UNIVERSITÄT
DES
SAARLANDES

Saarland University

Faculty of Mathematics and Computer Science

Department of Computer Science

Automated Security Analysis of Web Application Technologies

Malte Horst Arthur Skoruppa

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

Saarbrücken, 2017

Tag des Kolloquiums:	14. Dezember 2017
Dekan:	Prof. Dr. Frank-Olaf Schreyer

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Christian Rossow
Berichterstattende:	Prof. Dr. Michael Backes
	Prof. Dr. Andreas Zeller
Akademischer Mitarbeiter:	Dr. Robert Künnemann

Zusammenfassung

Das Web hat sich zu einem komplexen Netz aus hochinteraktiven Seiten und Anwendungen entwickelt, welches wir täglich zu kommerziellen und sozialen Zwecken einsetzen. Dementsprechend ist die Sicherheit von Webanwendungen von höchster Relevanz. Das automatisierte Auffinden von Sicherheitslücken ist ein anspruchsvolles, aber wichtiges Forschungsgebiet mit dem Ziel, Entwickler zu unterstützen und das Web sicherer zu machen.

In dieser Arbeit nutzen wir statische Analysemethoden, um automatisiert Lücken in JavaScript- und PHP-Programmen zu entdecken. JavaScript ist clientseitig die wichtigste Sprache des Webs, während PHP auf der Serverseite am weitesten verbreitet ist.

Im ersten Teil nutzen wir eine Reihe von Programmtransformationen und Informationsflussanalyse, um den JavaScript Helios Wahl-Client zu untersuchen. Helios ist ein modernes Wahlsystem, welches auf konzeptueller Ebene eingehend analysiert wurde und dessen Implementierung als sehr sicher gilt. Wir enthüllen zwei schwere und bis dato unentdeckte Sicherheitslücken.

Im zweiten Teil präsentieren wir ein Framework, das es Entwicklern ermöglicht, PHP Code auf frei modellierbare Schwachstellen zu untersuchen. Zu diesem Zweck konstruieren wir sogenannte Code-Property-Graphen und importieren diese anschließend in eine Graphdatenbank. Schwachstellen können nun als geeignete Datenbankanfragen formuliert werden. Wir zeigen, wie wir herkömmliche Schwachstellen modellieren können und evaluieren unser Framework in einer groß angelegten Studie, in der wir hunderte Sicherheitslücken identifizieren.

Abstract

The Web today is a complex universe of pages and applications teeming with interactive content that we use for commercial and social purposes. Accordingly, the security of Web applications has become a concern of utmost importance. Devising automated methods to help developers to spot security flaws and thereby make the Web safer is a challenging but vital area of research.

In this thesis, we leverage static analysis methods to automatically discover vulnerabilities in programs written in JavaScript or PHP. While JavaScript is the number one language fueling the client-side logic of virtually every Web application, PHP is the most widespread language on the server side.

In the first part, we use a series of program transformations and information flow analysis to examine the JavaScript Helios voting client. Helios is a state-of-the-art voting system that has been exhaustively analyzed by the security community on a conceptual level and whose implementation is claimed to be highly secure. We expose two severe and so far undiscovered vulnerabilities.

In the second part, we present a framework allowing developers to analyze PHP code for vulnerabilities that can be freely modeled. To do so, we build so-called code property graphs for PHP and import them into a graph database. Vulnerabilities can then be modeled as appropriate database queries. We show how to model common vulnerabilities and evaluate our framework in a large-scale study, spotting hundreds of vulnerabilities.

Acknowledgments

First of all, I wish to express my deep gratitude to my advisor Michael Backes. He has created this amazing working place and given me the opportunity to work in the midst of this warm and welcoming group, which has been continuously flourishing over the past years. Despite the growth of the group, Michael never lost sight of any of his students and always tried to bring out the best of each and everyone of us. It was a pleasure learning from him, on a scientific level and beyond. Thank you Michael.

I would also like to thank all my former and current colleagues for creating the enjoyable and inspiring working atmosphere inherent to this group, which I feel very fortunate to have had the chance to be a part of. Particular thanks go to Matthias Berg who introduced me to the group. Had it not been for the work with Matthias during my studies, I would probably not stand here. Certainly, Bettina Balthasar also deserves a special thank you for her long-lasting dedication and for taking care of all the bits and pieces vital to the proper workings of the group, always in her delightfully cheerful manner.

During my time as a PhD student, I have had the chance to work with brilliant people and I would like to thank all of my co-authors for our fruitful collaborations. In particular, I thank David Pfaff, Ben Stock, and Fabian Yamaguchi for numerous captivating discussions during my research that have been of tremendous help for the successful elaboration of our papers.

Additionally, I would like to thank Andreas Zeller, Christian Rossow, and Robert Künnemann for agreeing to be part of my thesis committee. Andreas, as my second referee; Christian, as the committee chair; and Robert, in his words, as my protocol droid. Thank you all!

I also wish to extend my gratitude to my friends for their refreshing company and for helping to take my mind off work. Last, but not least, I deeply thank my wonderful family, who has stood behind me throughout all these years with their unconditional love and faith in me. Your support is invaluable. Thank you for everything.

Contents

Publications

This dissertation builds on work pursued by the author as a researcher and PhD student at the chair of Information Security & Cryptography led by Prof. Dr. Michael Backes and as a part of the PhD program of the Saarbrücken Graduate School of Computer Science at Saarland University. It encompasses the following peer-reviewed publications. The author assures that he is the lead author of these publications.

- Michael Backes, Christian Hammer, David Pfaff and Malte Skoruppa. Implementation-level Analysis of the JavaScript Helios Voting Client. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing – SAC 2016*, pages 2071-2078. ACM, April 2016.
- Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, Fabian Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy – EuroS&P 2017*, pages 334-349. IEEE, April 2017.

In addition, the author is also the lead author of the following publication, which is not a part of this thesis.

- Michael Backes, Martin Gagné and Malte Skoruppa. Using Mobile Device Communication to Strengthen e-Voting Protocols. In *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society – WPES 2013*, pages 237-242. ACM, November 2013.

Finally, the author also contributed to the following paper as a co-author in the context of research conducted at the chair of Information Security & Cryptography.

- Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella Béguelin. Verified Security of Merkle-Damgård. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium – CSF 2012*, pages 354-368. IEEE Computer Society, June 2012.

Chapter 1
Introduction

During the last two and a half decades, the Internet has experienced a tremendous growth and evolved at an astounding rate. With the creation of the World Wide Web in the early nineties, it started out as an *Internet of content* of sorts, a medium mostly used for publishing content such as static websites containing bits and pieces of information. Yet in only a few years, it evolved to an *Internet of services* that offered a variety of online utilities, such as online banking, online shops, and other commercial services, as well as productivity and collaboration tools that triggered a paradigm shift in the way that people could communicate and work together. With the advent of smartphones and widely available mobile broadband access, this development experienced yet another boost and culminated in the *Internet of people* as we know it today: A thriving organism where billions of people all around the globe share their everyday lives in social media. The evolution is ongoing, with the *Internet of things* being the next revolution underway.

Accordingly, the number of Internet users has increased rapidly: Today, around 3.6 billion people have Internet access, amounting to almost half of the world's population [ILS 2017]. There are 1.86 billion active users on Facebook [FB 2017]. Worldwide business to consumer sales via the Internet reached $1.7 trillion U.S. dollars in 2015, and are estimated to reach $2.35 trillion by 2018 [HF 2017]. The number of websites has virtually exploded and grown almost exponentially, with around 1.8 billion websites as of February 2017, where the threshold of 1 billion websites was reached for the first time in September 2014 (see Figure 1.1).

As such, security on the Internet has become a concern of the utmost importance in a relatively short period of time. Indeed, 56% of all web traffic is generated by bots, impersonators, hacking tools, scrapers and spammers, and an estimated 37,000 websites are hacked every day [HF 2017]. Yet the Internet was not originally designed with security in mind: Protocols such as HTTP, IP, or BGP are utterly insecure. While they have been enhanced with cryptographic extensions, yielding protocols such as HTTPS, IPsec, and S-BGP, not all of these extensions have yet been widely adopted (and doing so is often challenging due to technical and economic issues). Even where they have indeed been adopted—such as in the case of HTTPS, which is the de facto standard for security-critical services such as online banking—attacks

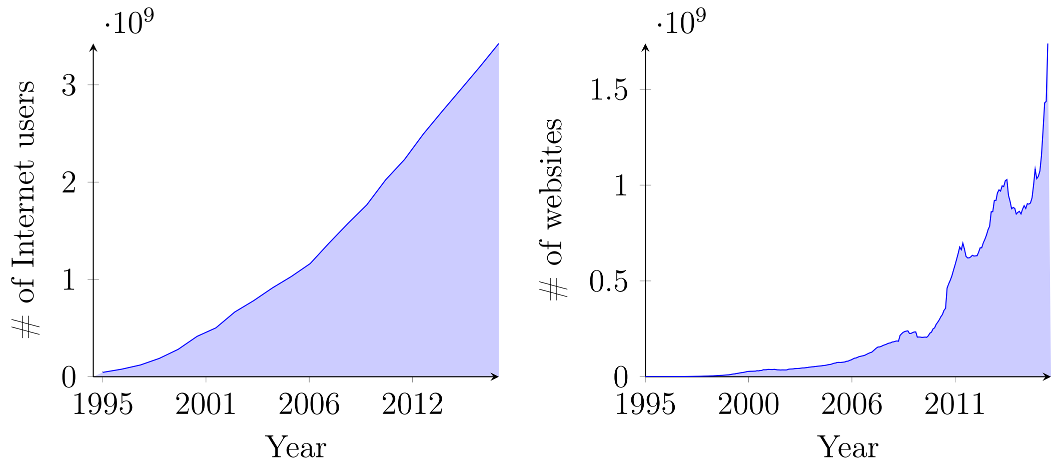

Figure 1.1: Number of Internet users and websites between 1995 and 2017. Data compiled from Netcraft [Netcraft 2017] & Internet Live Stats [ILS 2017].

on the protocols themselves are not the only viable means for attackers to compromise the security of applications.

Indeed, a plethora of attack vectors against web applications exists: Apart from breaching the cryptography underlying a given application, an attacker may target a victim's privacy (say, observe their traffic to unearth confidential data), use social engineering tactics (e.g., using scams or phishing emails which are abundant on the Internet), or abuse an application's implementation to gain partial or even full control over it. In fact, these attack vectors are typically more promising from an attacker's perspective: With the exception of cryptography which stands on a firm and thoroughly understood theoretical background, our understanding of the foundations of these other aspects of security remains somewhat more vague and informal.

Contributions. The present thesis is concerned with investigating the security of the *implementation* of web applications. Indeed, recent breaches in cryptographic applications long deemed secure, such as the prominent *Heartbleed* bug in the implementation of the OpenSSL cryptography library, or Apple's *goto fail* bug in its own SSL/TLS implementation, impressively demonstrate the need to devise methods to aid developers in spotting vulnerabilities at the implementation level early on and help to validate the security of implementations.

Over the past twenty-five years, countless technologies used in the development of web applications have emerged (whether they were originally developed for other purposes or not), such as HTML, CSS, Perl, Java EE, Ruby on Rails, Python with Django, etc.; the list goes on. Two of the most prominent languages that have established themselves as core technologies in the development of web applications are JavaScript and PHP: While JavaScript is the most widely used language to implement client-side logic by a far stretch, PHP has a similar significance for server-side code. For this reason, we will focus on devising methods to automatically discover vulnerabilities in applications written in either of these languages.

Static program analysis is the analysis of program code without executing the program in question. Instead, appropriate structures are created to represent a program's source code (or possibly its object code) and these structures are analyzed for patterns relevant to an application at hand. Static program analysis stands in contrast to *dynamic program analysis*, which executes a program (possibly symbolically) to observe its behavior. Dynamic program analysis is prominently used in software testing, such as for unit and integration tests. However, measures must be taken to reach an adequate code coverage and observe a satisfying percentage of a program's possible behavior. Static analysis techniques typically do not suffer from this problem and are more efficient since they do not require running a program for each input. Yet static analysis lacks access to runtime information and is consequently significantly less precise than dynamic analysis. Given the highly dynamic nature of PHP and JavaScript (which we elaborate on later), dynamic analysis may therefore appear to be the more natural approach to analyze applications written in these languages. However, particularly given the increasing amount and complexity of web applications, dynamic analysis techniques to discover vulnerabilities do not scale well, are expensive, and may miss some vulnerabilities that would become apparent more easily with static analysis techniques.

For this reason, we leverage static analysis to automatically highlight possible security vulnerabilities in source code in the most widespread languages for developing web applications on the client and on the server side, namely, JavaScript and PHP.

In the first part of the thesis, we consider JavaScript. More specifically, we analyze the implementation of the Helios voting client [Adida 2008]. Helios is a state-of-the-art, web-based, open-audit voting system that is continuously being deployed for real-life elections. While it has been exhaustively analyzed by the security community on a conceptual level, the JavaScript implementation of its client has not received the same scrutiny. Yet the original paper specifically details various technical measures that have been taken to make the implementation of the client highly secure. To analyze the JavaScript code that makes up the client, we must overcome various technical challenges, such as JavaScript's dynamic nature, its intermingling with the HTML DOM, or the use of highly complex third-party libraries. We use a series of code transformations, construct appropriate program representations and perform an automated information flow analysis. Thereby, we expose two severe and so far undiscovered security vulnerabilities that lead to voters' votes being sent over the network in plaintext, and enable attackers to control the voting client executed in a victim's browser.

In the second part, we turn our attention to PHP. We present an inter-procedural analysis technique based on the recently proposed concept of *code property graphs* [Yamaguchi *et al.* 2014]. These graphs incorporate a program's syntax, control flow, control dependencies and data dependencies in a single structure. This structure lends itself well to being stored in a *graph database*: Graph databases are an emerging technology that store data as graphs instead of tables, as traditional relational database systems do. We present a framework that automatically generates code property graphs for entire PHP projects and stores them as a graph database. Then, using appropriate queries that model vulnerability-related patterns, we are able to automatically identify various types of vulnerabilities. In addition to being very efficient, one of the core strengths of this approach is its high degree of flexibility: All a developer or analyst needs to do to model other types of vulnerabilities (e.g., very specific ones) is to write appropriate queries for the graph database. We proceed to model the most common types of vulnerabilities that occur in PHP code as queries to the graph database. Then, we leverage our framework and our queries to perform the largest security-centered study of PHP applications to date, scanning a total of 1,854 popular open-source projects comprising almost 80 million lines of code, and uncovering hundreds of vulnerabilities of various types in the process.

These contributions, as well as related work, will be discussed in more detail in their respective chapters.

Outline. The outline of this thesis is as follows. In Chapter 2, we review common types of vulnerabilities in web applications which are relevant for both JavaScript and PHP code (as well as other web development languages). In Chapter 3, we discuss various forms of program representations that we later leverage to perform our analysis. In Chapter 4, we report on our analysis of the JavaScript Helios voting client. In Chapter 5, we present our framework for automated discovery of vulnerabilities in PHP code and our large-scale study. Finally, Chapter 6 concludes.

Chapter 2

Web Application Vulnerabilities

Contents

As we discussed in the introduction, breaches of security in web applications have become an attractive and worthwhile target for attackers. Since we focus on security breaches concerning the *implementation* of web applications in this thesis, in this chapter we present and discuss the most common vulnerability classes in web application code, take a closer look at those instances which are covered by the work in this thesis, and review mitigation techniques. Subsequently, we identify common patterns in these vulnerabilities that enable us to implement techniques to assist in their automated discovery.

The vulnerabilities presented in this chapter are accompanied by illustrating examples. These examples are chosen in such a way that they are as simple as possible for the sake of presentation, but nevertheless realistic in the sense that these very code snippets might be used by actual developers in a given context. All examples are written in either PHP or JavaScript, as these are the two programming languages considered in this thesis.

We first present a general overview of the most prevalent classes of vulnerabilities in web applications in Section 2.1. We then look at the concrete instances of these classes of vulnerabilities in the following two sections, discerning between attacks targeting the server (Section 2.2) and attacks targeting the client (Section 2.3). For instance, attacks on the server may aim to corrupt the server's database or execute malicious code on the server. A typical example of an attack targeting the client is one that attempts to steal a user's credentials. We discuss recurring patterns in vulnerabilities and draw conclusions in Section 2.4.

