0	482	15.0	602	22.5
42	destructor	1			S	✓		------	102	1	59	4	78	5.5	88	7.0
43	tostring_echo_object	1			S	✓		----XY	9	12	76	14.5	86	25.0	108	16.0
44	verify_return_type	2			S	✓		---PX-	41	32	454	9.0	607	21	713	44
45	static_method_from_variable	1			D2	✓		---P-Y	23	3	98	2.0	166	4.0	235	4
46	object_to_array	1			D2	✓		------	50	13	290	4.0	405	4	475	5
47	Overriding	1			S	✓		---PXC	62	11	569	16	698	24	750	49
48	construct_with_inheritance	1			S	✓		---PX-	64	4.5	471	7	590	11	680	17
49	static_instance	1			S	✓		------	9	3	103	1	114	3	161	2
50	throw_exception	1			D2			------	84	24	610	9.0	739	13	810	21.0
51	catch_exception	1			S			RS---C	95	6	466	4.5	612	6.0	687	11
52	try_catch_finally	2			D2		✓	R---$\overline{X}\overline{Y}$	3	6	39	2	86	2.0	197	2
53	track_error	1			S			---P--	135	20	333	5	407	6	521	7
54	generators	1			S			R---X-	5	1.0	57	2	113	3	204	5.0
55	goto	1			S			----X-	3	8	68	8.0	95	4	155	5
56	exit	1		✓	S			RSWP--	261	3	182	2.0	185	4	226	4.0
57	JS_redirect	1			S			------	151	8	27	3.0	115	3	137	4.0
58	simple_array	2			D1		✓	R--PX$\overline{Y}$	338	336.5	973	120	970	224.5	963	439
59	foreach_with_array	1	✓		S			R----$\overline{Y}$	68	9.5	208	4.0	237	5	297	4
	foreach_with_array	1			S			R--PX$\overline{Y}$	262	21	831	10	883	21	917	29
60	array_walk	2	✓		D2,D4			------	19	1.0	43	1	60	2.0	74	2.0
61	array_map	2	✓		D2,D4			------	41	12	167	3	214	5.0	280	4.0
62	parse_str_function	1	✓		D4			R-----	17	4.0	76	1.0	88	3.0	112	2.0
63	substring_replace_function	1	✓		S			RS---Y	38	4.0	80	3.0	97	3	118	3.0

64	preg_match	1	✓		S			R−−−X−	100	6.0	277	6	279	9	355	6
65	system (system)	1	✓	✓	S			R−−PXY	1	1	20	3.0	37	3	41	2
	system (exec)	1	✓	✓	S			R−−PXY	20	3	78	2.0	100	3.0	130	3.0
	system (unlink)	1	✓	✓	S			R−−PXY	101	3	159	4	181	6	237	5
66	superglobals	1		✓	S			−SWPX−	177	7.0	331	4	355	6	413	6
	superglobals	1		✓	S			R−W−X−	323	19	112	4.0	133	8	148	5.0
	superglobals	1		✓	S			−S−−X−	9	1	58	1.5	81	2	102	2.0
	superglobals	1		✓	S			RSWP−Y	240	9	90	4.0	120	4.0	124	4.0
67	odbc	1	✓	✓	S			RS−PXY	1	1	48	2.0	60	2.0	63	2
68	compact	2	✓		D2-D4			−−−−−−−	1	1	48	2.0	60	2.0	63	2
69	create_function	1	✓		D1			−−−−−−−	1	3	49	2	38	4.0	44	2.0
70	extract	1	✓		D2			−−−−−−−	117	15	80	2.5	100	3.0	89	2
71	array_functions	1	✓		S			−−−−−$\overline{\text{Y}}$	23	1.0	86	2.0	114	2.0	155	3
	array_functions	1	✓		S			R−−−−$\overline{\text{Y}}$	10	1.0	23	2	37	1	40	2.0
72	procedural_queries	1	✓	✓	S			R−−PXY	187	26	30	2.5	33	4	24	4
	procedural_queries	2	✓	✓	S			−−−−XY	73	4	43	3	38	4	38	2.0
73	wrong_sanitizers	2	✓	✓	S			RS−PX−	174	5	242	3.0	303	5	332	5.0
74	dirname	1	✓	✓	D1			−−−−−−−	23	3.0	101	4	111	4	131	4
75	buffer	1	✓		S			−−−−−−−	27	2	150	2.5	190	3.0	181	4
76	function_variable	2			D2,D4			−−−−−−−	40	2.5	315	3	465	4	602	7.0
77	object_callable	2			D2,D4			−−−−−−−	12	6	89	2	141	3	252	4.0
78	autoloading_classes	1	✓		D2	✓		−−−P−−	50	1	123	1	138	1.0	150	2.0
79	dynamic_include	1			D1	✓		R−−PX−	337	44	549	5	600	5.0	636	6.0
	dynamic_include	1			D2	✓		R−−PX−	7	2	14	2.0	19	1	27	2
	dynamic_include	2			D3			−−−−−−−	137	2	155	5	190	4.5	219	4
	dynamic_include	1			D4			−−−−−−−	337	85.0	665	6	712	7.0	754	7.0
80	callback_functions	1			D1			−−−P−Y	41	2	128	3.0	159	3	208	2.5
	callback_functions	2			D2			−−−P−−	5	2	11	2	15	2	31	2
	callback_functions	1			D3			−−−−−−−	8	7	17	2	29	1	38	1.5

	callback_functions	1			D4			------	65	6	180	4.0	188	4.5	269	5
81	new_from_variable	1			D2	✓		------	12	7	90	5.0	122	3.0	127	3
	new_from_variable	1			D3	✓		------	0	0	0	0	0	0	0	0
	new_from_variable	1			D4	✓		------	39	8	394	4.0	503	5	600	6.0
82	methods_variable	1			D2	✓		------	14	4.0	99	4	163	4	211	4
	methods_variable	1			D4	✓		------	46	3	223	3	351	3	448	4.0
83	array_variable_key	2			D2		✓	R---$\overline{XY}$	74	8.0	173	8	255	8	277	5
	array_variable_key	2			D4		✓	----$\overline{XY}$	238	21	763	12	831	18	883	32
84	variable_variables	1			D2			------	24	5.0	20	5.0	27	5	47	5
	variable_variables	1			D4			------	123	5	81	4	108	4.5	147	3
	Total	122	26	16		39	7		7623	1716	22687	1031	27875	2004.5	32572	3097.5
	Average								74	16.66	220.2	10	270.6	19.5	316.23	30.07

Legenda for column Tools: RIPS (R), phpSAFE (S), WAP (W), Progpilot (P), Comm_1 (X), Comm_2(Y)

.1.2 Testability patterns for JS

The JS testability patterns are listed and detailed for the community in our repository [34]. 34 of these patterns resemble their PHP siblings, targeting basic language constructs and operators (OOP, functions and variables). All the others focus on JS peculiarities such as specific data structure handlers (e.g., Proxy, WeakSet) or web operations (e.g., Ajax requests). We classified each pattern instance according to the same dimensions introduced in Section 3.3.1 for PHP: over 153 patterns instances, 40 are about *OOP*, 20 capture *negative test cases*, 22 are *security* related, and 22 refer to *internal API* (recall that these dimensions overlap between each other). For what concerns the *Static vs Dynamic features* dimension, 101 instances are static (*S*) and the rest are dynamic (17 belong to *D1*, 17 to *D2*, 10 to *D3* and 8 to *D4*). Table 2 presents the testability patterns in Javascript.

Table 2: JS Patterns

ID	Pattern	#i	API	SEC	Dyn	OOP	Neg	Tools
1	unset_element_array	1			S		✓	$\overline{\text{L}}--\overline{\text{XY}}$
	unset_element_array	1			S			L−−XY
2	uri	1			S			L−ZXY
	uri	1		✓	S		✓	$--\overline{\text{Z}}-\overline{\text{Y}}$
3	evaluated_call_time	1			S		✓	$\text{L}--\overline{\text{X}}\text{Y}$
	evaluated_call_time	1			S			L−−XY
4	function_apply	1			S			−−−XY
5	variadic	1			S			−−−X−
6	callback_function	2			D1-D3			----Y
7	array_unshift	1			S			L−−X−
8	send_unpack	1			S			−−−XY
9	late_static_binding	1			D2			L−ZXY
10	spread_properties	1			S			−−−XY
11	closure_scope_chain	1			S			L−ZX−
12	NaN	1			S			L−ZX−
13	function_declared_immediately_executed	1			S			−−−−−
14	template_literals	1		✓	S			L−ZXY
15	reflect_delete	2			S-D4		✓	$---\overline{\text{XY}}$
	reflect_delete	1			D4			−−−XY
16	nullish_coaleshing_operator	1			S			L−Z−−
17	call	2			S			L−−XY
18	arguments	1			S			L−−X−
19	nested_function	1			S			L−ZXY
20	too_function_calls	2			S			L−Z−Y
21	new_target	1			S			L−ZX−
22	reduce	1			S			−−−XY
23	forEach_in_nested	1			S			L−Z−−
24	finite	1			S			L−ZXY

25	weak_map	1	✓		D2			L−−XY
	weak_map	1	✓		D2		✓	$\overline{\text{L}}$−−$\overline{\text{XY}}$
26	computed_properties	1			S			−−−XY
27	cast_string_array	1			S			L−ZXY
28	closures	2			D2			L−ZXY
	closures	1			D2			L−−−−
29	recursion	2			D1			L−ZXY
30	generator_delegation	1			D2			−−−X−
	generator_delegation	1			D4			−−−−Y
31	generatorfunction_constructor	1			D3			−−−−Y
32	array_shift	1			S			L−−XY
	array_shift	1			S		✓	$\overline{\text{L}}$−−$\overline{\text{XY}}$
33	array_lenght	1			S		✓	$\overline{\text{L}}$−−$\overline{\text{XY}}$
	array_lenght	1			S			L−−XY
34	bind	1			D1			−−−−Y
35	async_methods	1			D1			L−−−−
36	returned_function	1			S			L−ZX−
37	generators	1			D3			−−−X−
	generators	1			D3			−−−−Y
	generators	1			D3			−−−XY
	generators	1			D3		✓	−−−$\overline{\text{XY}}$
38	while_break	1			S			L−ZXY
39	function_get_arguments	1			D1			L−ZXY
	function_get_arguments	1			D2			−−−X−
	function_get_arguments	1			D4			−−−−−
40	function_name_conflict	1			S			L−ZXY
41	symbol	1	✓		S			−−−XY
42	anonymous_object	1			D1			L−−−−
43	window_global	1			S			L−Z−−
44	array_map	1			D2			L−−XY
45	escape_unescape(deprecated)	1	✓	✓	S		✓	−−$\overline{\text{ZX}}$Y

	escape_unescape(deprecated)	1	✓		S			−−ZXY
46	continue	1			S			L−ZXY
47	check_type	1	✓		S			L−ZXY
48	compare_variables	1			S			−−−XY
49	arrow_function	1			D2			L−Z−Y
50	conditional_assign	1			S			L−ZXY
51	global_variable	1			S			L−Z−Y
52	super_property	1			D1	✓		−−−−−
53	simple_set	1			S	✓		−−−XY
54	define_property	1			D1	✓		L−*Z*−Y
	define_property	1			D1	✓	✓	$\overline{\text{L}}$−$\overline{\text{Z}}$−$\overline{\text{Y}}$
55	inheritance	1			S	✓		−−−−Y
56	weak_ref	1	✓		D2	✓		−−−−Y
57	object_seal	1			S	✓		−−−XY
58	object_freeze	1			S	✓		−−−XY
59	simple_object	1		✓	S	✓		L−−−Y
60	object_create	1			D1	✓		−−−XY
61	delete_properties	1			S	✓	✓	$\overline{\text{L}}$−−−$\overline{\text{Y}}$
	delete_properties	1			S1	✓		−−−−Y
62	static_variable	1			S	✓		−−−−Y
63	to_string	1	✓		S	✓		L−−X−
64	assign_object	1			S	✓		−−Z−−
65	proto	1			D4	✓		−−−X−
66	static_methods_and_prop	1			S	✓		L−−−Y
67	symbol_to_string_tag	1	✓		D1	✓		−−−X−
68	promise	1			S	✓		L−−XY
	promise	2			S	✓		L−ZXY
69	set_and_get	1			S	✓		L−ZXY
	set_and_get	1			S	✓		L−−XY