2.1 General Overview

With the increasing complexity of web applications and the continuously growing number of languages and libraries available to implement them comes a plethora of vulnerabilities threatening both servers and clients. In this section, we briefly summarize the most common types of vulnerabilities as classified by the Open Web Application Security Project,[1] a non-profit organization with the aim of improving the security of software on the Web. Amongst other knowledge-based documentation, they provide a ranking of the top ten web application security flaws, representing a broad consensus of the most critical classes of web application vulnerabilities: The most recent ranking was published in 2013 [OWASP Top Ten 2013]. The purpose of this section is twofold. The first is to familiarize the reader with the most common classes of web application vulnerabilities before we move on to concrete instances and examples of vulnerabilities in the next sections. The second is to emphasize that the techniques presented in this thesis are not limited to the discovery of very specific types of vulnerabilities, but indeed enable the discovery of a broad range of severe security issues: As we will see in Chapters 4 and 5, we were able to discover instances of the top four classes of vulnerabilities in this ranking, as well as instances of the sixth class (the fifth class concerns security misconfigurations of servers which are unrelated to program code). We now briefly recapitulate these vulnerability classes, as described by OWASP. Italic letters denote a citation from the OWASP Top Ten [OWASP Top Ten 2013]:

1. **Injection.** *"Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization."*

Injection flaws are by far the most common type of vulnerability: In Sections 2.2.1, 2.2.2 and 2.2.3, we discuss SQL injections, command injections

[1] http://owasp.org

and code injections, respectively. In Chapter 5, we present a framework for detecting vulnerabilities in PHP code and use it to detect these three instances of injection-type vulnerabilities. Moreover, as we will see, the detection process can be customized as needed and in principle allows to detect any type of injection flaw.

2. **Broken Authentication and Session Management.** *"Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users' identities."*

The PHP framework that we present in Chapter 5 allows us to detect flaws wherein an attacker may impersonate a victim by hijacking their session, too. More specifically, we will see how to detect so-called *session fixation* flaws, which we discuss in more detail in Section 2.3.2.

3. **Cross-Site Scripting (XSS).** *"XSS flaws occur whenever an application takes untrusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victim's browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites."*

Cross-site scripting vulnerabilities are among the most widespread vulnerabilities on the Web, and we discuss them in more detail in Section 2.3.1. As we demonstrate in Chapter 5, our PHP framework can detect such vulnerabilities: In a large-scale experimental study, we see that the number of XSS vulnerabilities that we find exceeds that of all other types of vulnerabilities by far. This confirms the common belief that cross-site scripting vulnerabilities are nowadays the most pervasive threat to web applications. Additionally, we also demonstrate how to detect such vulnerabilities in JavaScript applications in Chapter 4.

4. **Insecure Direct Object References.** *"A direct object reference occurs when a developer exposes a reference to an internal implementation object, such as a file, directory, or database key. Without an access control check or other protection, attackers can manipulate these references to access unauthorized data."*

Path traversal attacks, which we present in Section 2.2.4, are a prominent example of this vulnerability class, wherein an attacker may manipulate file objects generated within the server-side code because the path used to instantiate that object depends on attacker-controllable input. In Chapter 5, we show how our PHP framework may be used to detect this kind of attack, too.

5. **Sensitive Data Exposure.** *"Many web applications do not properly protect sensitive data, such as credit cards, tax IDs, and authentication credentials. Attackers may steal or modify such weakly protected data to conduct credit card fraud, identity theft, or other crimes. Sensitive data deserves extra protection such as encryption at rest or in transit, as well as special precautions when exchanged with the browser."*

We discuss this type of vulnerability in more detail in Section 2.3.3. Subsequently, in Chapter 4, we discuss a semi-automated method to detect such leaks in JavaScript applications. We then apply this method in a case study on Helios, a popular e-voting application deemed highly secure and implemented explicitly in such a way that it requires no network interaction while a voter is casting their vote. As a result, we find that it may, in some cases, actually send out the voter's choices over the network unencrypted.

2.2 Vulnerabilities Threatening the Server

In this section and the next, we present the specific types of vulnerabilities covered in this thesis. The illustrative code examples in this section are all given in the PHP language. Note that even though PHP is a server-side language and JavaScript is most commonly used as a client-side language, a vulnerability that is found in PHP code does not necessarily result in an attack targeting the server, but may equally result in an attack targeting the client, and vice versa for JavaScript. For instance, a vulnerability in PHP code that allows an attacker to print arbitrary characters in a page visited by a victim may very well be used to execute arbitrary code in the client's browser, i.e., affect the client, as we will see in Section 2.3. Similarly, a vulnerability in JavaScript code can result in an attack on the server: This is obviously the case when JavaScript code is executed by the server itself, such as with frameworks like Node.js.[2] But even when JavaScript code is executed on the client side, an attack may target the server, e.g., if the target is a site administrator who has been granted special access privileges to the server's database.

For server-side attacks, a multitude of vulnerabilities has to be considered. In the following, we present the vulnerabilities relevant to our case studies which we present in Chapters 4 and 5. These vulnerabilities represent popular and widespread instances of the classes of vulnerabilities discussed in Section 2.1. In addition, we also discuss some specific mitigation techniques which ensure (when used properly) that a potentially critical use of data within a program cannot be exploited.

[2] https://nodejs.org

```php
<?php
function foo() {

  $x = $_GET["id"];

  if(isset($x)) {
    $sql = "SELECT * FROM users
            WHERE id = '$x'";
    return query($sql);
  }
}
?>
```

Figure 2.1: Classical SQL vulnerability in PHP.

2.2.1 SQL Injections

Web applications often rely on a database back end to read and write persistent data. This data may contain sensitive information like passwords, credit card numbers, and so forth. Hence, it constitutes an attractive target for attacks. Besides stealing information, an attacker may also wish to corrupt the database or compromise the web server, say, change another user's password or drop a table in the database.

SQL Injection Attacks are a widespread and well-known type of privilege escalation attack [Halfond *et al.* 2006] which allow an attacker to gain elevated access to a database used by a web application. Since SQL queries are often generated dynamically depending on user input, an attacker may be able to submit their own SQL syntax as input and thus inject it into an SQL query performed by the web application, thereby modifying the original SQL query intended by the programmer.

A trivial example of an SQL injection vulnerability in PHP is shown in Figure 2.1. The GET parameter `$_GET["id"]` is a user input that is assigned to the variable `$x` and flows into an SQL query without being checked or sanitized. Therefore, an attacker can submit an input such as `' OR 1=1; --` to have the application return the entire table of users, which, depending on the context, may unintentionally leak information (the `--` starts a comment so as to mask the rest of the original SQL query). If the SQL database back end supports batched statements, the attacker may even be able to easily execute arbitrary statements, using an input such as, say, `'; DROP TABLE users; --`. But even if the database back end does not support batched statements, once an attacker has found a way to modify a given query, achieving their desired goal is more often than not a simple routine task.

A common mitigation technique consists in applying custom or built-in sanitization functions like `mysql_real_escape_string` (in the case of PHP) to

```php
<?php
function foo() {

  $x = mysql_real_escape_string($_GET["id"]);

  if(isset($x)) {
    $sql = "SELECT * FROM users
            WHERE id = $x";
    return query($sql);
  }
}
?>
```

Figure 2.2: Modified example of the SQL vulnerability in Figure 2.1.

escape special characters that have a syntactical effect on the SQL query, such as single and double quotes or backslashes. However, this kind of sanitization does not constitute an ideal defense in all situations. Consider the modified example in Figure 2.2: Although a sanitization function is applied, the query is not safe, because the query does not use quotes to enclose the variable `$x`. As long as the input is an integer as expected, this will work fine, however, an attacker can still modify the query by using, for example, `0 OR 1=1` as input.

Prepared statements are generally regarded as the safest way to prevent SQL injections. However, even prepared statements only protect against first-order SQL injection vulnerabilities such as the ones in the previous two examples. They do not protect against second-order SQL injection vulnerabilities, wherein an attacker may have stored some malicious input in the database, say, as part of a username. If that value is later retrieved and used in another SQL query within the web application, the application may still be vulnerable.

Summing up, even though SQL injections are rather popular and well-known to most developers, it is still necessary for developers to pay close attention to them while writing an application, and since database queries are a common task in web applications, mistakes resulting in security flaws are not uncommon.

2.2.2 Command Injections

Web applications that run on a host machine may want to spawn external processes for a variety of reasons. Use cases range from simple tasks, such as extracting an archive or resizing an image, to more complex tasks, e.g., monitoring and controlling a set of processes running on the host machine. For this purpose, PHP offers several tools to execute commands on the system shell, like `shell_exec`, `passthru`, `popen`, `system` and the backtick operator (all of these only exhibit minor technical differences).

```php
<?php
$search = $_GET['search'];
$command = "find /var/www/userfiles/ -regex ".$search;
echo shell_exec($command);
?>
```

Figure 2.3: Example of a command injection vulnerability.

If a web application uses user-supplied input to dynamically generate a shell command for the underlying operating system, an attacker may be able to exploit this fact to modify the intended behavior of the command or even to execute commands of their choosing altogether by injecting input that has a syntactical effect when inserted in the original command. The attacker-injected command is then executed with the privileges of the vulnerable application. In the case of a Unix server for example, PHP applications are commonly run by the virtual user `www-data`, which typically has extensive privileges in the file system hierarchy pertaining to web applications running on a given server.

Consider the example in Figure 2.3, which allows users to find files whose names match a given pattern in a certain directory. The input is not sanitized, therefore, an attacker could provide, for instance, the input `.; rm *` to close the `find` command and instead execute a command to delete all files in the current directory.

This type of attack is usually protected against by validating the input first or running some type of sanitization routine. PHP offers the function `escapeshellcmd`, which ensures that an attacker cannot trick the shell into executing another command by escaping characters that can be used to do so, such as `;`, `&`, `|`, etc. This, however, must also be used with caution: Even if the code snippet in Figure 2.3 used `escapeshellcmd` to sanitize the variable `$command` before passing it to the shell, it would still be vulnerable. For instance, an attacker could provide the input `.* -delete` as a search string to instruct the `find` command itself to delete any files matching the regular expression `.*` in the search directory, without the need to invoke another command. Therefore, PHP also provides the function `escapeshellarg` which ensures that a given input can only be used as a single argument by surrounding it with single quotes and escaping any single quotes within the input. Which of the two built-in escaping functions should be used depends on the context, and requires developers to be fully aware of such little quirks and focus on avoiding security threats while writing the application.

```
<html>
  <body>
    <form method="GET">
      <input type="text" name="formula" value="<?php echo $_GET['formula']; ?>">
      <input type="submit" value="=">
      <?php if( isset($_GET['formula'])) eval( "echo ".$_GET['formula'].";"); ?>
    </form>
  </body>
</html>
```

Figure 2.4: Example of a code injection vulnerability.

2.2.3 Code Injections

Many languages, including PHP and JavaScript, provide constructs to evaluate strings as code at runtime. The prime example is the function `eval`, which exists both in PHP and JavaScript. Sometimes using `eval` is convenient for the developer (in the case of JavaScript for example, typical use cases considered as acceptable include fallback JSON parsing, and asynchronous content and library loading [Richards *et al.* 2011]); in others, the use of `eval` results from poor understanding of the language and its features.

Regardless of the use case, using `eval` is precarious from a security perspective. Indeed, since code run within `eval` is executed in the current scope of the program as if it were normal code, it is able to reach deeply into the program state and make arbitrary changes. For instance, it may add, modify or remove fields or methods from existing objects, overload existing operators or functions, redefine custom or built-in classes, load additional libraries, and so forth. Hence, if text passed as an argument to `eval` includes code that an attacker can supply or influence in a critical way without it being properly checked or sanitized, the attacker may be able to force the application to execute code of their choosing. In addition to being a security risk, invocations of `eval` are also a hindrance to static analysis, as we will discuss in Chapter 4.

Consider the example in Figure 2.4 which shows a simple implementation of a calculator in PHP. A user may enter an arbitrary formula (e.g., `1+1`), click on `=` and get the result printed on the screen. While `eval` is certainly convenient for the programmer here (the actual PHP code computing the submitted formula is a one-liner), this code is also vulnerable to a code injection attack. Indeed, it enables an attacker to submit arbitrary PHP code which will be executed by the PHP interpreter on the server side.

There are variants of `eval` which are equally dangerous. For instance, JavaScript has a number of technically similar facilities such as `setTimeout`, `setInterval`, and `Function`. In the case of PHP, the constructs `include` and `require` (and their variants `include_once` and `require_once`) cause the PHP interpreter to read and interpret the contents of the passed file at the point where they are included and in the scope of the current program. If an attacker can influence the value passed to these language constructs, a vulnerability arises that may be exploited in different ways. For instance:

1. The attacker may be able to place a malicious PHP file on the server. Consider for example a forum software that allows to upload avatars. Note that the PHP interpreter does not place any restrictions on included filenames, i.e., a filename with a `.jpg` ending instead of a `.php` ending is perfectly acceptable.
2. The attacker may know the location of a file already on the server that they can misuse for their purposes, say, an administrative PHP script that is not usually publicly accessible.
3. In certain setups, the PHP interpreter even allows remote file inclusion over HTTP(S) [PHP Group 2017c], such that even remote URLs may be used as arguments, resulting in the possibility to load and execute remote code.

This kind of attack is often referred to as a *file inclusion* attack, but it is only a special case of a code injection attack.

Finally, as the necessary payload depends on the exact nature of the flawed code, there is no general sanitizer which may be used to thwart all these attacks in either PHP or JavaScript (or any other language as far as the author of this thesis is aware); the kind of acceptable input varies depending on the use case and must be manually checked or sanitized by the developer. In summary, great care has to be taken by developers when using `eval` and its variants, and misusing it may lead to a plethora of vulnerabilities.

2.2.4 Arbitrary File Accesses

Web applications read and store files on the server frequently. This is not limited to some configuration files or logs. In the interactive Internet that we live in today, people exchange files such as images, videos, or music all the time, which may be processed by web applications in a wide range of use cases.

Therefore, it is not rare for web applications to give users a certain amount of control over the files that they want to handle. Consider for example an interactive photo album, where a user may upload and download their own photos, edit or delete them, read meta-information contained in the pictures, etc. The web application may process user input to generate the path for accessing a certain file (say, to retrieve a photo based on a timestamp). If this application does not check or sanitize the user input sensibly, attackers may be able to make the web application unintentionally access sensitive files.

A common scenario in PHP is that an application opens a file by passing a string such as `$prefix."/".$input` to a call to `fopen`, where `$prefix` is a

```php
<?php
$logfile = $_GET['logfile'];
$handle = fopen( "/var/www/logfiles/".$logfile, "r");
$contents = fread( $handle, 4096);
echo $contents;
?>
```

Figure 2.5: Example of a path traversal vulnerability.

fixed path and `$input` is a file name specified by the user (the dot denotes string concatenation). However, when this is done naively, an attacker can specify input containing characters that have a special meaning for the filesystem. On Unix-like operating systems, for instance, the sequence `../` can be used to traverse the directory hierarchy upwards. An attacker may hence provide an input prefixed with a repetition of this sequence so as to traverse directories backwards as far as needed and access any file on the file system. For this reason, this technique is called *path traversal*. The attacker is, of course, limited by the access control of the operating system, but not by the application. Note that the path traversal technique also lends itself well for file inclusion attacks, which we saw in the previous section.

The example in Figure 2.5 illustrates the basic idea. In this program, a user may specify the name of a logfile, which is then read from the directory `/var/www/logfiles` and printed. An attacker could simply submit the input string `../config.php` to have the application read and print the contents of the file `/var/www/config.php`, which may contain sensitive information such as database passwords.

In scenarios where a suffix is appended to the user input in addition to being prepended by a prefix, this kind of vulnerability becomes more difficult, but not impossible, to exploit. In some combinations of operating system, web server software and PHP version, null bytes (`%00` in the query string) can be used to terminate the string and mask the suffix. Another possibility in older PHP versions was to suffix the filename with the character `/` followed by a long repetition of the character sequence `./`. The PHP interpreter treated files like directories and therefore ignored this suffix, but at the same time, was prevented from reading the actual intended suffix, because filenames were also truncated to a certain maximum length.

This kind of vulnerability is often defended against by using regular expressions, which aim to remove, e.g., dots from the input. A canonical way is to use the built-in function `basename`, which strips the directory part of a given path and leaves behind only the filename itself. This prevents the above attack. Yet it still allows for attacks that aim to read sensitive files which are in the same directory as the files which users are allowed to read (or nested deeper within the directory hierarchy). This can only be warded against by using appropriate filesystem permissions and using a carefully planned filesystem structure, all of which put additional burdens on developers.

2.3 Vulnerabilities Threatening the Client

In this section, we take a look at some prominent attacks targeting the client. Such attacks aim to exploit users of a web application. For instance, an attacker may be interested in stealing a user's credentials, in hijacking their session, or in making their browser behave in a certain unexpected and undesirable way.

Here, we only focus on attacks that target clients by exploiting insecure application code, but other vectors of attacks exist that are abundant on the Internet. To give a few prominent examples, *phishing attacks* attempt to make a victim believe that they are interacting with a trustworthy agent—when they are actually interacting with the attacker—so as to make the victim disclose sensitive information. *Clickjacking attacks* attempt to make the user perform unwanted actions on an authentic page (say, a shop or a banking site) by loading this page as a transparent layer on top of a seemingly innocuous page. When users try to interact with this supposedly harmless page, they are actually interacting with the authentic page and perform undesirable actions without their knowledge. Another well-known example is that of *cross-site request forgeries*. These are performed by malicious websites which cause a victim's browser to send unwanted requests to a trusted website where the user is currently authenticated. Since the victim's browser is authenticated on that website, the attacker can thereby cause it to perform actions on the victim's behalf. CSRF attacks are easily defended against by using tokens, i.e., a trusted website should not allow single requests to actually perform an action, but instead should send, upon a first request, a random token to the user's browser. An action initiated by a (second) request should be allowed only if the user's browser sends this token back along with the request. This defense thwarts CSRF attacks completely, unless, as we will discuss in the next section, a cross-site scripting vulnerability is additionally present on a trusted website.

We now discuss three kinds of vulnerabilities directly caused by insecure server-side code: The first is the above-mentioned cross-site scripting, one of the most widespread types of vulnerabilities in web applications. The second is session fixation, a less common, but relevant type of vulnerability which allows an attacker to hijack a victim's session. The third is sensitive data exposure caused by insufficient transport layer protection, another fairly common vulnerability involving a passive network attacker.

2.3.1 Cross-Site Scripting (XSS)

Client-side executable code, such as JavaScript, gives developers of web applications the possibility to make their applications faster, more interactive and allows them to shift parts of the business logic to the client, thereby substantially reducing traffic and requirements for computational power on the server. However, client-side executable code also brings with it additional security risks. Since JavaScript allows to read and manipulate the DOM of a web page, it is an effective way for an attacker to control a victim's browser in the context of a vulnerable application. Fortunately, doing so is not as simple as writing a malicious page which loads a trusted website in an invisible frame and reads or manipulates its DOM: This is prevented by the *same-origin policy*, a central security concept for web applications implemented by all major browsers. Under this policy, a browser does not permit a script contained in a web page to access the DOM of another web page if the two pages do not have the same origin. Here, *origin* is defined as the triplet of protocol, hostname, and port number (e.g., `(http, example.com, 80)`). Hence, to achieve their goal, an attacker typically aims to *inject* a malicious script into the page sent to a victim from a trusted application. In this way, the same-origin policy is circumvented, as the malicious script looks as if it originated from the trusted website. This type of attack is known as *cross-site scripting* (XSS). Attackers may leverage this attack in a variety of ways. Apart from the well-known attacks which target the theft of session cookies [Kirda *et al.* 2006], cross-site scripting vulnerabilities even enable attackers to extract plaintext passwords [Stock & Johns 2014].

It is very easy for developers to make programming mistakes that lead to cross-site scripting vulnerabilities. Consider for example the code snippet in Figure 2.6, which shows a simple user message displayed by a search engine. The search parameter provided by the user is echoed in the output without any kind of sanitization. Therefore, an attacker can create a link such as

```
http://example.com/search.php
          ?search=<script>alert('XSS');</script>
```

and lure the victim into visiting that link. The result is that the script will be embedded into the output and evaluated by the victim's browser. As the payload is reflected in the output, this kind of cross-site scripting vulnerability is called a *reflected* XSS flaw. A second kind is that of *persistent* XSS vulnerabilities, where an attacker is able to cause a server to inadvertently store malicious code and display it on normal pages visited by victims: Consider for instance an online forum where users are allowed to post messages containing HTML code. This type of vulnerability is typically even more devastating, as more victims can be affected more easily and

```
<?php
$search = $_GET['search'];
echo "Your search for <em>$search</em> returned the following results:";

// process and display results
?>
```

Figure 2.6: Example of a reflected cross-site scripting vulnerability.

```
<html>
  <head>
    <script language="javascript">
      function getQueryVar( key) {
        // ... return GET parameter with given key from query string ...
      }
      function compute() {
        formula = getQueryVar( "formula");
        document.getElementById( "formula").value = formula;
        document.getElementById( "result").innerText = eval( formula);
      }
    </script>
  </head>
  <body onload="compute()">
    <form method="GET">
      <input id="formula" name="formula" type="text">
      <input type="submit" value="=">
      <span id="result"></span>
    </form>
  </body>
</html>
```

Figure 2.7: Example of a DOM-based cross-site scripting vulnerability.

without further action by the attacker. The third and final type is that of *DOM-based* XSS vulnerabilities. As opposed to the two first types, these do not exploit server-side code flaws at all, and in fact can even be performed against static web pages. For illustration, consider again our example of a calculator (see Section 2.2.3), but this time implemented on the client side, in Figure 2.7. A JavaScript obtains a given formula from a GET parameter, then evaluates it and displays the result (note that there is no native method for extracting GET parameters in JavaScript, but a plethora of custom implementations exist). Here, an attacker can make a victim visit a URL akin to `http://example.com/calculator.html?formula=alert('XSS');` to make the victim's browser evaluate a JavaScript of their choice. For simplicity, this example makes it particularly easy for an attacker, but even when the input is not evaluated on the client side using `eval`, but simply echoed somewhere in the DOM, an attacker can often execute arbitrary scripts on the client side by surrounding the input with `<script>` tags, whence the name DOM-based XSS attack.

Summing up, cross-site scripting vulnerabilities are a powerful tool for attackers to circumvent the same-origin policy and trick a victim's client into evaluating malicious scripts in the context of a web application. They may also be used to perform other kinds of attacks, e.g., cross-site request forgeries are possible in the presence of an XSS vulnerability even when tokens are used as CSRF protection, since the attacker can use the XSS vulnerability to learn these tokens. Generally, XSS attacks allow an attacker to simulate an arbitrary interaction between an affected client and a web server, even without the knowledge of the client. What is more, programming mistakes that lead to these vulnerabilities are easily made and quickly overseen, as it already suffices for a developer to insert untrusted data in certain locations, either in the server- or in the client-side code. PHP ships built-in sanitizers for different use cases, such as `htmlspecialchars`, `htmlentities`, or `strip_tags`, to mitigate this problem, while JavaScript does not come with any built-in sanitizers at all. It is therefore essential for developers to be well-aware of cross-site scripting vulnerabilities and pay close attention while developing the application. Accordingly, human error is frequent and XSS vulnerabilities are common and widespread.

2.3.2 Session Fixation

HTTP is a stateless protocol. Therefore, cookies are used to identify a particular client of a web application across several requests. Typically, upon a first request, a new *session identifier* is generated by the server and stored internally. This session identifier is sent to the client, who stores it as a cookie, and sends it to the web application with each further request, allowing the server to identify the client. Note that this does not imply that the client needs to be *authenticated* to the web application in any way. Consider for example an online shop where a given user has an account, but is not currently authenticated. The user may still use the shop and put articles into their cart across several requests. Once they are done shopping, the client may decide to actually authenticate to the online shop to order the items in their cart. At this point, the session identifier that has been used by the user while shopping is internally mapped to the given user account and considered as authenticated by the server. Thus, a session identifier may correspond to an authenticated or an unauthenticated session, and this status (i.e., authenticated or unauthenticated) can change when a client logs in or out.

For an attacker, learning the session identifier of an honest client authenticated to a web application is clearly an interesting objective, as it allows them to impersonate that client and act on their behalf. Broadly speaking, there are three ways for an attacker to achieve that goal: (1) They may be able to

```php
<?php
if( isset( $_GET[ 'PHPSESSID'] ))
  session_id( $_GET[ 'PHPSESSID']);

session_start();

// do something
?>
```

Figure 2.8: Example of a session fixation vulnerability.

predict session identifiers in some way, e.g., because the random generator used by the web application is not secure; (2) they may *capture* a session identifier used by a client, e.g., they may exploit a browser vulnerability on the client side, or a network attacker may eavesdrop on an unencrypted communication; and (3) finally, they can attempt to *fixate* the session identifier, that is, trick a victim into using a particular session identifier chosen by the attacker, wait for the victim to authenticate with that session identifier, and then use that session identifier to impersonate the victim via-à-vis the web application. The last approach is one that may be caused by insecure code and the one we want to discuss in in this section.

Fixating session identifiers is particularly easy for an attacker when a web application is written in such a way that clients may not only send their session identifiers to the web application as cookies, but also (or even exclusively) as GET or POST parameters. This is a bad idea from a security standpoint, yet it is a practice commonly observed. Consider for instance the code snippet in Figure 2.8. The application checks whether a GET parameter called `PHPSESSID` was passed, and if so, internally sets the session identifier to the passed value. Then, it starts the session (which means here that a global array `$_SESSION` is populated with user data stored by the server for the given session identifier, and that a `Set-Cookie:` field is sent in the HTTP header of the response to the client to cause it to store the session identifier as a cookie). Now, all an attacker has to do is trick a victim into clicking a link such as `http://example.com/index.php?PHPSESSID=abad1dea` to the above page. This will cause the victim's session identifier to be set to `abad1dea`, which the attacker knows. If the victim now authenticates to the web application, the attacker will have the session identifier of an authenticated user and therefore will be able to impersonate the user.

In earlier PHP versions (i.e., prior to PHP 7), the kind of behavior depicted in Figure 2.8 was actually the default behavior upon simply calling `session_start()`: If no session identifier was defined by a cookie, this built-in method would then try to find one in the GET or POST parameters. Fortunately, this is not any longer the case. Note that, even if only session identifiers via cookies are accepted by an application, this does not utterly thwart session fixation attacks. For instance, if a call such as `set_cookie($name,$value)` is used somewhere in the code and the attacker can control both the name and the value of the cookie via GET or POST parameters, they can use it to implement a session fixation attack in a place completely unexpected by the developer, by exploiting this call to set a session identifier cookie in the victim's browser. Additionally, even if an application is implemented in such a way that it only accepts session identifiers that it generated itself, this does not help against session fixation attacks at all: The attacker may simply connect to the web application themselves, obtain a valid session identifier and use it for their purposes.

A reliable approach to securely defend against session fixation attacks is to regenerate a new session identifier upon each new request by a client (and internally remap all associated user data with the new session identifier). While this is easy to do when developing a new application, integrating this feature into an already existing complex application can be very cumbersome. For this case, some alternate, more involved approaches have been proposed [Johns *et al.* 2011]. Session fixation attacks should not be underestimated, as they have a critical impact on the security of an application, but are much less known than, for example, the rather popular SQL injection attacks, and thus receive significantly less attention [Johns *et al.* 2011].

2.3.3 Sensitive Data Exposure

In this last section on web application vulnerabilities, we discuss a vulnerability that is technically slightly different from the ones previously discussed. While the vulnerabilities discussed earlier in this chapter assume an attacker who either directly sends input to a server, or else causes an innocuous client to send a given input to a server (e.g., by having them click on a link), the vulnerability type presented in this section assumes a passive network attacker, i.e., an attacker that is able to passively listen in on messages being exchanged over a network, but does not actively send any kind of message to any network party on their own. Such attackers are not limited to powerful entities such as governments' intelligence services. In fact, freely available tools such as Wireshark[3] exist that allow even technically non-savvy users to log all messages sent over a local network, such as, say, at a company or at a university.

Since messages in HTTP are sent in plaintext, this protocol is not suitable to exchange sensitive data, such as passwords, credit card numbers, etc. Consider for instance the code snippet in Figure 2.9. It is a JavaScript function that

[3] https://www.wireshark.org

```
function login() {
  var url = "http://example.com/login";
  var username = document.getElementById( "username").value;
  var password = document.getElementById( "password").value;
  var params = "user="+username+"&pass="+password;
  var request = new XMLHttpRequest();

  request.onload = function() {
    if (request.status >= 200 && request.status < 400) {
      // process successful answer
    }
    else {
      // process server error
    }
  };

  request.onerror = function() {
    // process connection error
  };

  request.open( "POST", url);
  request.setRequestHeader( "Content-type", "application/x-www-form-urlencoded");
  request.send( params);
}
```

Figure 2.9: Example of sensitive data exposure.

sends a login request to a server using `XMLHttpRequest`, with a username and a password provided by a user. The problem is that the URL the login data is sent to is an HTTP resource, and, hence, the submitted data is sent in plaintext. Therefore, it exposes the username and the password to anyone listening on the network, allowing an attacker to impersonate victims who use this application to authenticate.

The appropriate countermeasure is encryption. Sensitive data should only be sent over the network in an encrypted way, either using a secure protocol such as HTTPS, or by manually encrypting the sensitive data using a strong encryption scheme before sending it over a public network. In practice, however, this is not always trivial. For instance, while the vulnerability is obvious in the above example, in many cases the URL would not be hardcoded, but relative to the current domain, that is, the second line would simply read `var url = "/login";`. In this case, the code is fine as long as it is run in an HTTPS context, but not in an HTTP context. If this software is distributed to hundreds or thousands of users who deploy it on their own servers, not all of them may be aware of this problem. Even if they are aware of the problem, they may lack the technical knowledge, the time or the money to obtain a valid SSL/TLS certificate. As a result, many of them may make the application accessible via HTTP. Accordingly, forums, blogs, and other online communities with unencrypted login forms are commonly found on the web. Adding to this problem is the fact that most users only have one or two passwords that they re-use on various sites. Then, when they authenticate without encryption to some web application, an attacker may compromise their identities for other, normally well-secured, web applications as well.

2.4 Characteristics of Vulnerable Code

In this chapter, we discussed several security flaws in web application code that lead to attacks threatening either servers or clients. In order to build automated tools that help in identifying these flaws, we firstly need to identify the key properties that characterize them. These properties should be formulated abstractly enough to encompass all of the above flaws, but concretely enough so that when piece of code exhibits these properties, a vulnerability is likely.

Looking at the flaws presented in this section, we notice that they are all caused by the *propagation* of data of some kind through the program. More concretely, in all cases but the sensitive data exposure discussed in the last section, the attacker controls some data input to the program. This input then flows to a security-critical function call without being properly validated or sanitized. That is, we identify the following three re-occurring characteristics:

1. **Attacker-controllable source.** The *source* of the data, i.e., the input to the program, can be controlled by the attacker. Either an attacker sends input directly to a server (e.g., SQL injection, arbitrary file accesses), or causes a victim to send the attacker's input to the server (e.g., reflected cross-site scripting, session fixation).

2. **Insufficient validation or sanitization.** The web application does not validate that the input has an expected format, nor sanitizes the input in such a way that it cannot be harmful, or does so only insufficiently. As is typical for security flaws, the key problem is that developers tend to develop applications such that for some expected input, something expected happens; yet in order to write secure applications, they must ensure that for *any* unexpected input, nothing unexpected happens.

3. **Sensitive sink.** Finally, data originating in the attacker-controllable source propagates to a sensitive *sink*, i.e., a security-critical operation of some kind, such as an SQL query or a system shell call which the attacker leverages to conduct a successful attack.