70	reflect_get	1		✓	S	✓		−−−XY
	reflect_get	1		✓	S	✓		−−−−Y
71	named_class	1			D1	✓		L−−−−
	named_class	1			D1	✓		L−−X−
	named_class	1			D1	✓	✓	$\overline{\text{L}}$−−$\overline{\text{X}}$−
72	errors	7		✓	S	✓		L−−X−
73	weak_set	1	✓		S	✓		−−−XY
74	object_argument	1			S	✓		−−ZX−
75	functions_in_object	1			D1	✓		−−−−Y
76	reference_argument	1			S	✓		−−Z−−
77	object_clone	1			S	✓		−−−−−
78	asynchronous_event_handler	1	✓		D2			L−Z−Y
	asynchronous_event_handler	1	✓		D4			L−Z−−
79	inline_function	1			D1			L−Z−Y
80	json	1	✓		S			L−ZXY
81	text_encoder	1	✓		S			−−−XY
82	location_assign_with_search	1	✓	✓	S			L−Z−Y
83	getAttribute	1	✓	✓	S			−−−−Y
84	try_catch	1			S			L−Z−Y
	try_catch	1		✓	D2			L−Z−Y
	try_catch	1		✓	D2		✓	$\overline{\text{L}}$−$\overline{\text{Z}}$−$\overline{\text{Y}}$
85	block_scope	1			S			L−ZX−
86	type_juggling	1			D3			−−−−−
	type_juggling	2			D3			L−−XY
	type_juggling	1			D3		✓	$\overline{\text{L}}$−−$\overline{\text{XY}}$
87	modules	1			D2			−−−−Y
88	with(deprecated)	1			S			L−−XY
89	proxy	1	✓		S			L−−XY
	proxy	1	✓		S		✓	$\overline{\text{L}}$−−$\overline{\text{XY}}$

90	simple_array	1			S			L−ZXY
	simple_array	1			S		✓	$\overline{\text{L}}$−ZX$\overline{\text{Y}}$
91	destructuring	1			S			L−ZXY
92	set_to_array	1			S		✓	−−−−$\overline{\text{Y}}$
	set_to_array	1			S			−−−−Y
93	for_of	1			S			L−−XY
94	matrix	1			S			−−ZXY
	matrix	1			S			−−−XY
	matrix	1			S		✓	−−−X$\overline{\text{Y}}$
95	arithmetic_operation_array_index	1			S			L−−−Y
	arithmetic_operation_array_index	1			S		✓	$\overline{\text{L}}$−−−$\overline{\text{Y}}$
96	object_literals	1			S			L−−XY
	object_literals	1			S			L−ZXY
	object_literals	1			S			L−Z−Y
97	vulnerable_key_dictionary	2			S-D4			−−−−−
98	throw_exception	1		✓	D2			L−−−−
	throw_exception	1		✓	D4			−−−−−
99	GET_ajax	1	✓	✓	D2			L−ZXY
100	replace_substring	1	✓		S			L−ZXY
101	innerHTML_outerHTML	2	✓	✓	S			L−−−Y
	Total	153	22	22		40	20	

Legenda for column Tools: LGTM (L), NodeJsScan (N), Comm_3 (Z), Comm_1 (X), Comm_2(Y)

.2 Appendix B

126

Table 3: Dataset: Information and Results

ID	Project	Project info				Comm_2			Comm_1			Stitches	Iter
		stars	functions	LoC	sources	before	after	diff	before	after	diff		
1	easyappointments	2281	4561	54619	139	0	0	0	19	19	0	9	2
2	ampache	3074	9894	127334	662	12	66	54	557	557	0	29	1
3	amp-wp	1750	8038	111224	213	0	0	0	1	1	0	38	1
4	sqli-labs	4092	107	6791	214	77	77	0	83	122	39	63	3
5	jetpack	1402	20737	238372	1080	25	32	7	139	365	226	248	3
6	bjyadmin	1743	29217	178780	397	491	492	1	40	40	0	39	2
7	CodeIgniter	18183	3657	35663	308	0	0	0	18	18	0	6	2
8	boinc	1500	22237	124889	385	18	20	2	240	243	3	20	2
9	brefphp	2512	965	6867	145	7	7	0	5	5	0	0	0
10	upload-labs	2872	54	2649	129	25	25	0	43	43	0	11	3
11	cacti	1242	19825	169391	402	5429	6364	935	2397	2948	551	539	3
12	cashmusic	1166	60516	501394	1028	125	153	28	289	294	5	163	4
13	organizr	3914	56312	161912	293	10	10	0	46	46	0	1	1
14	unmark	1544	3151	40325	167	4	6	2	29	29	0	12	2
15	razor	1127	12572	91366	354	55	63	8	431	431	0	66	1
16	CodeIgniter4	4205	9602	110251	776	8	8	0	10	14	4	9	1
17	DVWA	6335	1555	21169	217	28	28	0	33	33	0	18	2
18	diskoverdata	1180	3598	34282	163	661	678	17	277	277	0	29	2
19	docker-labs	10842	23412	167147	1576	1444	1591	147	138	197	59	668	4
20	dolibarr	3339	55001	897702	8609	17249	19010	1761	13846	15044	1198	3860	6
21	drupal	3735	60469	834417	208	0	0	0	41	41	0	7	1
22	elementor	4519	8661	69504	102	4	4	0	20	20	0	18	4
23	facebook-php	3303	238	2794	115	5	5	0	2	2	0	0	0
24	yii2-fecshop	4952	17751	174633	118	4	6	2	162	162	0	11	2
25	loklak-wp	1511	28212	437612	1980	2191	2191	0	262	322	60	780	3
26	phimpme-drupal	1537	74385	699289	618	0	0	0	39	40	1	47	3
27	phimpme-wp	1534	9824	114294	1397	1224	1272	48	155	215	60	554	3
28	freescout	1563	36940	304065	229	0	0	0	56	56	0	14	1
29	FreshRSS	4542	2597	65362	168	29	29	0	82	82	0	3	1
30	Froxlor	1441	1395	60567	453	10	10	0	172	172	0	59	2
31	rhaphp	1078	15980	58617	201	5	9	4	54	55	1	15	2
32	Geocoder	3814	1453	17788	240	0	0	0	35	35	0	0	0
33	glpi	2691	13103	349606	3719	780	1925	1145	1223	1278	55	626	2