In the case of the sensitive data exposure discussed in the last section, the vulnerability is equally characterized by an undesirable propagation of data through the program. However, in this case, the situation is reversed: It is

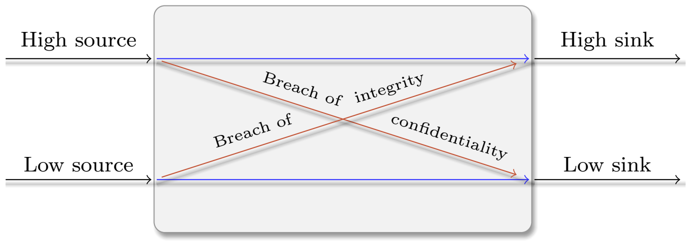

Figure 2.10: Undesirable propagation of data through a program.

the *source* of the data that is sensitive, and this sensitive data flows, without an appropriate means of protection, to a *sink* that the attacker can observe. In other words, the characteristics that we identified for the other cases are mirrored:

1. **Sensitive source.** The data in question is only known to the application itself and possibly to the honest client using the application, such as a key or a password. This data should not be exposed to outside observers.

2. **Insufficient protection.** The data is not protected properly, e.g., by means of encryption. In certain cases, a certain amount of leakage may be considered as acceptable. For instance, a password checker inherently leaks the information whether a given password is correct or not. (Some works are concerned with quantifying the amount of leaked information in the communication between two processes [e.g., Backes *et al.* 2009]).

3. **Attacker-readable sink.** Lastly, the sensitive data propagates to an attacker-readable output. This may be a public network, for example, but may also be an unintentional bit of information buried within an output intended for the attacker.

This duality of undesirable data propagation in a program was first observed by Biba [Biba 1977]. In terms of programming language theory, we say that a flow from *low* (i.e., untrusted) data to *high* (i.e., sensitive) data without a proper means of *endorsement* (i.e., validation or sanitization) corresponds to a breach of *integrity* of the application. Dually, a flow from high data to low data without proper *declassification* (such as encryption) of the data corresponds to a breach of *confidentiality* [Sabelfeld & Myers 2003, Askarov & Myers 2010]. Figure 2.10 depicts the idea of undesirable data flow though a program: Ideally, high data should only propagate to high data, and low data to low data. Whenever there is a flow from low data to high data or vice versa, endorsement (resp. declassification) of the data is needed, or a vulnerability may result.

We will explore techniques to identify vulnerabilities that compromise the integrity or the confidentiality of a web application. We focus on JavaScript on the client side and PHP on the server side. Nevertheless, the techniques described here can be applied to other languages as well. In the next chapter, we will first review some technical background on various kinds of representations of application code which these techniques are based on and which will be required in the remainder of this thesis.

Chapter 3

Program Representations

Contents

Meaningful program analysis geared towards vulnerability discovery requires suitable representations of the code to be analyzed as a foundation. Clearly, any such analysis can only be as good as its view on the application code permits. We should therefore strive to devise program representations that are both comprehensive and as complete as possible. Fortunately, we need not start from scratch, as a variety of avenues to model application code with different purposes have been explored in the past. Most of these are rooted in the design and construction of compilers. A standard textbook on the principles and techniques used in compiler design is the *Dragon Book* (called thus because of its memorable cover design) by Aho *et al.* [Aho *et al.* 2006]. For a deeper understanding of the structures presented in this chapter, we refer the reader to this work.

Firstly, we shall discuss how to model a program's syntactical structure in Section 3.1. This is the canonical place to start, as all the other views that we discuss thereafter are eventually computed from the *abstract syntax trees* presented in that section. In addition, some types of vulnerabilities may be discovered at a purely syntactical level, as we will see. Next, to understand the propagation of either sensitive or attacker-controllable data throughout a program, two views are particularly useful. First, *control flow graphs*, which we discuss in Section 3.2, model the possible execution orders of statements in a program as well as the conditions under which a given path is taken. Second, *program dependence graphs*, presented in Section 3.3, expose dependencies between statements and predicates as well as data dependencies between statements and enable us to explore the flow of data in a program. Finally, both control flow graphs and program dependence graphs are defined at a function level only. Therefore, while they constitute powerful tools for reasoning about control and data flow, they only enable us to do so at a (local) procedural level. As the vast majority of programs are composed of hundreds of functions and vulnerabilities arising from unexpected propagation of data typically span across several function calls, we discuss *call graphs* in Section 3.4 to remedy this problem and enable interprocedural analysis for vulnerability discovery.

Throughout this chapter, we will use the PHP code in Figure 3.1 as a running example. We saw this very code when discussing SQL injections in Section 2.2.1. While chosen to be as simple as possible for the sake of presentation, it nevertheless illustrates the underlying ideas of all the structures presented in this chapter. It presents a function `foo` that reads a GET parameter `id` and assigns it to a local variable `$x` (line 4). If this parameter is set (line 6), a string containing an SQL query is constructed (lines 7-8) and this string is passed to another function `query` (line 9), responsible for sending queries to the database back end.

```
1   <?php
2   function foo() {
3
4     $x = $_GET["id"];
5
6     if(isset($x)) {
7       $sql = "SELECT * FROM users
8               WHERE id = '$x'";
9       query($sql);
10    }
11  }
12  ?>
```

Figure 3.1: Code of the running example.

3.1 Syntactical Representations

In this section, we discuss two syntactical representations of code: First, *parse trees* are trees constructed by a parser from the source code of an application and reflect the exact structure of the source code. Second, *abstract syntax trees* are derived from parse trees. They contain all semantically relevant information about the source code, but abstract away from syntactical details.

3.1.1 Parse Trees

The syntax of a programming language is usually specified as a grammar, most commonly a context-free grammar. Such a grammar naturally describes the hierarchical structure of all the language constructs of the language and can be used to derive any valid program. A context-free grammar is defined by the following four components [Aho *et al.* 2006]:

1. A set of *terminal symbols*. These are the elementary symbols defined by the grammar, such as keywords, punctuation symbols, or strings.

2. A set of *non-terminal symbols*. Non-terminal symbols describe a set of sequences of non-terminal and terminal symbols. Eventually, any non-terminal symbol can be converted into a sequence consisting only of terminal symbols.

3. A set of *production rules*. Each production rule consists of a left side, an arrow, and a right side. The left side consists of exactly one non-terminal symbol (this is a distinguishing characteristic of a *context-free* grammar). The right side describes one or more sequences of non-terminal and terminal symbols that the left side can be rewritten into.

4. A *start symbol*, which is one of the non-terminal symbols.

To derive a program, a grammar starts with the start symbol and repeatedly applies production rules for each non-terminal until a string consisting only of terminal symbols is obtained. The set of all derivable strings constitutes the language defined by the grammar.

As an example, Figure 3.2 shows a subset of the actual PHP grammar. This subset is chosen in such a way that the production rules shown illustrate how to derive the PHP code of our running example. For the sake of presentation, the subset of the PHP grammar given in Figure 3.2 elides some details (such as the optional possibilities of adding documentation comments or return type hints when declaring functions) and simplifies some intricacies of the language,

FUNC	→	function *STRING* (*PARAM_LIST*) { *STMT_LIST* }
PARAM_LIST	→	*NON_EMPTY_PARAM_LIST* | ϵ
NON_EMPTY_PARAM_LIST	→	*PARAM* | *NON_EMPTY_PARAM_LIST*, *PARAM*
PARAM	→	...
STMT_LIST	→	*STMT_LIST STMT* | ϵ
STMT	→	*EXPR*; | { *STMT_LIST* } | *IF_STMT* | ...
IF_STMT	→	*IF_STMT_WITHOUT_ELSE*
	|	*IF_STMT_WITHOUT_ELSE* else *STMT*
IF_STMT_WITHOUT_ELSE	→	if (*EXPR*) *STMT*
	|	*IF_STMT_WITHOUT_ELSE* elseif (*EXPR*) *STMT*
FUNC_CALL	→	*NAME* (*ARGUMENT_LIST*) | ...
NAME	→	...
ARGUMENT_LIST	→	*NON_EMPTY_ARGUMENT_LIST* | ϵ
NON_EMPTY_ARGUMENT_LIST	→	*ARGUMENT* | *NON_EMPTY_ARGUMENT_LIST*, *ARGUMENT*
ARGUMENT	→	...
EXPR	→	*VAR* | *VAR* = *EXPR* | *FUNC_CALL* | ...
VAR	→	$*STRING* | $*STRING*[*EXPR*] | ...

Figure 3.2: Simplified subset of the PHP grammar.

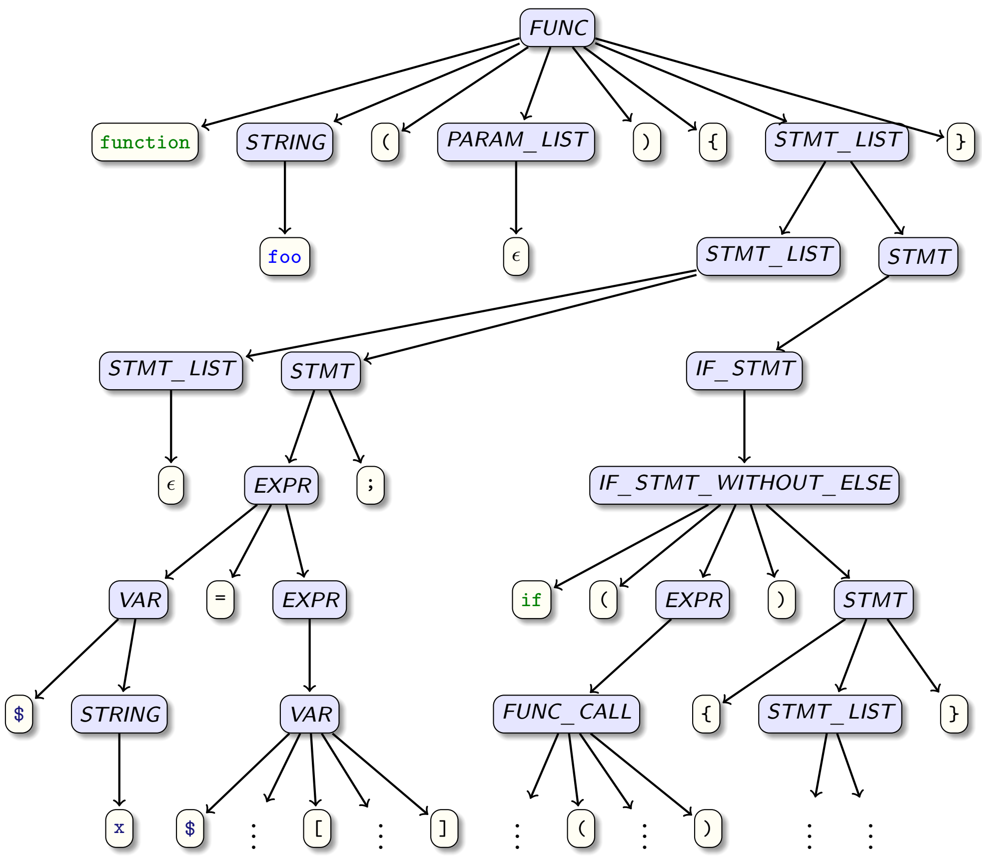

Figure 3.3: Parse tree of the running example.

but is faithful otherwise.[1] We use the symbol | to separate different sequences of symbols that a single non-terminal may be rewritten into. The sequence of zero terminals, called the *empty string*, is designated by ϵ.

The inverse process of taking a sequence of terminal symbols and determine whether and how it can be derived from the start symbol of the grammar is called *parsing*. Parsing constitutes one of the most fundamental problems in compiler design. The first step of a compiler's front end (after running the tokenizer) is to parse the tokenizer's output. For a syntactically valid program, a *parse tree* can be generated by creating a node for each encountered symbol and connect it to the node representing the non-terminal symbol that produced it. Formally, a parse tree is defined as follows [Aho *et al.* 2006]:

1. The root node corresponds to the start symbol.
2. Each interior node corresponds to a non-terminal symbol.
3. Each leaf node corresponds to a terminal symbol.
4. If an interior node corresponds to non-terminal symbol S and this node has children corresponding to symbols S_1 through S_n (in that order, from the leftmost child to the rightmost child), then the grammar contains a production rule $S \to S_1 \dots S_n$.

Parse trees are sometimes called *concrete syntax trees* to distinguish them from the *abstract syntax trees* which we discuss in the next section. Figure 3.3 shows (a part of) the parse tree of our running example according to the grammar given in Figure 3.2. Here, the leaf nodes (corresponding to terminal symbols) are colored yellow and the interior nodes (corresponding to non-terminal symbols) are colored blue. For clarity of presentation, the terminal symbols are colored such that they match the highlighting in Figure 3.1. We do not show the complete parse tree for reasons of space and because doing so would not add any new information.

While this representation of code certainly makes it easier to recognize patterns that may correspond to vulnerabilities than working on pure text, the parse tree's verbosity and clumsiness make this task cumbersome. In the next section, we therefore discuss *abstract syntax trees* which are a more succinct and elegant representation that significantly eases this task.

3.1.2 Abstract Syntax Trees

At first sight, abstract syntax trees (ASTs) resemble the parse trees discussed in the previous section. However, as opposed to parse trees, instead of reflecting

[1] The full PHP grammar is specified in a Bison/Yacc file within the source code of the PHP framework [PHP Group 2017a].

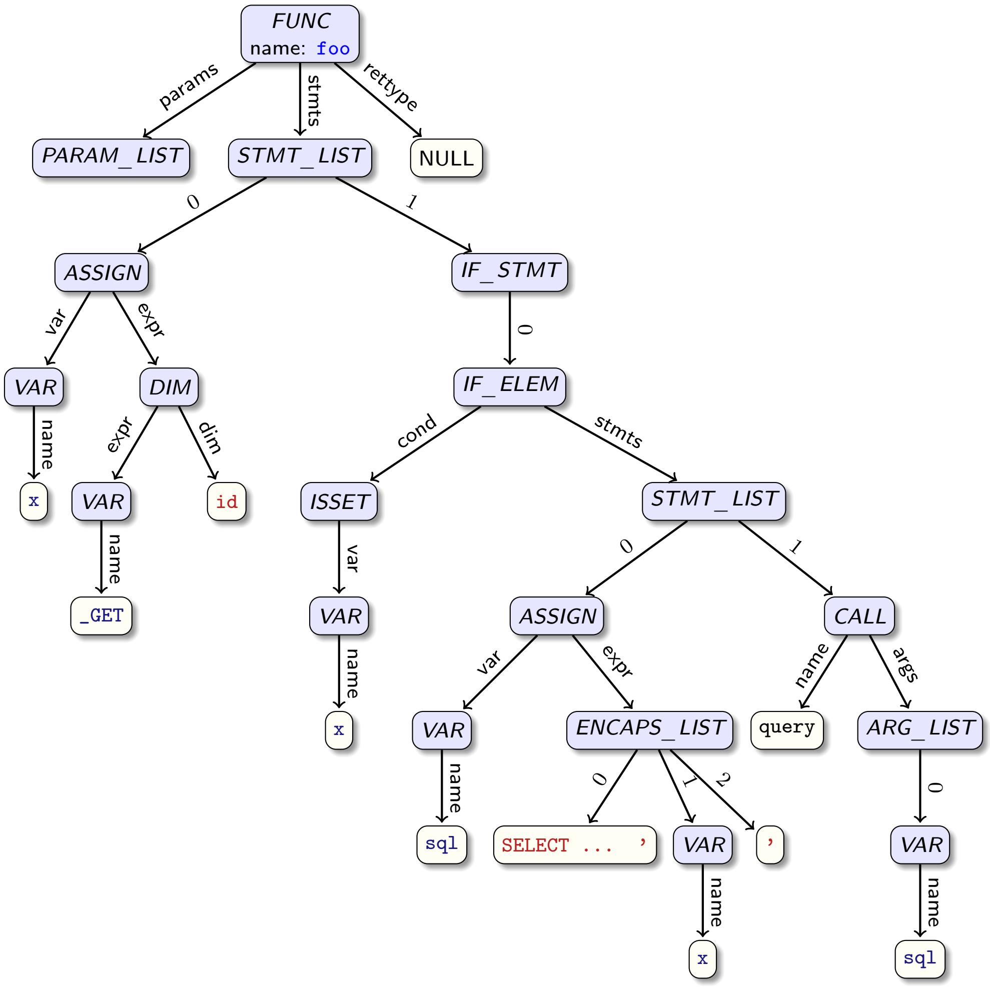

Figure 3.4: Abstract syntax tree of the running example.

the parsed code to the letter, they constitute an *abstraction* of a program's syntax in the sense that they abstract away from details of formulation and idiosyncrasies of a language and retain only the structural elements that make up the code. ASTs can be obtained from parse trees by walking the parse tree and recursively applying appropriate translation rules [Aho *et al.* 2006].

Figure 3.4 shows the complete AST of our running example. Barring a few technicalities (e.g., flags on the function node that reflect modifiers such as `public` or `private`), this is the very syntax tree of our running example as generated by the internal parser of the actual PHP interpreter.[2]

When comparing the parse tree from the previous section with the abstract syntax tree in this section, we notice several differences. First, in an

[2]This is still in a beta stage. See the corresponding PHP RFC for abstract syntax trees [PHP Group 2014].

abstract syntax tree, interior nodes represent programming constructs instead of non-terminals. For instance, the first assignment in our running example (`$x = $_GET["id"]`) was a statement node in the parse tree and the assignment operator itself was a child of that statement (and a leaf node). By contrast, in the abstract syntax tree, there is a distinguished assignment node: In effect, the assignment operator has been moved up in the hierarchy. Second, language keywords and punctuation symbols have been abstracted away. Third, some nodes have been collapsed into a single node. For instance, nodes representing a list of statements, labeled *STMT_LIST* both in the parse tree and the abstract syntax tree, can have an arbitrary number of children in the AST, each representing a statement in the list, instead of using a nested structure as is the case in the parse tree. This holds for any node in Figure 3.4 that has outgoing edges labeled with digits (and for the node *PARAM_LIST* too, which happens to have no children here), including the node *IF_STMT* which can have one or more children of type *IF_ELEM*, each corresponding to a single `if`/`elseif`/`else` block in a given if-statement. All other nodes have a fixed number of children. The exact number depends on the type of node. For instance, a *FUNC* node always has exactly three children representing its parameter list, its body, and its return type, respectively. Here, the return type is unspecified and hence the node representing the return type is a *NULL* node, yet it is present for structural integrity. Lastly, nodes may have properties. In our example, the *FUNC* node has a property defining its name. Which node-defining characteristics are represented as a property on the node and which characteristics are represented as children of the node depends on the used definition of the abstract syntax tree, which is in turn dictated by the practical needs in a given context. As a rule of thumb, however, properties are typically used for simple, primitive characteristics such as a function's name.

Note that edges are not commonly labeled in ASTs. We have done so in Figure 3.4 for clarity of presentation. However, a child node's role is uniquely defined by its index under its parent node, hence there is no need for edge labels.

Abstract syntax trees may be implemented slightly differently by different parsers, even for the same language. However, the idea is always to abstract away from the concrete syntax and retain only the structural information. As such, ASTs are an intermediate representation of code: A compiler may have different front ends for several languages that all generate the same type of AST, and the compiler's back end needs only know how to translate this AST to a given machine code. Similarly, in the context of vulnerability detection, we can use ASTs to detect patterns that correspond to vulnerabilities *independently* of the language, as long as two languages can be compiled into the same type of AST and give rise to the same vulnerability patterns.

```php
<?php
if( hash( 'sha1', $_GET['pwd']) == stored_hash)
  login();
?>
```

Figure 3.5: Magic hash vulnerability.

Even though ASTs do not contain semantic information such as control or data flow, some types of vulnerabilities may be detected at a purely syntactical level. As an illustration, consider the PHP code in Figure 3.5. It illustrates the problem of *magic hashes* [Hansen 2015]: When the == operator is used instead of the === operator, PHP attempts to perform some type conversions depending on the value of the compared variables. In our example, if the stored hash happens to start with the characters `0e` (both of which are valid hexadecimal characters), PHP will assume that it should convert the string to a number by somehow casting the rest of the string into a number, then taking 0 to the power of this number, yielding the final result 0. Hence, all an attacker needs to do to exploit this code is find a password that has a SHA1 hash starting with `0e` (which happens with a probability of $1/16^2 = 1/2^8$, i.e., it is very easy to find). This password can then be used as a master password for all users who have a password that happens to hash to a value starting with `0e` too.

This kind of vulnerability can easily be detected at a syntactic level. We simply search the AST for the following pattern:

- There is a node representing the == operator.
- One of the two children of this node must be a node representing a call to the function `hash`.
- The second child of the argument list child node of this function call node must be an array access to an attacker-controllable variable such as `$_GET` or `$_POST`.

Depending on the context, these conditions do not absolutely guarantee that there is a vulnerability, but if a match is found, a vulnerability is not unlikely. Some other types of vulnerabilities, such as integer overflows, can also be found at a purely syntactical level [Yamaguchi *et al.* 2014].

In practice, however, most types of vulnerabilities require a deeper understanding of the control and data flow of the given program. In particular, it is usually necessary to be able to reason about the flow of attacker-controlled data. Therefore, we introduce additional structures in the next sections that model control flow and data dependencies in a program.

3.2 Modeling Control Flow

In this section, we discuss structures that reflect properties of the control flow of an application. First, we look at control flow graphs, which model the control flow itself. Subsequently, we discuss the notions of dominance and post-dominance and present dominator and post-dominator trees, which express slightly more intricate properties derived from the control flow graph.

3.2.1 Control Flow Graphs

Abstract syntax trees emphasize the syntactical structure of a program, but they cannot directly be used to reason about the program's control flow, i.e., the order in which statements are executed: This type of reasoning requires semantic information about the constructs of a given programming language. The idea of using a directed graph to express control flow relationships in a program so as to enable control flow analysis was pioneered by Turing award winner Frances Allen in 1970 [Allen 1970]. This type of graph is known as the *control flow graph* (CFG).

In an imperative programming language, the *statements* of a procedure or function are executed sequentially. Boolean expressions are used in the guard of `if`, `while`, and similar statements to alter the flow of control based on conditions evaluated at runtime. We refer to such boolean expressions as *predicates*. A control flow graph defines a node for each statement and predicate of a function. The edges of the graph reflect the interplay of the statements and predicates, i.e., the order in which they are evaluated. Labels on the edges indicate the conditions that result in a given control flow: Edges originating in statements are labeled with ϵ to denote unconditional control flow, and edges originating in predicates are labeled with true or false to indicate conditional control flow. A control flow graph is defined *per function*, and artificial *ENTRY* and *EXIT* nodes are defined by the graph to indicate the entry and exit points of that function.

Figure 3.6 shows the control flow graph of the function `foo` from our running example. The arrow from the *ENTRY* node to the assignment `$x = $_GET["id"]` indicates that this statement is executed first. Next, the predicate `isset($x)` is evaluated. If it evaluates to true, control is transferred to the first statement of the if-statement's body. Otherwise, the end of the function is reached, as denoted by the arrow to the *EXIT* node.

Control flow graphs can be easily extended to account for more advanced programming constructs. For instance, one can introduce edges labeled exception to model the flow of control inside `try`/`catch` statements. CFGs can be directly computed from ASTs by defining appropriate rules for each type of

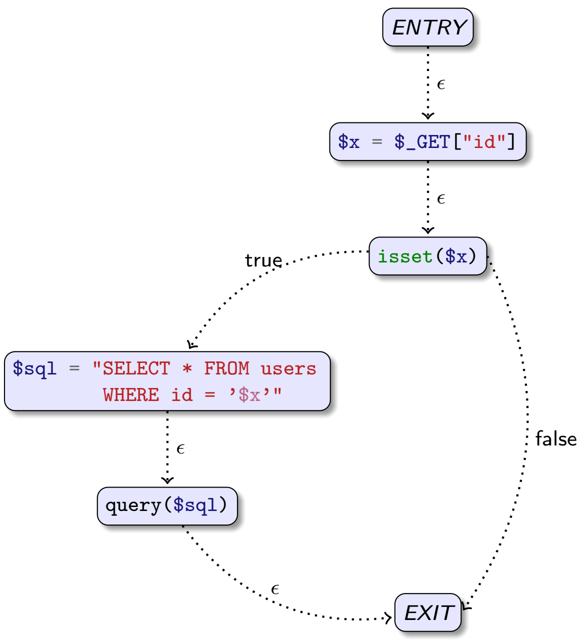

Figure 3.6: Control flow graph of the running example.

construct that the language provides and then applying the following two-step algorithm [Yamaguchi 2015]:

1. First, edges are computed for structured control flow statements such as `if`, `while` or `for` statements, using a recursive tree-walking algorithm that implements appropriate rules for all types of nodes of the abstract syntax tree.

2. Subsequently, the graph is corrected to take into account unstructured control flow statements such as `break`, `continue` or `goto`. This is now easily possible since the entry and exit points of loops have been computed in the previous step, and the rule for nodes denoting `label` statements in the previous step can also store the labels and their corresponding nodes to handle `goto` statements in this step.

Control flow graphs can be used to identify control flows that may lead to certain vulnerabilities. For instance, consider the example in Figure 3.7. Here, a user can send a message that is appended to a guestbook file on the server, unless the message is too long. Unfortunately, if the message is too long, the file resource is not properly closed. In any case, a lengthy process is called later on. Here, an attacker can perform a denial-of-service attack by calling the script many times in parallel with long messages, causing the server to open many file resources until its limit is reached. Although this example

```php
<?php
$handle = fopen( "guestbook.txt", "a");

if( strlen( $_GET["message"]) > 10000) {
  echo "Your message is too long!";
}
else {
  fwrite( $handle, $_GET["message"]);
  fclose( $handle);
}

lengthy_process();
?>
```

Figure 3.7: Denial-of-service vulnerability.

is somewhat contrived, in more complex applications, a situation where a resource is not properly closed before a long process is called is realistic.

This vulnerability can be expressed by the following pattern in the control flow graph:

- There is a call to `fopen`.

- There is a path from the call to `fopen` to the *EXIT* node that does not contain a call to `fclose`.

Ideally, from the attacker's perspective, the path should also contain a predicate that uses an attacker-controlled variable, such as `$_GET["message"]` in our example, as the attacker needs to be able to cause the program to actually take the vulnerable path. However, this is not a necessity in general, as a program may possibly take the vulnerable path without any input.

Control flow graphs allow us to reason about certain kinds of control flow type vulnerabilities such as the one in Figure 3.7. However, detecting this vulnerability requires us to inspect all paths from `fopen` to *EXIT* so as to find one that does not contain `fclose`. This kind of inspection can be prohibitively expensive for complex applications where it is necessary to take into account loops as well as a multitude of nested levels of control flow branchings (leading to an exponential growth of the number of possible paths). Thus, a representation better suited for efficiently detecting this type of vulnerability would be convenient. To this end, we discuss the notions of dominance and post-dominance in the next section. In addition, computing post-dominance relationships is also a prerequisite for calculating more involved program representations, as we discuss in Section 3.3.

3.2.2 Dominator and Post-dominator Trees

Automated discovery of vulnerabilities in program code often involves determining whether some statement is executed before or after some other statement on all possible execution paths through a program. For instance, it is interesting to determine whether a validation or sanitization function is always called on some attacker-controlled input before that input is used in a sensible operation to avoid injection-type attacks (see Chapter 2), or, conversely, whether a resource is always freed after having been allocated as in the example of Section 3.2.1.

The notions of *dominance* and *post-dominance* express exactly this kind of dependence for control-flow graphs: We say that a node d of the control flow graph *dominates* another node s iff every path from *ENTRY* to s contains d. Conversely, we say that a node p *post-dominates* a node s iff every path from s to *EXIT* contains p. Note that by definition, a node always dominates and post-dominates itself. The notion of dominance was first introduced by Prosser [Prosser 1959] as a unification of both definitions.

We can express dominance and post-dominance in tree structures by extending these definitions to the notions of *immediate* dominance and post-dominance [Lowry & Medlock 1969], respectively. We say that for two nodes d and s such that $d \neq s$, d is an *immediate dominator* of s iff:

1. d dominates s; and
2. there exists no node n different from d and s such that d dominates n and n dominates s.

In the *dominator tree*, each node's children are the nodes that it immediately dominates. Every node except *ENTRY* has exactly one immediate dominator [Lowry & Medlock 1969], and hence, the tree is uniquely defined. The notions of *immediate post-dominance* and the resulting *post-dominator tree* are defined analogously [Ferrante *et al.* 1987].

Figure 3.8 and Figure 3.9 show the dominator tree and the post-dominator tree of our running example, respectively. We can see that `isset($x)` immediately dominates both the assign statement of `$sql` and the *EXIT* node, since both statements are immediately preceded by this predicate on any program path. The statement `query($sql)`, on the other hand, does not dominate any other node, since the *EXIT* node is not necessarily preceded by that statement. In the post-dominator tree, we see that the *EXIT* node immediately post-dominates both `query($sql)` and `isset($x)`, as any program path executes either of these nodes immediately before reaching the *EXIT* node. The assignment statement of `$sql` does not post-dominate any other statement, since it does not necessary follow the predicate `isset($x)`.

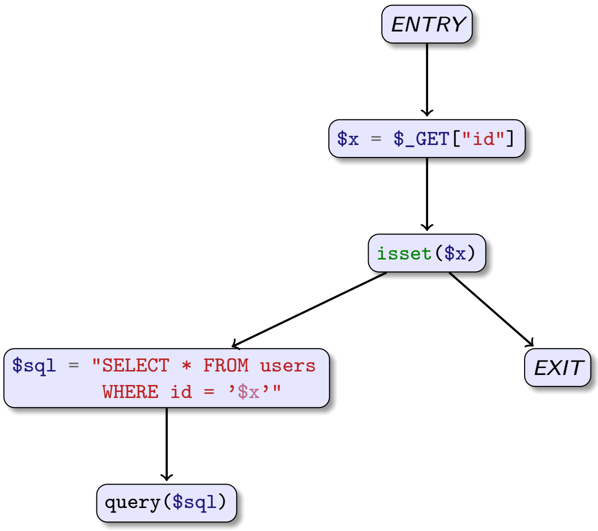

Figure 3.8: Dominator tree of the running example.

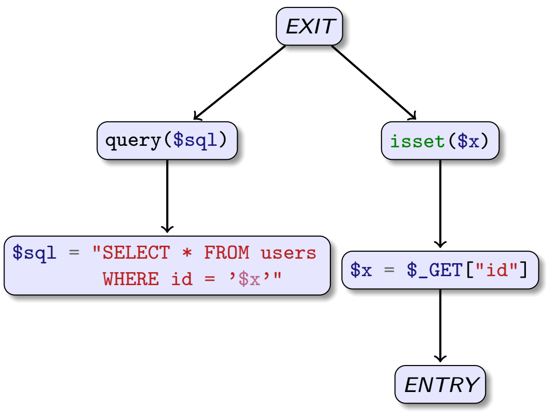

Figure 3.9: Post-dominator tree of the running example.

Various algorithms have been proposed to compute dominators [Allen & Cocke 1972, Purdom & Moore 1972, Aho *et al.* 2006, Lengauer & Tarjan 1979, Cooper *et al.* 2006] (note that the citation of the book by Aho *et al.* refers to the second edition, but the first edition was published in 1977). The algorithm by Lengauer and Tarjan has an asymptotic complexity of $O(m \cdot \log(n))$, where m is the number of edges and n is the number of vertices, but it is highly involved. A much simpler algorithm which is presented by Aho *et al.* and goes back to work by Allen and Cooke on data-flow analysis is shown in Figure 3.10.

```
// The ENTRY node dominates only itself.
Dom(ENTRY) = {ENTRY}
// Set all nodes as dominators for all other nodes.
for each v in V \ {ENTRY}
    Dom(v) = V
// Iteratively remove nodes that are not dominators.
// preds(v) is the set of predecessors of v in the control flow graph.
while changes to any Dom(v) occur
    for each v in V \ {ENTRY}
        Dom(v) = ( ⋂_{p∈preds(v)} Dom(p) ) ∪ {v}
```

Figure 3.10: Algorithm for computing dominators in a CFG [Aho *et al.* 2006].

In fact, this is an instance of a classic data-flow analysis algorithm, of which we shall see another instance in Section 3.3.2. This algorithm has an asymptotic running time of $O(n^2)$, however, Cooper *et al.* show that using carefully chosen data structures, it can actually outperform the algorithm by Lengauer and Tarjan in practice.

The dominator tree can be used to determine whether a given statement, such as a validation or a sanitization function, is always executed before some other statement that corresponds to a sensitive operation, by checking whether the sensitive statement is dominated by the validation or sanitization function. The post-dominator tree of a control flow graph corresponds to the dominator tree of the reversed control flow graph, therefore, its computation is straightforward too. It can be used to check whether some statement is always executed after another, for instance to check for vulnerabilities resulting from failure to free resources such as the vulnerability discussed in Section 3.2.1. In addition, the post-dominator tree also plays a key role to compute control dependencies, as we will see. While post-dominator trees are useful to detect certain types of vulnerabilities, the vast majority of vulnerabilities requires us to be able to reason about the propagation of attacker-controlled data in a program. In the next section, we discuss a program representation that allows us to expose statement dependencies, in particular statement dependencies caused by data flows.