34	imgurl	1566	2542	36465	164	1	2	1	56	57	1	20	2
35	scws	1573	417	6722	121	12	36	24	105	105	0	6	1
36	VueThink	1347	1998	24799	119	19	53	34	50	50	0	21	6
37	icecoder	1352	1515	11963	228	192	207	15	137	137	0	21	1
38	php-webshells	1713	1114	67034	1986	2519	2541	22	2401	2435	34	133	2
39	openflights	1187	3189	10642	133	331	362	31	117	152	35	69	3
40	KodExplorer	5685	5197	58321	187	76	85	9	31	33	2	18	2
41	php-benchmark	1032	12335	155976	886	3	3	0	102	107	5	40	3
42	laravel	27494	23460	186354	443	0	0	0	2	2	0	1	1
43	Leantime	1279	2803	25268	386	29	29	0	37	37	0	34	1
44	siler	1124	1354	9518	175	0	0	0	0	0	0	0	0
45	laragon	2681	2532	49531	237	19	19	0	12	12	0	7	1
46	librenms	2801	14740	189156	597	39	103	64	969	990	21	158	6
47	Carbon-Forum	1822	3217	18462	113	33	38	5	85	85	0	20	3
48	Heimdall	4541	38854	500566	852	0	0	0	41	41	0	1	1
49	livehelperchat	1657	16291	259066	1026	455	801	346	615	791	176	335	3
50	Bonfire	1412	9957	94654	290	12	12	0	254	254	0	42	2
51	maccms10	1315	8759	79683	247	64	114	50	78	78	0	43	4
52	mailcow	5379	10489	117508	312	2201	2201	0	196	196	0	0	0
53	mantisbt	1430	4619	75258	101	48	113	65	273	286	13	197	4
54	matomo	16591	56514	278007	479	12	21	9	5	7	2	127	3
55	wp-heroku	1310	16100	364351	1579	1289	1447	158	214	271	57	671	4
56	microweber	2441	37534	223627	552	486	502	16	186	187	1	110	1
57	RPi-Jukebox	1027	689	7418	229	262	299	37	222	222	0	10	2
58	revolution	1303	13340	212276	261	4	5	1	72	96	24	36	3
59	Tieba-Cloud	1457	354	6752	106	155	161	6	154	212	58	41	2
60	nextcloud	19615	36516	374898	149	0	1	1	67	67	0	31	3
61	TeamPass	1423	36731	308672	190	100	102	2	1111	1112	1	46	2
62	php-saml	1041	3718	12467	126	0	0	0	3	3	0	29	1
63	opencart	6524	20323	140010	102	0	0	0	44	44	0	1	1
64	openemr	1950	43594	614196	5700	709	709	0	2165	2165	0	0	0
65	opnsense	2045	7011	90751	748	796	923	127	918	918	0	141	2
66	osTicket	2468	9694	186413	1372	152	569	417	546	549	3	284	2
67	owncloud	7785	141876	1142883	288	9	28	19	167	180	13	124	2
68	pfsense	3751	7660	144419	4501	841	944	103	312	330	18	382	3
69	codefever	2157	9893	84621	170	2	2	0	17	17	0	19	2
70	phabricator	12264	42138	507825	119	0	0	0	24	25	1	22	1

71	cphalcon	10593	12477	136374	454	0	0	0	0	0	0	0	0
72	phoronix	1717	3491	69063	444	320	360	40	734	739	5	64	3
73	phpbb	1545	10958	373623	260	5	6	1	1051	1054	3	55	3
74	phpipam	1722	8574	81371	2633	392	3447	3055	2611	2711	100	640	5
75	phpmyadmin	6103	16629	169239	3082	9	30	21	417	433	16	353	5
76	AdminLTE	1539	889	9289	222	7	7	0	40	42	2	9	1
77	Piwigo	1971	9214	165970	743	263	314	51	572	581	9	200	2
78	PrestaShop	6544	39985	388000	762	153	181	28	333	351	18	125	2
79	PrivateBin	4191	1637	11487	182	4	4	0	1	2	1	8	1
80	q2a	1533	3052	39491	110	66	73	7	43	57	14	125	3
81	raspap-webgui	3682	6620	8196	263	16	16	0	81	81	0	6	1
82	chevereto	2573	2095	24861	451	33	58	25	243	246	3	76	3
83	roundcubemail	4481	5990	76416	507	105	114	9	340	407	67	191	3
84	SuiteCRM	3053	40136	453028	7669	1496	1979	483	4157	4620	463	1021	6
85	skratos	2444	2765	27039	102	0	0	0	6	6	0	3	1
86	CMS-Hunter	1560	3770	48828	337	401	401	0	20	20	0	18	2
87	vesta	2672	2512	38648	1571	146	146	0	91	113	22	173	4
88	sw-platform	1925	41493	523192	294	0	0	0	149	149	0	0	0
89	shopware	1282	51066	355113	179	0	5	5	2094	2094	0	14	1
90	dokuwiki	3476	7869	223130	569	13	17	4	57	65	8	57	1
91	symfony	27127	40468	825093	400	0	0	0	78	78	0	3	1
92	testlink-code	1162	42148	461497	977	734	1023	289	409	502	93	780	4
93	ThinkUp	3316	9827	124552	768	182	349	167	75	94	19	290	2
94	WDScanner	1616	476	5443	193	172	172	0	125	138	13	11	2
95	thinkphp	2856	3768	51134	325	90	154	64	18	18	0	21	2
96	typecho-fans	1428	24394	91606	232	15	27	12	154	158	4	131	6
97	vanilla	2548	25780	220101	217	2	34	32	102	102	0	30	3
98	adminer	5328	1612	26199	357	116	147	31	239	302	63	167	3
99	wordless	1405	7686	70934	105	53	53	0	35	35	0	6	2
100	valet-plus	1527	579	6141	100	0	0	0	11	13	2	2	1
101	Gazelle	1729	2277	71583	1361	2362	2391	29	1224	1808	584	99	2
102	mediawiki	3046	53017	618354	210	5	6	1	41	63	22	74	1
103	woocommerce	8062	20420	252407	968	0	0	0	68	68	0	204	3
104	WordPress	16450	44715	287812	1806	192	222	30	149	167	18	273	4
105	custom-fields-pro	1112	5458	18635	123	0	0	0	1	1	0	0	0
106	PicUploader	1017	67475	688682	184	5	5	0	390	402	12	26	1
107	FruityWifi	2026	4099	18112	186	63	63	0	31	32	1	2	1

108	yii	4830	11056	743321	1015	2	7	5	21	21	0	13	1
109	yii2	13966	11888	117759	431	0	0	0	3	3	0	3	1
110	YOURLS	8268	2548	42477	179	44	61	17	39	50	11	62	3
111	pikachu	2264	204	6934	223	211	211	0	71	76	5	39	2
112	zoneminder	3744	14517	213544	1531	219	323	104	334	688	354	313	2
113	skycaiji	1578	17181	137508	251	44	102	58	148	156	8	97	4
114	dzzoffice	3499	28897	150297	328	501	2562	2061	679	780	101	322	5
	SUM	448K	1.9M	21.4M	85K	49231	61583	12352	50217	54985	4768	17308	251

Project info: Stars in Github, num of functions, PHP line of code, num of Sources. XSS, SQLI and FileM Discoveries: num of Comm_2 alerts (before, after and diff), num of Comm_1 alerts (before, after and diff). Our approach - other info: num of stitches and the iterations till no more stitches.

.3 Appendix C

Table 4: MVC Challenges in Laravel, Symfony, CodeIgniter, and CakePHP

ID	Relation	Framework	Code
1	Controller ↔ Model	Laravel	$products = Products::all();

2	Controller ↔ Model	Symfony	function method_name(ManagerRegistry $doctrine){
			$entityManager = $doctrine->getManager();
			$products = $entityManager->getRepository(Entity_name::class)->findall();
			}

3	Controller ↔ Model	CodeIgniter	$this->load->model('ModelName','PropertyName');
4	Controller ↔ Model	CakePHP	$this->loadModel('ModelName');
5	Controller ↔ View	Laravel	view('view_name', ['var1'=>'val1','var2'=>'val2']); view('view_name')->with('var1', 'val1');
6	Controller ↔ View	Symfony	$this->render('view_name', ['var1'=>'val1','var2'=>'val2']);
7	Controller ↔ View	CodeIgniter	$this->load->view('view_name', ['var1'=>'val1','var2'=>'val2']);

130

8	Controller ↔ View	CakePHP	$this->set('var1', 'val1'); $this->render('view_name');
9	View ↔ View	Laravel	@include('view_name', ['var' => 'val'])

10	View ↔ View	Symfony	// Parent template
			<html><title>{%block title%}Parent{%endblock%}</title></html>
			// Child template
			{% extends 'parent.html.twig' %}
			{% block title %}Child{% endblock %}

11	View ↔ View	CodeIgniter	$data['nested'] = $this->load->view('view2', '', TRUE); $this->load->view('view1', $data);

12	View ↔ View	CakePHP	echo $this->fetch('view2',['var1'=>'val1','var2'=>'val2']);
			$this->extend('view2',['var1'=>'val1','var2'=>'val2']);
			echo $this->element('element_name',['var1'=>'val1','var2'=>'val2']);
			$this->cell('Cell_name::Method_name', ['val1','val2']);
			$this->loadHelper('helper_name');

13	Controller ↔ Controller	Laravel	redirect()->route('route_name', ['var1'=>'val1','var2'=>'val2']); redirect()->action([ControllerName::class,'Method'],['var1'=>'val1','var2'=>'val2']);
14	Controller ↔ Controller	Symfony	$this->redirectToRoute('page_name', ['car1' => val1]); $this->redirect('http://symfony.com/doc');
15	Controller ↔ Controller	CodeIgniter	redirect("className","methodName"); redirect("className");
16	Controller ↔ Controller	CakePHP	$this->redirect(['controller' => 'ControllerName', 'action' => 'MethodName']); $this->setAction('MethodName');

17	Input Management	Laravel	public function action_name(Request $request){
			$name = $request->input('name');
			}

18	Input Management	Symfony	$input = $request->query->get('page', 1);

19	Input Management	CodeIgniter	$this->input->post('param_name');
20	Input Management	CakePHP	$this->loadComponent('RequestHandler'); $input = $this->request->getParam('param_name');

21	Session Management	Laravel	public function action_name(Request $request){
			$request->session()->put('key', 'value');
			$value = $request->session()->get('key');
			}

22	Session Management	Symfony	$session = $this->requestStack->getSession(); $session->set('var', 'val'); $var = $session->get('var');
23	Session Management	CodeIgniter	$this->load->library('session'); $this->session->set_userdata('item', 'value'); $var = $this->session->userdata('item');
24	Session Management	CakePHP	$this->loadComponent('RequestHandler'); $session = $this->request->getSession(); $name = $session->read('item');

132

References

[1] 2021 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[2] Acunetix Web Application Vulnerability Report 2020. https://www.acunetix.com/white-papers/acunetix-web-application-vulnerability-report-2020.

[3] Blade template in Laravel. https://laravel.com/docs/9.x/blade.

[4] Brakeman Pro - static analysis security tool (SAST) for Ruby on Rails. https://brakemanpro.com/.

[5] Code property graph: specification, query language, and utilities resources. https://github.com/ShiftLeftSecurity/codepropertygraph.

[6] Codeigniter Projects in Github. https://github.com/topics/codeigniter.

[7] CodeIgniter Usage Statistics. https://trends.builtwith.com/framework/CodeIgniter.

[8] Comparison of the usage statistics of PHP vs. ASP.NET for websites. https://w3techs.com/technologies/comparison/pl-aspnet,pl-php.

[9] ESSENTIAL SMALL BUSINESS WEBSITE STATISTICS [2023]: HOW MANY BUSINESSES HAVE A WEBSITE. https://www.zippia.com/advice/small-business-website-statistic.