3.3 Statement Dependencies for Intraprocedural Analysis

In this section, we discuss *program dependence graphs* (PDGs), a powerful tool for reasoning about the flow of attacker-controlled data. This type of graph was first introduced by Ferrante *et al.* [Ferrante *et al.* 1987] with the original intent to make compiler optimization more efficient (many of the typical optimizations done by compilers operate more efficiently on the PDG than on the CFG) but is also useful in other contexts such as program slicing [Weiser 1981] and, in our case, for automated vulnerability detection.

As for the control graph, the nodes of the PDG are the statements and predicates of a program. It makes explicit the essential control flow and data flow relationships of a program. To this end, it contains two types of edges exposing *control dependence* and *data dependence*, respectively. We briefly discuss these two types of dependencies in the next two sections.

3.3.1 Control Dependencies

Control flow graphs express the control flow relationships in a program, however, they also contain a potentially unnecessary sequencing of operations. For instance, if our running example in Figure 3.1 contained an additional statement at the beginning to store another GET parameter to be used in a subsequent query—say, some search term—it would not make any difference which statement is executed first, i.e., whether the code started with `$x = $_GET['id']; $y = $_GET['search'];` or with the assignments in reversed order as in `$y = $_GET['search']; $x = $_GET['id'];`; the statements do not depend on each other.

A *control dependence* arises between a predicate and a statement when the execution of the statement depends on the result of the evaluation of the predicate. For instance, in our running example, the evaluation of the statement `query($sql)` depends on the evaluation of the predicate `isset($x)`. The control dependence edges of a PDG expose this type of relationship and are labeled with either `true` or `false` to express the value that the predicate must evaluate to for the dependent statement to be executed. As an example, the control dependence edges for our running example are depicted as dash-dotted edges in Figure 3.12.

Formally, we define control dependence as follows. Let v and w be two distinct nodes in a control flow graph. We say that w is *control dependent* on v iff:

1. There exists a path from v to w in the CFG such that all nodes on the path (excluding v) are post-dominated by w; and

2. v is *not* post-dominated by w.

The reasoning is as follows. If w is control-dependent on v, then v has two exits, one which leads to the statement w to be executed, and one which does not. All statements on the path between v and w (except v) are post-dominated by w (note that if w immediately follows v on that path, the condition is fulfilled, since w post-dominates itself). However, v is not post-dominated by w, since it has an exit that does not lead to w being executed.

Control dependencies can be computed efficiently by computing the *dominance frontier* of each node in the reversed control flow graph [Cytron *et al.* 1989]. The dominance frontier of a node v is the set of nodes such that the following holds. For every node d in the dominance frontier of v, we have that $d \neq v$ and there exists a path from v to *EXIT* though d such that d is the first node not dominated by v. Computing the dominance frontier of a node requires both the control flow graph and the dominator tree. Since we need to compute the dominance frontier of every node in the *reversed* control flow graph and the dominator tree of the reversed control flow graph corresponds to the post-dominator tree of the control flow graph, computing control dependence edges in effect requires calculating the control flow graph and the post-dominator tree.

3.3.2 Data Dependencies

Besides control dependencies, another dependence between statements enforcing a sequential evaluation exists that is not related to control flow. Consider our running example in Figure 3.1. The assignment of the variable `$sql` is not control dependent on the assignment of the variable `$x`. However, the definition of `$x` must necessarily be executed first, since the value of `$x` is used in the definition of `$sql`.

A dependence between statements caused by the fact that one of the two statements *defines* the value of a variable subsequently *used* in the other statement is called a *data dependence*. In our example, the call to `query($sql)` is data dependent on the definition of `$sql`, which is in turn data dependent on the definition of `$x`. Additionally, the predicate `isset($x)` is also data dependent on the definition of `$x`. The data dependence edges in a PDG expose this relationship. These edges are labeled with the name of the variable being defined in the source statement and used in the target statement. Figure 3.12 presents the data dependencies for our running example as dashed edges.

Calculating the data dependence edges requires solving a classical data-flow analysis problem called *reaching definitions* [Aho *et al.* 2006]. A *definition* of a variable x is generated by a statement that may assign a new value to x. We say that a definition d of x *reaches* a statement or predicate s if there is a path from the statement generating d to s such that no other definition of x intervenes along that path. If, for a given path from a statement generating a definition d of variable x to s, there is another definition d' of x, we say that the definition d' *kills* the definition d, and in this case d does not reach s along that path (but d may still reach s along another path). Hence, the set of reaching definitions for a given statement or predicate s is the set of definitions that may last have defined one of the variables available to s. In the PDG, we create a data dependence edge from a statement generating a definition d of x to another statement s iff d reaches s and s uses x.

Computing the reaching definitions for a program essentially involves calculating, for each statement and predicate, the set of definitions it generates and the set of definitions it kills, and propagating this information along the control flow graph. As in the dragon book [Aho *et al.* 2006], let gen_s be the set of definitions generated by a statement, and kill_s the set of definitions killed by that statement. For example, if a statement s generates a definition d of a variable x, then $\text{gen}_s = \{d\}$ and kill_s is the set of all other definitions of x in the program. Computing these sets for each statement first requires computing, for each statement in a program, which variables it *defines*. Since we also need to know which variables a statement *uses* so as to create the data dependence edges later on, we begin by running a use/def analysis on the program to determine, for each statement and predicate, which variables it uses and which variables it defines. That is, we run a recursive algorithm that is aware of the semantics of the particular language being analyzed and that is thus able to determine the used and defined variables of all statements and predicates. Using this information, it is straightforward to compute the sets gen_s and kill_s for all statements and predicates s of the program. Then, the reaching definitions problem can be solved using the algorithm by Aho *et al.* [Aho *et al.* 2006] depicted in Figure 3.11. It outputs, for each node v of the control flow graph, the sets $\text{In}(v)$ and $\text{Out}(v)$ containing the reaching definitions immediately before and immediately after node v.

In essence, the algorithm starts by conservatively initializing the sets of reaching definitions for all statements and predicates as empty, then iteratively propagates reaching definitions along the control flow graph until no more changes occur. Ultimately, for any node v, the set $\text{In}(v)$ is the set of reaching

```
// The ENTRY node does not generate any definitions.
Out(ENTRY) = ∅
// Initialize all other Out(v) as empty too.
for each v in V \ {ENTRY}
    Out(v) = ∅
// Iteratively add reaching definitions until a fixpoint is reached.
// preds(v) is the set of predecessors of v in the control flow graph.
while changes to any Out(v) occur
    for each v in V \ {ENTRY}
        In(v) = ( ⋃_{p∈preds(v)} Out(p) )
        Out(v) = gen_v ∪ (In(v) \ kill_v)
```

Figure 3.11: Algorithm for computing reaching definitions [Aho *et al.* 2006].

definitions of v. As can be seen, this algorithm is quite similar to the one for computing dominators presented in Figure 3.10. In fact, both are instances of the classical *data-flow analysis schema* which constitutes an abstraction of this type of algorithm. At its heart, any such data-flow analysis algorithm always propagates some kind of desired information along a control flow graph (either forwards or backwards) until a fixpoint is reached. We saw its instantiations to compute dominators and to compute reaching definitions, but it can also be used, as another example, to perform a *live-variable analysis* (to determine whether a variable could still be used starting from a given point in a control flow graph; a useful information for register allocation) as well as many other types of analyses. We refer the reader to the work by Aho *et al.* for the abstract algorithm and a deeper discussion on the subject.

3.3.3 Program Dependence Graphs

As we stated earlier, program dependence graphs were originally introduced by Ferrante *et al.* [Ferrante *et al.* 1987] in the context of compiler optimization. Their nodes are the same as the nodes of the control flow graph (save for the ENTRY and EXIT nodes) and their edges are the control and data dependence edges discussed in Section 3.3.1 and Section 3.3.2, respectively. Together, these edges represent the *necessary* sequencing of operations, i.e., the necessary dependencies between statements, exposing potential parallelism in a program. That is, the set of all dependencies can be seen as a partial ordering of the statements and predicates in a program. Preserving this ordering also preserves the semantics of the program [Ferrante *et al.* 1987].

Figure 3.12 shows the program dependence graph of our example, where the dash-dotted edges labeled with C represent control dependencies and the dashed

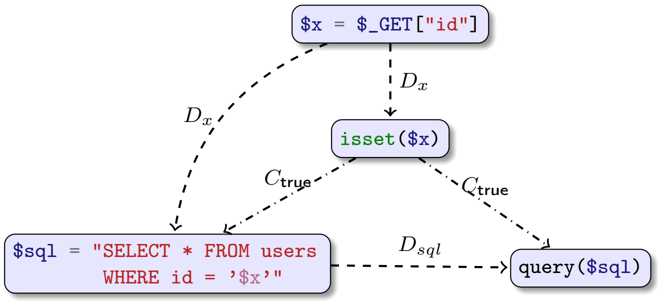

Figure 3.12: Program dependence graph of the running example.

edges labeled with D represent data dependencies. The subscript indicates the value that a predicate must evaluate to for the dependent statement to be evaluated, respectively the variable that induces the data dependency.

Program dependence graphs are also extremely useful to statically analyze the data flow in a program for vulnerability discovery. In particular, the data dependence edges enable us to efficiently analyze the propagation of attacker-controlled data. Consider our running example, which contains a classical SQL injection vulnerability. Starting at the source of the attacker-controlled data (`$_GET["id"]`), we can follow the data dependency forward to the assign statement of the variable `$sql` and from there to the call `query($sql)`, which is a sensitive sink. Reciprocally, we can also start from the sensitive sink and follow the data dependence edges backwards to the source of the attacker-controlled data. Since no sanitization of data is used on this path, this data flow can be automatically determined to be suspicious.

3.4 Call Graphs for Interprocedural Analysis

While dominator trees, control flow and program dependence graphs give us a powerful means to reason about vulnerabilities as discussed in the previous sections, all of these representations are only defined at a function level and, consequently, only allow us to reason intraprocedurally. As we saw in this and the last chapter, these types of representations are often sufficient for vulnerabilities that arise from simple programming mistakes, such as carelessness in syntactical formulation, failure to free resources, various injection vulnerabilities, and so forth. Yet in practice, programs are composed of hundreds or even thousands of functions. Accordingly, we can realistically expect that many vulnerabilities are hidden more subtly in the program code because vulnerable data flows span across multiple function calls. Therefore, it is highly desirable to have a means to follow the propagation of data across function borders.

Call graphs [Ryder 1979] are a type of control flow graph that model calling relationships between the functions of a program. Each node corresponds to a particular function, and a directed edge from function A to function B indicates that A may call B. We can easily transfer this idea to other types of graphs and extend them accordingly, such as control flow or program dependence graphs, by connecting nodes that correspond to call statements to the function definition nodes of the called functions. This is trivial when the function does not take parameters; however, in the presence of parameters, we need to be careful when modeling the arising dependencies between arguments on the caller site and parameters on the callee site. Since the exact details of how we achieve this differ in our analysis of JavaScript in Chapter 4 and our framework for PHP programs in Chapter 5 according to the respective program representations that we use, we defer a detailed discussion of this subject to the respective chapters.

Here, we briefly explain the general idea. Figure 3.13 shows a combined control flow and program dependence graph of our running example. In addition, we added the control flow graph of the called function `query`, which we model as a trivial wrapper function for the PHP built-in function `mysql_query`. The call of the function `query` in function `foo` is connected to the definition of function `foo` with a *call edge*. Generally speaking, such a graph exposing data flow within functions and call relationships between all functions of a program enables an interprocedural analysis: In the example of Figure 3.13, it is now visible that the attacker-controlled source `$_GET["id"]` in function `foo` flows to the sensitive sink `mysql_query` in function `query`. We use similar types of graphs to detect vulnerable interprocedural data flows in the following chapters.

3.5 Discussion

In this chapter, we have seen various types of program representations. Abstract syntax trees allow us to focus on the syntactical structure of a program while abstracting away from its particular details of formulation and idiosyncrasies of the language it is written in. Control flow graphs model the order in which statements and predicates of a program are executed and what conditions lead to a particular path being taken through a program. Dominator and post-dominator trees express which statements or predicates are always executed before or after one another on all possible paths through the control flow graph, while program dependence graphs expose the necessary control and data dependencies between statements and predicates, i.e., they impose a partial ordering on statements and predicates that must be preserved for the semantics

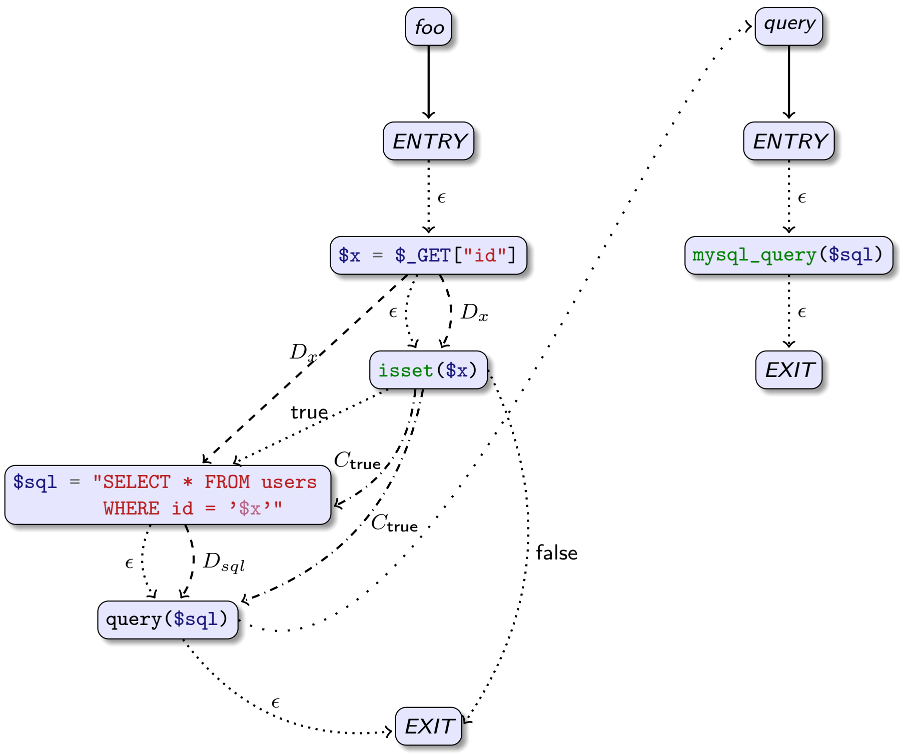

Figure 3.13: Interprocedural control flow, control dependencies and data dependencies of the running example. Dotted arrows indicate control flow, dash-dotted arrows indicate control dependencies and dashed arrows indicate data dependencies. Solid arrows connect function definition nodes to their respective entry nodes, and the loosely dotted arrow represents a call edge.

of a program to remain the same. Program dependence graphs in particular allow us to reason about the flow of sensitive or attacker-controlled data through a program. Finally, call graphs allow us to reason interprocedurally.

As we have seen, all of these structures can be used, in one way or another, to express patterns that correspond to potential vulnerabilities in programs. In Chapters 4 and 5, we will use these structures to implement automatic vulnerability detection for the two currently most widely-deployed languages for web applications: JavaScript and PHP. Chapter 4 will discuss a case study for a popular and particularly security-critical web application written in JavaScript. Chapter 5 presents a framework for automatic detection of vulnerabilities in PHP code and a large-scale study that demonstrates its effectiveness.