[10] Framework Usage Distribution in the Top 1 Million Sites. https://trends.builtwith.com/framework.

[11] Github-clone-all tool. https://github.com/rhysd/github-clone-all.

[12] Laravel Projects in Github. https://github.com/topics/laravel.

[13] Laravel Usage Statistics. https://trends.builtwith.com/framework/Laravel.

[14] Log4j vulnerability: CVE-2021-44228. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44228.

[15] Opcodes List. http://php.adamharvey.name/manual/en/internals2.opcodes.php.

[16] Our Anonymized Github Repository. https://anonymous.4open.science/r/TestingMVC.

[17] OWASP - Cross Site Scripting (XSS). https://owasp.org/www-community/attacks/xss.

[18] OWASP - SQL Injection (SQLI). https://owasp.org/www-community/attacks/SQL_Injection.

[19] OWASP Testability Patterns for Web Applications. https://owasp.org/www-project-testability-patterns-for-web-applications/.

[20] OWASP top ten security risks - Injection. https://owasp.org/Top10/A03_2021-Injection.

[21] Plume. https://github.com/plume-oss/plume.

[22] Small Business Statistics: The Ultimate List in 2023. https://www.luisazhou.com/blog/small-business-statistics.

[23] Testability Tarpits Library for PHP. https://github.com/enferas/TestabilityTarpits/tree/main/PHP/TestabilityPatterns.

[24] The fifth annual Developer Ecosystem Survey conducted by JetBrains. https://www.jetbrains.com/lp/devecosystem-2021/php/.

[25] Twig template in Symfony. https://twig.symfony.com.

[26] UK Online Shopping and E-Commerce Statistics for 2017. https://www.nasdaq.com/articles/uk-online-shopping-and-e-commerce-statistics-2017-2017-03-14.

[27] Understanding the Impact of Apache Log4j Vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.

[28] Vulncode-DB. The vulnerable code database. https://www.vulncode-db.com.

[29] WHIP Repository. https://github.com/enferas/WHIP.

[30] Why escape-on-input is a bad idea. https://lukeplant.me.uk/blog/posts/why-escape-on-input-is-a-bad-idea.

[31] XSS filtering in Codeigniter. https://www.codeigniter.com/userguide3/libraries/input.html?highlight=xss\#xss-filtering.

[32] Filtering false alarms of buffer overflow analysis using smt solvers. *Information and Software Technology*, 52(2):210–219, 2010.

[33] LGTM. https://lgtm.com/, Accessed April 30, 2023.

[34] Repository of testability patterns and experiments. https://github.com/enferas/TestabilityTarpits, Accessed April 30, 2023.

[35] A. Abraham. Nodejsscan. https://ajinabraham.github.io/nodejsscan/, Accessed April 30, 2023.

[36] F. Al Kassar, G. Clerici, L. Compagna, F. Yamaguchi, and D. Balzarotti. Testability tarpits: the impact of code patterns on the security testing of web applications. NDSS 2022, 2022.

[37] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1015–1022, New York, NY, USA, 2009. Association for Computing Machinery.

[38] A. Algaith, I. A. Elia, I. Gashi, and M. Vieira. Diversity with intrusion detection systems: An empirical study. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–5, 2017.

[39] A. Algaith, P. Nunes, F. Jose, I. Gashi, and M. Vieira. Finding sql injection and cross site scripting vulnerabilities with diverse static analysis tools. In *2018 14th European Dependable Computing Conference (EDCC)*, pages 57–64, 2018.

[40] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 377–392, Baltimore, MD, Aug. 2018. USENIX Association.

[41] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 947–957, 2012.

[42] M. Alkhalaf, S. Roy Choudhary, M. Fazzini, T. Bultan, A. Orso, and C. Kruegel. Viewpoints: Differential string analysis for discovering client- and server-side input validation inconsistencies. *2012 International Symposium on Software Testing and Analysis, ISSTA 2012 - Proceedings*, 07 2012.

[43] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), jul 2018.

[44] N. Alshahwan, M. Harman, A. Marchetto, and P. Tonella. Improving web application testing using testability measures. In *2009 11th IEEE International Symposium on Web Systems Evolution*, pages 49–58. IEEE, 2009.

[45] A. Antoniadis, N. Filippakis, P. Krishnan, R. Ramesh, N. Allen, and Y. Smaragdakis. Static analysis of java enterprise applications: Frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 794–807, New York, NY, USA, 2020. Association for Computing Machinery.

[46] A. Arcuri and X. Yao. Search based software testing of object-oriented containers. *Inf. Sci.*, 178:3075–3095, 08 2008.

[47] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *SIGPLAN Not.*, 49(6):259–269, jun 2014.

[48] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and flexible discovery of php application vulnerabilities. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 334–349, 2017.

[49] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*, NDSS 11, January 2011.

[50] T. Ball, V. Levin, and F. Xie. Automatic creation of environment models via training. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2004.

[51] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, New York, NY, USA, 2007. ACM.

[52] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Active learning of points-to specifications. *SIGPLAN Not.*, 53(4):678–692, jun 2018.

[53] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 332–345. IEEE, 2010.

[54] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. Notamper: Automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 607–618, New York, NY, USA, 2010. ACM.

[55] A. P. Black and T. D. Millstein, editors. *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. ACM, 2014.

[56] A. Brucker, L. Bruegger, P. Kearney, and B. Wolff. Verified firewall policy transformations for test case generation. pages 345–354, 01 2010.

[57] M. Buchler, J. Oudinet, and A. Pretschner. Spacite – web application testing engine. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:858–859, 2012.

[58] S. Burbeck. Applications programming in smalltalk-80: how to use model-view-controller (mvc). 01 1992.

[59] M. Bures. Framework for assessment of web application automated testability. In *Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems*, RACS, page 512–514, New York, NY, USA, 2015. Association for Computing Machinery.

[60] M. Bures. Metrics for automated testability of web applications. In *Proceedings of the 16th International Conference on Computer Systems and Technologies*, CompSysTech '15, page 83–89, New York, NY, USA, 2015. Association for Computing Machinery.