Chapter 4

Information Flow Analysis of JavaScript

Contents

JavaScript is, by far and large, the most widespread programming language for client-side web applications [W3Techs 2017a]. Today, virtually every website uses JavaScript, and every major browser supports it natively. It is a high-level, dynamic, and untyped programming language that unites object-oriented, imperative and functional programming paradigms. JavaScript was not designed during years of careful planning. On the contrary, its original prototype was written in only ten days by Brendan Eich at Netscape Communications in 1995, at a time where a number of emerging technologies fought for market domination and timing as well as the availability of a prototype were crucial factors. While its syntax and large parts of its standard library are reminiscent of Java, this is mostly due to the fact that Netscape Communications collaborated with Sun Microsystems at the same time to integrate Java applications into its browser (known as *applets*), and wanted their scripting language to use a similar syntax. In reality, however, JavaScript is more influenced by other programming languages, in particular the prototype-based object-oriented language Self and the functional language Scheme [Ecma 2007]. Even though JavaScript was submitted for standardization to Ecma International in 1996, ultimately resulting in the ECMAScript language specification, different browsers implemented various dialects of JavaScript and in part their JavaScript interpreters behaved differently for more than a decade. Third-party libraries such as jQuery [jQuery 2005] and library plugins attempted to remedy these problems by providing additional APIs for DOM traversal and manipulation on top of native JavaScript and enjoyed widespread adoption. While the situation has recently improved and standardized APIs have been adopted by all major browsers, these libraries are still heavily used.

Overall, an automated security analysis of JavaScript code is a difficult and challenging task. Because of JavaScript's highly dynamic nature, most approaches in the literature tend to opt for dynamic analysis [e.g., Curtsinger *et al.* 2011, Hedin & Sabelfeld 2012, Hedin *et al.* 2016]. While dynamic analysis can be much more precise since it has access to runtime information, it cannot simulate every possible input to trigger every possible behavior and is therefore inherently incomplete. Static analysis of JavaScript has received significantly less attention so far (we discuss related work at the end of this chapter). Yet from a scientific point of view, given the steadily rising number of security-critical client-side web applications, it is an equally worthwhile avenue to investigate. In addition, static analysis has the potential to be integrated into IDEs so as to support developers in avoiding security-critical mistakes even before the release of an application.

Electronic voting. A prime example of a security-critical web application is that of electronic voting. Electronic voting protocols have received tremendous attention by the scientific community in the last few years. Their appeal and their increased acceptance even for real-life elections are fueled by their ability to offer efficient, sound tallying while at the same time providing users the convenience of voting remotely. One of the most widely deployed electronic voting protocols is Helios [Adida 2008, Adida *et al.* 2009]: a state-of-the-art, web-based, open-audit voting system that has seen real-life deployment in a variety of different settings. Among others, it has been used for the election of the university president at the Université de Louvain [Adida *et al.* 2009], for student elections in Louvain [Bulens *et al.* 2011] and Princeton [Adida 2013, Princeton USG 2013], as well as for the election of the IACR committee, the International Association for Cryptologic Research [IACR 2010].

From a security perspective, remote electronic voting protocols such as Helios typically exhibit a highly complex design that uses advanced cryptographic primitives such as homomorphic encryptions, mixnets, and zero-knowledge proofs, that involves many interactions between different parties, and that intends to achieve a wide range of sophisticated security properties. Consequently, securely designing such protocols constitutes a highly challenging and intriguing task. The high complexity of their designs as well as the sophistication of their intended security properties impose significant challenges for rigorously assessing the security of these protocols. A multitude of approaches have been recently proposed to automatically ascertain central security properties for electronic voting, such as vote privacy or vote verifiability. In particular, the security of the Helios protocol has been thoroughly investigated (see related work) and its security rigorously proven for many of its intended security properties. As of now, Helios constitutes one of the most widely examined voting protocols in scientific literature.

However, virtually all existing approaches for confirming the security of voting protocols focus on identifying conceptual (logical) or algorithmic (cryptographic) attacks against the protocol considered, i.e., they consider a protocol's symbolic abstraction or algorithmic description, and are therefore agnostic to security violations that arise in the actually deployed implementation. However, history has shown that even security protocols long deemed and even formally proven secure can exhibit severe implementation-level vulnerabilities: Earlier in this thesis, we already mentioned particularly illustrative examples such as the *Heartbleed* bug in the OpenSSL cryptography library and Apple's *goto fail* bug in its own implementation of SSL/TLS. An implementation-level analysis is thus of the utmost importance for every security protocol that should see widespread real-life deployment while intending to offer strong security guarantees. Electronic voting protocols, with their strong dependency on societal acceptance, clearly cannot afford severe implementation-level vulnerabilities, and thus naturally call for corresponding analyses.

Contributions. From the above, a security analysis of an electronic voting application written in JavaScript appears interesting from two different perspectives: Statically analyzing a real-world JavaScript application on the one hand; and formulating and analyzing desired security properties of electronic voting protocols on an implementation level on the other hand.

In this chapter, we investigate techniques to tackle the highly challenging task of a static analysis of a real-world security-critical JavaScript application by performing an analysis of the expected security properties of the Helios voting client at the *implementation* level. To this end, we provide code transformations and static analysis to track potentially harmful information flows that undermine the confidentiality and integrity properties in a JavaScript implementation. The transformations involve the replacement of specific features, whose presence makes reliable static analysis impossible. By replacing these features with functionally equivalent code, we enable existing static analysis techniques and are able to faithfully model the information flow within a complex JavaScript program by a static dependency graph. By phrasing integrity and privacy properties as an information flow problem, we can use graph slicing to significantly reduce the number of nodes under consideration from roughly 7 million to a handful of potentially threatening flows, of which two can be leveraged into real-world exploits. This approach describes a general means to enable and conduct a security reification through static analysis in real-world JavaScript programs.

We demonstrate the feasibility and usefulness of our approach by reporting on the presence of two vulnerabilities in the JavaScript Helios voting booth client, that, despite years of manual and conceptual analysis, have not yet been revealed:

1. a cross-site scripting (XSS) attack resulting in *arbitrary script execution*; and

2. an undocumented feature, which causes the client to send unencrypted plaintext votes without the prior consent of or notification to the user.

The aforementioned code transformations yield two independent benefits. First, they make the voting client amenable to static analysis. Second, they yield an implicitly hardened version of the Helios voting client that uses fewer external dependencies and thus exhibits a reduced attack surface. This hardened version is publicly and freely available:

`https://github.com/malteskoruppa/heliosbooth`

For the sake of exposition and reproducibility of the individual steps, this version intentionally does not yet integrate the fixes to the vulnerabilities that we report on. We stress that, while our analysis was performed on the transformed code, the vulnerabilities that were found are equally present in the original code. Indeed, we will see escalated exploits performed on the original client according to the insights gained from our analysis.

Outline. The remainder of this chapter is organized as follows. In Section 4.1, we discuss the challenges of analyzing a real-world JavaScript application such as the Helios voting client. In Section 4.2, we briefly review the Helios protocol and take a closer look at the implementation of the voting client. We then describe our approach to overcome these challenges and perform static analysis in Section 4.3. We report on our findings and explain how to escalate them to actual exploits in Section 4.4. Then, we discuss things learned from our approach in Section 4.5. Finally, we present related work in Section 4.6 and summarize this chapter in Section 4.7.

4.1 Challenges in the Analysis of the JavaScript Helios Client

In this section, we present the most notable challenges that we encountered while performing a static analysis of the JavaScript Helios voting client.

4.1.1 Reification of security properties

Existing approaches that verify security properties for Helios focus on the security protocol on a symbolic or algorithmic level. Thus, they are agnostic to security violations that happen on the application layer. *Reification* is the process of verifying that the protocol's security properties are preserved in the actual realization. However, the complexity of the components to realize the Helios protocol in practice grows far beyond the scope of the original protocol analysis. The symbolic primitives are too coarse-grained to capture the minute details of the JavaScript language semantics. Hence, it does not suffice to simply re-hash the symbolic and algorithmic definitions and to apply them to this setting. In addition, clearly not all properties of a protocol can be suitably verified by checking the implementation of only one of the protocol agents (i.e., the client in this case).

Therefore, we must re-interpret security properties related to the client and appropriate for the implementation level in such a way that they suitably portray the original properties' purpose: A breach of confidentiality and the associated loss of privacy is perceived as communication of secret data (e.g., a vote), without proper encryption. Alternatively, by compromising the client system's integrity, an attacker can force the client to behave in an unintended manner. We will later phrase these issues as information flow problems. Note that confidentiality is a property already contained within the protocol's security properties, whereas integrity is an entirely new problem specific to the realization of the protocol.

To analyze the security of the implementation with respect to the mentioned properties, we leverage static analysis techniques to derive semantics-preserving abstractions of the program as described in Chapter 3. These representations can then be used to analyze the (in-)security of the program according to the aforementioned properties. That is, we can automatically search potential breaches of confidentiality or integrity for each possible execution path in the source code, and report suspicious paths. These paths can then be manually inspected and possibly be escalated into full-blown vulnerabilities.

4.1.2 JavaScript is highly dynamic

JavaScript uses higher-order functions and closures, extensive type coercion rules, and a flexible prototype-based object model where objects can be changed at runtime by adding or removing fields and methods. These dynamic and abstract features encumber static analysis. On top of these language features, ECMAScript contains a standard library that contains hundreds of functions and objects that need to be modeled, with new ones being integrated frequently as the language is being continuously developed and improved.

Moreover, as discussed in Section 2.2.3, the function `eval` and its variants allow to dynamically interpret strings as program code. Reasoning about such code requires a-priori knowledge of the strings that can appear and their analysis is therefore not generally amenable to static analysis. Hence, highly dynamic features like `eval` need to be removed or their effects conservatively approximated. One may also assume that *any* attacker-controlled data that flows into an `eval` may constitute a breach of integrity. In practice, however, this may lead to a high number of false positives. Fortunately, the core Helios client does not make use of `eval`. Third-party libraries used by Helios, however, do. We consider the difficulties arising in this context in Section 4.1.4.

In summary, JavaScript's dynamic features present a tremendous challenge to static analysis techniques, which leverage the semantically fixed parts of a program to make guarantees about the program's runtime behavior.

4.1.3 HTML DOM and browser API

JavaScript programs are usually executed in a rich environment. Web applications execute in a browser environment that interacts with the Document

```
<script>
  var src = "foo.png";
</script>
<img src="bar.png" onclick="alert(src)"/>
```

Figure 4.1: HTML DOM structure interfering with the expected variable resolution outcome.

Object Model (DOM) representing the page's HTML, as well as sophisticated libraries such as jQuery [jQuery 2005]. Unfortunately, some of those interfaces, such as the DOM, are not implemented in JavaScript but in C++, which prevents fully automatic analysis. Typical client-side programs are thus specified in a combination of scripting, specification and low-level programming languages. A specification of all of these components is required to obtain a semantically faithful analysis result.

Execution in JavaScript is *event-driven*; hence the analysis must also model the event system, which includes the dynamic registration of event handlers, event bubbling and capturing (recursive triggering of events in nested DOM components) and event-specific object properties. Additionally, all event handlers are callback functions, i.e., they are queued when a specific event triggers. This leads to asynchronous execution, resulting in fragmented code and unstructured execution paths that static analysis must somehow resolve.

The HTML document structure also interferes when resolving variable names defined in HTML attributes. If an event handler is triggered, the scope chain includes *all* DOM objects in the lookup path from the HTML element containing the trigger up to the root of the document. Consequently, JavaScript and HTML are deeply entangled for static analysis. Consider the example in Figure 4.1: The `onclick`-event handler references the variable `src`. However, it is not the variable `src` previously defined in the script tag, but the `src` variable in the `img` tag. Thus, the `onclick` action will trigger an alert containing the string `bar.png` rather than the string `foo.png`. Furthermore, the HTML API features a number of non-trivial and non-obvious interactions. For example, setting the `onclick` property of an HTML element at runtime causes a string to be interpreted as event handler code.

4.1.4 Included libraries

To compound the problems described above, applications are often based on libraries that ease common tasks, such as navigating the DOM or sending and receiving network messages via Ajax, or help developers mitigate incompatibility problems between browsers, as described earlier. Among others, the Helios voting client uses jQuery [jQuery 2005] and Underscore [Ashkenas 2009] to perform a variety of practical tasks in the most compatible way possible. In addition, it used class.js [Resig 2008], which simulates classical object-oriented programming paradigms including class inheritance.

From a static analysis perspective, these libraries—while convenient for a programmer—complicate the analysis process severely. By providing their own abstraction *on top* of very abstract features such as event handling and DOM objects, a high degree of context- and flow-sensitivity is required in order to produce helpful results. jQuery in particular provides the `$` (or `jQuery`) function that has completely different semantics based on its argument, which can be anything ranging from an HTML string to a DOM element.

4.1.5 Problem summary

In conclusion, we face theoretical as well as practical problems in the analysis of a complex real-world JavaScript application like the Helios voting client. On the one hand, we need to model our security concerns (confidentiality and integrity) such that they can be checked by static analysis. On the other hand, we need to be able to execute the static analysis on the JavaScript source code in a sound manner. The analysis itself needs to handle problematic components of Helios, such as frequent requests to jQuery and to the DOM API. To that end, we perform the flow analysis on a refactored, but functionally equivalent[1] version of the Helios client. We then leverage WALA, which is able to represent the client's HTML structure as JavaScript code, such that a thorough static analysis can be performed for the whole program. To satisfy the statically irresolvable dynamic features, we replace them with functionally equivalent functions that do not rely on dynamic input. In the case of Helios, this conversion is fully possible without sacrificing expressiveness or performance, and without altering the functionality of the protocol. In particular, the Helios source code itself does not make use of the `eval` function.

4.2 The Helios Voting System

The Helios voting system [Adida 2008, Adida *et al.* 2009] is a popular web-based, open-audit e-voting system that has been amply studied in the literature, both in symbolic and computational models (see Section 4.6). It is available in well-documented open source form [Adida 2009]. In addition, a public server is running on `http://heliosvoting.org` to allow interested users to test the system and create and run their own election. Since its original publication in 2008 and following experience obtained in practical deployments as well as

[1] We elaborate on the exact nature of these changes in Section 4.3.1.

insights due to the scrutiny of researchers, it has continuously been revised and improved. We focus on the latest version, also known as Helios 3.1.

The server-side code is mostly implemented in Python (using the Django framework), while the client-side code (which runs in the user's browser) is written in HTML and JavaScript. Our analysis focuses exclusively on the client-side code implementing the voting booth. To provide some context, we shortly review the Helios protocol in Section 4.2.1, then take a closer look at the client in Section 4.2.2.

4.2.1 A short review of the Helios protocol

In Helios, any registered user may create a new election. In the initial setup phase, the user who created the election, considered as the *administrator*, can set up the ballot and other election data, and specify a list of eligible voters. A key pair is automatically generated by the Helios server, or a set of trustees, for each new election.

Once the administrator is ready, they can *freeze* the election and move on to the submission phase, in which eligible voters may submit ballots. On a high level, the submission phase, depicted in Figure 4.2, is very simple:

1. A voter requests a specific election from the Helios server.

2. The Helios server sends back the browser voting application, called the *voting booth*, as well as the corresponding election data.

3. The voter uses the voting booth to record their answers and to encrypt them. They may then choose to audit the encryption, or to seal their ballot (discarding randomness and plaintexts) and send the encryption to the server. If the voter chooses to audit the encryption, the voting booth will show them the randomness that was used (enabling them to verify the encryption with an external verification program) and re-encrypt the ballot with a new randomness.

4. Only once the voter chooses to seal and submit their ballot, the voting server requests them to authenticate.

5. The voter authenticates and thereby confirms their wish to cast their ballot. The voting server records their encrypted ballot along with their identity (or an alias) on a public bulletin board.