[61] D. Chandra and M. Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 463–475. IEEE, 2007.

[62] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM.

[63] C. Csallner and Y. Smaragdakis. Check 'n' crash: Combining static checking and testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 422–431, New York, NY, USA, 2005. Association for Computing Machinery.

[64] C. Csallner, Y. Smaragdakis, and T. Xie. Dsd-crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), may 2008.

[65] J. Dahse and T. Holz. Simulation of built-in php features for precise static code analysis. 01 2014.

[66] J. Dahse and T. Holz. Static detection of second-order vulnerabilities in web applications. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 989–1003, San Diego, CA, Aug. 2014. USENIX Association.

[67] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State: A State-Aware Black-Box Vulnerability Scanner. In *Proceedings of the*

2012 USENIX Security Symposium (USENIX 2012), Bellevue, WA, August 2012.

[68] A. Doupé, M. Cova, and G. Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *DIMVA*, 2010.

[69] Edgescan. 2020 vulnerability statistics report.

[70] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 752–761, Piscataway, NJ, USA, 2013. IEEE Press.

[71] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, Berkeley, CA, USA, 2010. USENIX Association.

[72] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis. Static analysis of java dynamic proxies. ISSTA 2018, page 209–220, New York, NY, USA, 2018. Association for Computing Machinery.

[73] A. Freeman. *Unit Testing MVC Applications*, pages 159–189. Apress, Berkeley, CA, 2016.

[74] V. Garousi, M. Felderer, and F. N. Kilicaslan. A survey on software testability. *Inf. Softw. Technol.*, 108:35–64, 2019.

[75] S. M. Ghaffarian and H. R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017.

[76] GitHub. CodeQL. `https://github.com/github/codeql`, 2021.

[77] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 631–642, New York, NY, USA, 2016. Association for Computing Machinery.

[78] B. Guangdong, M. Guozhu, L. Jike, S. V. Sai, S. Prateek, S. Jun, L. Yang, and D. Jinsong. Authscan: Automatic extraction of web authentication protocols from implementations. In *Annual Network &*

Distributed System Security Symposium (NDSS), 2013. The Internet Society, 2013.

[79] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association.

[80] S. Guarnieri and B. Livshits. Gulfstream: Staged static analysis for streaming javascript applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, WebApps'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[81] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 177–187, New York, NY, USA, 2011. Association for Computing Machinery.

[82] V. Gupta, K. Aggarwal, and Y. Singh. A fuzzy approach for integrated measure of object-oriented software testability. *Journal of Computer Science*, 1, 02 2005.

[83] L. Hao. Application of mvc platform in bank e-crm. *International Journal of u-and e-Service, Science and Technology*, 6(2):33–42, 2013.

[84] M. Harman. Refactoring as testability transformation. pages 414 – 421, 04 2011.

[85] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[86] D. Hauzar and J. Kofron. Framework for Static Analysis of PHP Applications. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 689–711, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[87] N. Heidke, J. Morrison, and M. Morrison. Assessing the effectiveness of the model view controller architecture for creating web applications.

In *Midwest instruction and computing symposium, Rapid City, SD*, 2008.

[88] R. Hierons, M. Harman, and C. Fox. Branch-coverage testability transformation for unstructured programs. *The Computer Journal*, 48, 05 2005.

[89] M. Hills. Variable feature usage patterns in php (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 563–573, 2015.

[90] M. Hills, P. Klint, and J. Vinju. An empirical study of php feature usage: A static analysis perspective. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 325–335, New York, NY, USA, 2013. Association for Computing Machinery.

[91] K. Hough, G. Welearegai, C. Hammer, and J. Bell. Revealing injection vulnerabilities by leveraging existing tests. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 284–296, 2020.

[92] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 40–52, New York, NY, USA, 2004. ACM.

[93] C. Jaspan. Checking framework interactions with relationships. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 901–902, 2008.

[94] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy*, pages 6 pp.–263, 2006.

[95] A. Kalaji, R. M. Hierons, and S. Swift. A testability transformation approach for state-based programs. In *2009 1st International Symposium on Search Based Software Engineering*, pages 85–88, 2009.

[96] S. Kals, E. Kirda, C. Krügel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference*

on World Wide Web, WWW 2006, Edinburgh, Scotland, UK, May 23-26, 2006, pages 247–256, 2006.

[97] R. Kembang Hapsari, A. Azinar, and S. Sugiyanto. Architecture application model view controller (mvc) in designing information system of msme financial report. *Quest Journals Journal of Software Engineering and Simulation*, 3:36–41, 04 2017.

[98] B. Korel, M. Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. pages 10 pp. – 254, 12 2005.

[99] J. A. Kupsch and B. P. Miller. Manual vs. automated vulnerability assessment: A case study. In *In First International Workshop on Managing Insider Security Threats (MIST), West*, 2009.

[100] P. Kyriakakis, A. Chatzigeorgiou, A. Ampatzoglou, and S. Xinogalos. Exploring the frequency and change proneness of dynamic feature pattern instances in php applications. *Science of Computer Programming*, 171:1–20, 2019.

[101] D. Landman, A. Serebrenik, and J. J. Vinju. Challenges for static analysis of java reflection - literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 507–518, 2017.

[102] Y. Li, T. Tan, and J. Xue. Understanding and analyzing java reflection. *ACM Trans. Softw. Eng. Methodol.*, 28(2), feb 2019.

[103] S. Lipp, S. Banescu, and A. Pretschner. An empirical study on the effectiveness of static c code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 544–555, New York, NY, USA, 2022. Association for Computing Machinery.

[104] B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis, 2005.

[105] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: Specification inference for explicit information flow problems. *SIGPLAN Not.*, 44(6):75–86, jun 2009.

[106] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[107] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in java applications with static analysis. In *USENIX security symposium*, volume 14, pages 18–18, 2005.

[108] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1007–1024, 2017.

[109] M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of javascript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 499–509, New York, NY, USA, 2013. ACM.

[110] A. Majeed and I. Rauf. Mvc architecture: a detailed insight to the modern web applications development. *Peer Review Journal of Solar & Photoenergy Systems*, 1(1):1–7, 2018.

[111] R. Mangal, X. Zhang, A. V. Nori, and M. Naik. A user-guided approach to program analysis. ESEC/FSE 2015, page 462–473, New York, NY, USA, 2015. Association for Computing Machinery.

[112] M. Martin and M. S. Lam. Automatic generation of xss and sql injection attacks with goal-directed model checking. In *Proceedings of the 17th conference on Security symposium*, pages 31–43. USENIX Association, 2008.

[113] S. McAllister, E. Kirda, and C. Kruegel. Leveraging user interactions for in-depth testing of web applications. In *Recent Advances in Intrusion Detection*, pages 191–210. Springer, 2008.

[114] G. G. Meade. Cas static analysis tool study - methodology. 2013.

[115] I. Medeiros and N. Neves. Impact of coding styles on behaviours of static analysis tools for web applications. DSN 2020, pages 55–56, 06 2020.

[116] N. Meng, Q. Wang, Q. Wu, and H. Mei. An approach to merge results of multiple static analysis tools (short paper). In *2008 The Eighth International Conference on Quality Software*, pages 169–174, 2008.

[117] C. Mitre. `https://cve.mitre.org/`.

[118] T. Muske and A. Serebrenik. Survey of approaches for handling static analysis alarms. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 157–166, 2016.

[119] Nikto. `https://cirt.net/Nikto2` (accessed on 2019-11-14).

[120] NIST. NIST - Source Code Security Analyzers. `https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers`.

[121] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira. On combining diverse static analysis tools for web security: An empirical study. In *2017 13th European Dependable Computing Conference (EDCC)*, pages 121–128, 2017.

[122] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira. Benchmarking static analysis tools for web security. *IEEE Transactions on Reliability*, 67(3):1159–1175, 2018.

[123] P. J. C. Nunes, J. Fonseca, and M. Vieira. phpsafe: A security analysis tool for oop web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 299–306, 2015.

[124] O.-J. Oluwatosin, A. Balogun, S. Basri, A. Akintola, and A. Bajeh. Object-oriented measures as testability indicators: An empirical study. *Journal of Engineering Science and Technology*, 15:1092–1108, 04 2020.

[125] OWASP. Code Review Guide v2. `https://owasp.org/www-project-code-review-guide/assets/OWASP_Code_Review_Guide_v2.pdf`.

[126] OWASP. OWASP - Source Code Analysis Tools. `https://owasp.org/www-community/Source_Code_Analysis_Tools`.

[127] R. Paramitha and Y. D. W. Asnar. Static code analysis tool for laravel framework based web application. In *2021 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, 2021.

[128] G. Pellegrino and D. Balzarotti. Toward black-box detection of logic flaws in web applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[129] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. Using dynamic analysis to crawl and test modern web applications. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, pages 295–316, 2015.

[130] progpilot. progpilot - A static analyzer for security purposes. `https://github.com/designsecurity/progpilot`.

[131] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". *SIGPLAN Not.*, 50(1):111–124, jan 2015.

[132] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ISSRE '04, page 245–256, USA, 2004. IEEE Computer Society.

[133] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608, 2015.

[134] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528, Washington, DC, USA, 2010. IEEE Computer Society.

[135] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. 01 2010.

[136] T. N. Security. Nessus. `http://www.tenable.com/products/nessus-vulnerability-scanner` (accessed on 2019-11-14).

[137] M. S. Singh. Mvc framework: a modern web application development approach and working. *International Research Journal of Engineering and Technology*, 2020.

[138] F. Spoto, E. Burato, M. D. Ernst, P. Ferrara, A. Lovato, D. Macedonio, and C. Spiridon. Static identification of injection attacks in java. *ACM Trans. Program. Lang. Syst.*, 41(3), July 2019.

[139] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4f: Taint analysis of framework-based web applications. *SIGPLAN Not.*, 46(10):1053–1068, oct 2011.

[140] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 435–458, Berlin, Heidelberg, 2012. Springer-Verlag.

[141] A. Sudhodanan, A. Armando, R. Carbone, and L. Compagna. Attack patterns for black-box security testing of multi-party web applications. In *Network and Distributed Systems Security Symposium*, NDSS 16, February 2016.

[142] L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir. On the soundness of call graph construction in the presence of dynamic language features-a benchmark and tool evaluation, 09 2018.

[143] T. J. Team. Joern - The Bug Hunter's Workbench. `https://joern.io`, 2021. Retrieved July 2021.

[144] J. Toman and D. Grossman. Concerto: A framework for combined concrete and abstract interpretation. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[145] O. Tripp, P. Ferrara, and M. Pistoia. Hybrid security analysis of web javascript code via dynamic partial evaluation. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 49–59, New York, NY, USA, 2014. ACM.

[146] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *PLDI '09*, 2009.

[147] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web applications. *SIGPLAN Not.*, 44(6):87–97, jun 2009.

[148] Verizon. 2020 data breach investigations report. https://enterprise.verizon.com/resources/reports/2020-data-breach-investigations-report.pdf.

[149] Q. Wang, N. Meng, Z. Zhou, J. Li, and H. Mei. Towards soa-based code defect analysis. In *2008 IEEE International Symposium on Service-Oriented System Engineering*, pages 269–274, 2008.

[150] R. Wang, S. Chen, X. Wang, and S. Qadeer. How to shop for free online–security analysis of cashier-as-a-service based web stores. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 465–480. IEEE, 2011.

[151] O. WAP. Web Application Protection Project. https://securityonline.info/owasp-wap-web-application-protection-project.

[152] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *ACM Sigplan Notices*, 42(6):32–41, 2007.

[153] S. Website. Sourcecodester - free source codes. https://www.sourcecodester.com/php-project, 2021.

[154] S. Wei and B. G. Ryder. Practical blended taint analysis for javascript. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 336–346, New York, NY, USA, 2013. ACM.

[155] Q. Wu, Y. Hu, and Y. Wang. Unit testing and action-level security solution of struts web applications based on mvc. In *2010 International Conference on Biomedical Engineering and Computer Science*, pages 1–4, 2010.

[156] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[157] F. Yamaguchi. *Pattern-based vulnerability discovery*. PhD thesis, Göttingen, Georg-August Universität, Diss., 2015, 2015.

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