The ballot is encrypted using Exponential ElGamal, a variant of ElGamal (see [Adida *et al.* 2009, Cortier & Smyth 2011] for details). In the tallying phase, an encrypted tally is computed from all published ballots using homomorphic

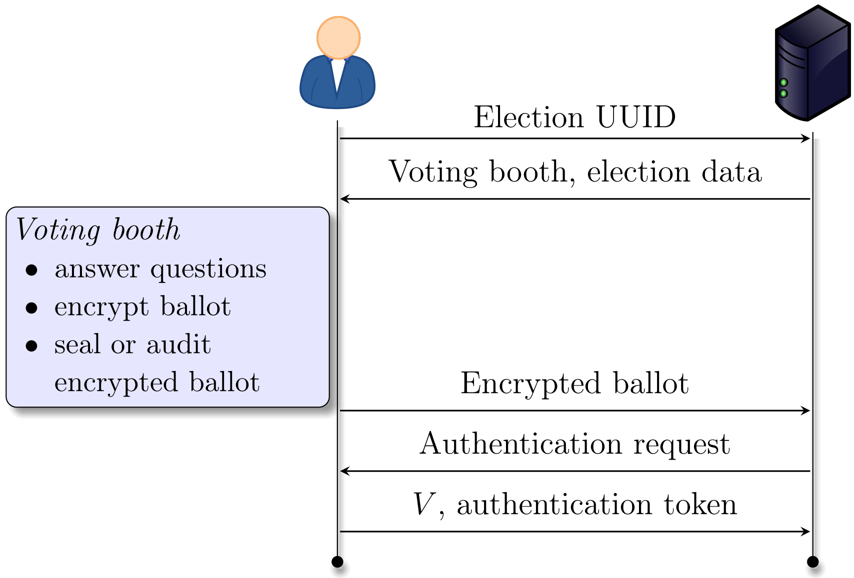

Figure 4.2: High-level overview of the submission phase.

properties of the encryption scheme (see [Cramer *et al.* 1997, Adida *et al.* 2009]) which is then jointly and verifiably decrypted by the trustees. This procedure can be publicly audited by anyone.

In Helios 3.x, authentication in the submission phase is typically achieved via third-party web services such as Google, Facebook or Twitter, and corresponding authentication frameworks such as OAuth [OAuth 2006], although a classical username/password authentication to the Helios server is also configurable. We refer the reader to [Adida *et al.* 2009, Smyth & Pironti 2013] for some interesting discussions and deeper insights into the authentication mechanisms deployed by Helios.

4.2.2 The Helios voting booth

The most complex element during the submission phase—and the one which we focus on in this chapter—is the voting booth. Its behavior is depicted in Figure 4.3:

1. First, the voting booth requests the election data (i.e., the questions and possible answers, etc.) from the Helios server according to the current election id.
2. The voting booth guides the voter through the questions and records their answers.
3. Once the voter is satisfied with their ballot, they may use a *Proceed* button that triggers a JavaScript function to encrypt their ballot and generate non-interactive zero-knowledge proofs of correct encryption.

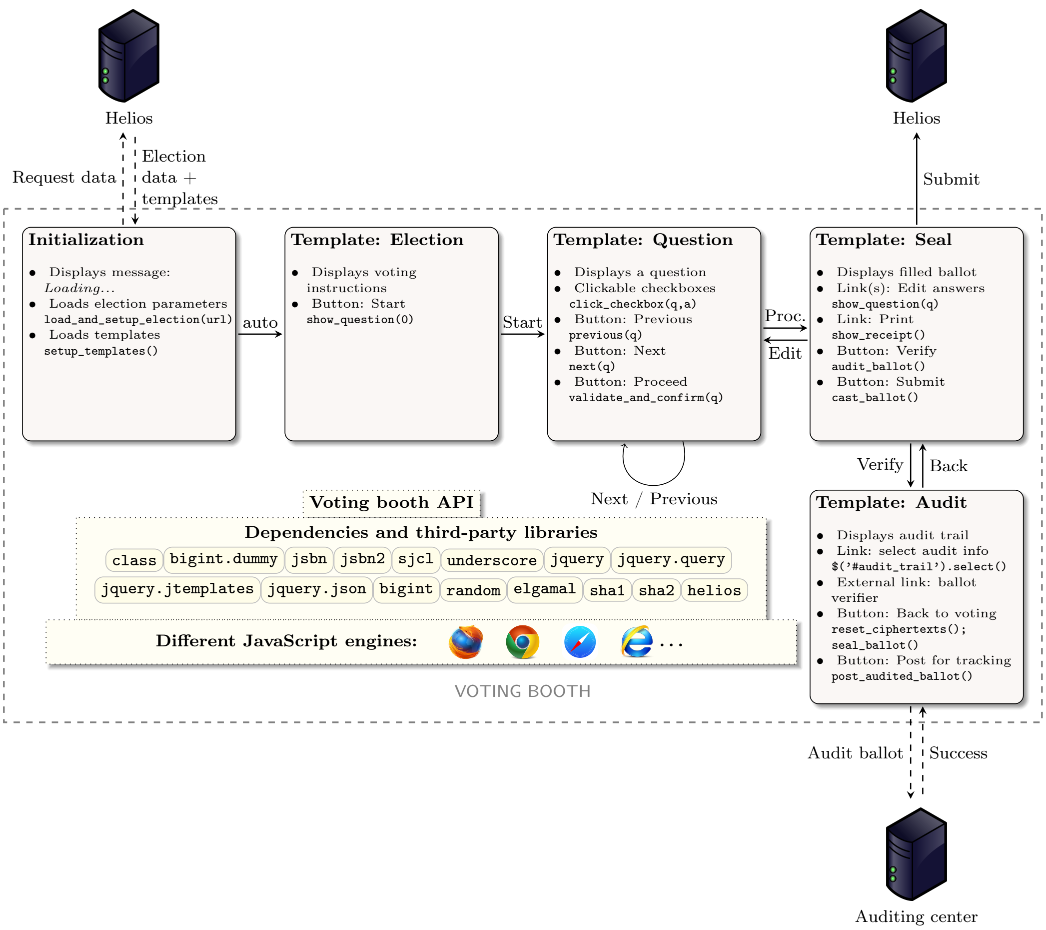

Figure 4.3: User interaction and implementation of the voting booth.

4. A fingerprint of the encrypted ballot is computed. The voter is shown their answers as well as the fingerprint. They may now choose to either *audit* their ballot or *seal* and submit their ballot to the server.
5. If the voter chooses to audit their ballot, the voting booth reveals the entire encrypted ballot along with the plaintext answers and any randomness that was used. The voter can now copy this information and use an external application to re-perform the encryption and verify that the fingerprint displayed earlier matches. Auditing the encrypted ballot will also cause the voting booth to *re-encrypt* the ballot with new randomness. The voter returns to item 4 with the new encrypted ballot.
6. If the voter chooses to *seal* their ballot, all plaintexts and randomness are discarded, and the encrypted ballot is submitted to the voting server.

The idea to use an *auditable* encrypting device in Helios is inspired by Benaloh's Simple Verifiable Voting protocol [Benaloh 2006] with the intent of increasing the voters' confidence in the encrypting device. Since the voting booth does not know at encryption time whether the ballot will be submitted or audited, any cheating attempts will be noticed by a random auditing process with high probability. Consequently, it is also important to separate the *ballot encryption* process and the *ballot casting* process [Benaloh 2007]: The encrypting device should not have any information about who is using it to eliminate targeted cheating. Additionally, the possibility to inspect the voting booth even without being an eligible voter improves auditability.

The possibility to audit the voting booth reflects Helios's concern to guarantee vote *integrity* on every level of the voting process. In all, Helios makes integrity of the vote its prime concern: Voters can audit the voting booth, the correct recording of their encrypted ballot on the bulletin board, and finally the tallying and decryption process. At the same time, Helios also takes great care to ensure vote *privacy*. To this end, the voting booth is written as a *single-page web application*: After initially pre-loading the election data and page templates, the voting booth makes no further network requests until the ballot is sealed and submitted to the voting server. JavaScript functions implement the entire functionality of the voting booth and take care of updating the rendered HTML user interface during the interaction with the voter. On a side note, we add that the reason for the booth to re-encrypt a ballot once it has been audited constitutes a small attempt to thwart coercion and thereby also protect vote privacy: If the voter does not know the randomness of their vote, they cannot prove how they voted. (Of course, such measures are barely any help against coercion, and Helios emphasizes that it is intended for low-coercion elections).

Our aim is to verify that the voting booth fulfills the expected security requirements and that neither its integrity nor its privacy can be compromised. We stress that the threat model here is not a corrupt voter (who could use another application in the first place), but rather a (passive or active) attacker trying to exploit vulnerabilities in the actual voting booth implementation interacting with an honest voter in order to learn, or even surreptitiously modify, a voter's vote.

To this end, we analyze the behavior of the actual voting booth's *implementation* using automated tools in order to discover potential flaws that are too complex to be easily spotted by a manual code review. Unfolding the inner workings of the booth shows that it contains by far the most complex interaction during the entire vote casting process, as can be seen in Figure 4.3, which depicts the voter interaction with the voting booth, what JavaScript functions are called upon various actions, and how the page templates are updated. When a voter requests the voting booth for a particular election from the Helios server (see Figure 4.2), the voting booth application is first sent to the voter and parsed by the voter's browser. Upon initializing, the voting booth then requests the election data and the HTML templates to be displayed during the different phases of the ballot preparation process in the background. When this data has finished loading, the internal JavaScript scope is updated and the voting booth displays the first question. The voter is now guided through the ballot preparation process, as explained earlier. The only network requests that are made are either to post an audited ballot to the auditing center, or to submit an encrypted ballot. As it will be of interest later, we note that Helios implements an independent ballot verification program also written in JavaScript to realize the auditing center. The link to it is dynamically generated (in particular, it includes a GET parameter that specifies the election id) when a user clicks on the button to audit their ballot. Additionally, the analysis of this code is complicated by the fact that it depends on a multitude of complex dependencies and third-party libraries, such as jQuery, which pose a significant challenge as we discussed in Section 4.1.

In the next section, we discuss how we tackle the analysis of such a complex JavaScript application.

4.3 Implementation-level Analysis

As mentioned previously, we focus on the client-side code implementing the voting booth for our vulnerability analysis. The phases of our analysis are outlined in Figure 4.4 and demonstrate our approach to tackle the challenges discussed in Section 4.1:

1. First, we present code transformations that resolve the majority of the problems caused by included libraries (see Section 4.1.4).

2. We then process the results of our transformations with WALA to provide a unified program representation and a number of analyses tackling problems caused by the JavaScript and HTML components (see Section 4.1.2 and Section 4.1.3).

3. Finally, we formulate the reification process (see Section 4.1.1) as a graph slicing and information flow problem and apply it to the slices. Our findings as well as the applicability of our findings to the original code will be discussed in Section 4.4.

WALA (the IBM T.J. *WA*tson *L*ibraries for *A*nalysis) [IBM 2006] is a Java library originally designed to provide static analysis capabilities for Java bytecode. Released under an open source license in 2006, it has since been used in several research projects as well as further analysis tools, such as JOANA [Hammer & Snelting 2009] or Andromeda [Tripp *et al.* 2013].

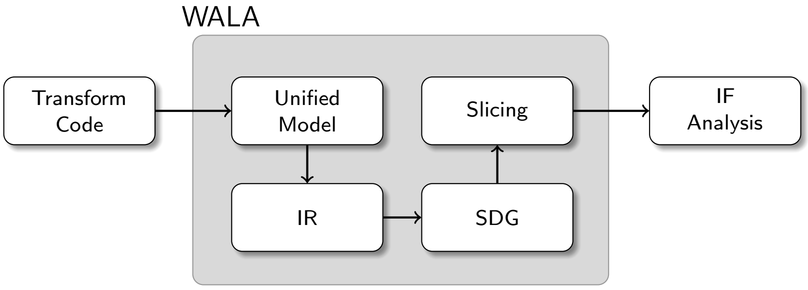

Figure 4.4: Overview of our approach.

Most of the WALA API internally leverages the WALA IR (intermediate representation) instead of source code, which is represented in SSA (static single assignment) form [Cytron *et al.* 1989]. The intermediate representation implemented by WALA is general enough to represent and thus to analyze other languages as well. To demonstrate the applicability of WALA to other languages, the authors implemented a front end for JavaScript code using the Rhino parser [Mozilla 1998]. Among other analyses, WALA supports analysis of class hierarchies and type systems (more relevant to Java), mature call graph construction and pointer analysis (for JavaScript, using variants of Andersen's analysis [Andersen 1994]), interprocedural data flow analysis using an RHS solver [Reps *et al.* 1995] (with extensions, e.g., to handle exceptions), and context-sensitive tabulation-based program slicing [Weiser 1979].

WALA is well-suited for our analysis because it features a unified model for programs consisting of HTML and JavaScript; it uses a well-structured intermediate representation for static analysis and supports a wide range of different approaches to static analysis, most notably program slicing using system dependence graphs, which makes all possible information flows explicit.

In the next sections, we elaborate on the individual steps of our approach as shown in Figure 4.4.

4.3.1 Code transformations

The implementation of the Helios voting booth heavily relies on the third-party libraries jQuery [jQuery 2005], Underscore [Ashkenas 2009] and an implementation of class inheritance for JavaScript [Resig 2008]. Due to the highly reflective nature of these libraries, it is extremely hard to perform automated static analysis on the Helios voting booth's code. Their sheer size is another problem: The uncompressed version of jQuery 1.2.2 (as used by Helios) amounts to 100 kilobytes (its compressed version 60 kilobytes), as compared to about 50 kilobytes for the Helios voting booth itself (excluding smaller dependencies).

While jQuery and other libraries make developing web applications easier, they typically prevent automated static analysis, as current tools, including WALA, can only cope with some of the dynamic features that are present in these libraries, and even for these only in a very limited way (this is subject to active research as discussed in Section 4.6). To enable static analysis, we hence refactor the Helios implementation so as to use native JavaScript equivalents. These code transformations yield a client that is independent of the aforementioned libraries and potential security bugs induced by the libraries. The changes are canonical and could even be refactored automatically. The modified code is functionally identical to the original code, i.e., the voting booth works in exactly the same way.

In this section, we briefly describe these code transformations, so that the interested reader may ensure that they soundly model the behavior of the original code. The complete list of transformations is described in Appendix A. In addition, the modified code is publicly available:

`https://github.com/malteskoruppa/heliosbooth`

We organize this section according to the different libraries whose functionality we emulate. We begin with core jQuery functionality and jQuery plugins, then discuss the simpler Underscore library, and finally look into JavaScript class inheritance.

4.3.1.1 jQuery

The jQuery library for JavaScript provides facilities for accessing and updating the DOM, handling events or writing Ajax applications, in a convenient and portable manner. One of its key benefits is that it avoids the need for developers to deal with JavaScript DOM API idiosyncrasies across browsers, and allows them to write concise and legible code. However, newer standards for the browser, like the DOM API, CSS and HTML5, provide equivalent functionality for most of jQuery's APIs, which yields a straightforward refactoring. Examples for accessing DOM nodes by ID or class and other minor refactorings can be found in Appendix A.1.

Among the core jQuery functions used by Helios are those for performing asynchronous HTTP requests. Namely, Helios uses the `.get()`, `.getJSON()` and `.post()` methods (all of which are wrapper functions for jQuery's `.ajax()` method that sets up a JavaScript `XMLHttpRequest` object). These functions are particularly interesting for our analysis, since they constitute information sinks that may potentially lead to confidential information being sent over the network, as discussed later. The JavaScript code in Figure 4.5 uses jQuery to perform an asynchronous HTTP request to the URL `url`, and if the

```
$.get( url,
  function( response) {
    /* process answer */
  });
```

Figure 4.5: Using jQuery to perform an asynchronous HTTP request.

```
var request = new XMLHttpRequest();
request.open( "GET", url, true);
request.onload =
  function() {
    if( request.status >= 200 && request.status < 400) {
      var response = request.responseText;
      /* process answer */
    }
  };
request.send();
```

Figure 4.6: Using `XMLHttpRequest` to perform an asynchronous HTTP request.

server successfully sends an answer, calls the given success handler function to process it. The same functionality contains somewhat more boilerplate in pure JavaScript using `XMLHttpRequest`, as shown in Figure 4.6.

The code for jQuery's `.getJSON()` and `.post()` functions is analogous. The main differences are that the former additionally parses the response data as a JSON string and hands the resulting JavaScript object to the success handler function, while the latter performs a POST instead of a GET request. The POST method is used to submit data to the server, and jQuery encodes the submitted data as a URL query string using an internal function `.param()`. This encoding is expected by the Helios server, so we also model it for our refactorings. The corresponding transformations are detailed in Appendix A.2.

4.3.1.2 jQuery plugins

Several plugins extend jQuery's core functionality, of which Helios uses:

1. the jQuery JSON plugin [Harris 2008];

2. the query object [Mitchelmore 2009]; and

3. the jTemplates template engine [Gloc 2007].

The jQuery JSON plugin provides (de-)serialization functionality, as shown in Figure 4.7. The deserialization function `.secureEvalJSON()` calls JavaScript's `eval` to convert a JSON string to a JavaScript object, but attempts to prevent malicious script injection by filtering the given string first. However,

```
// serialization
$.toJSON( obj);
// deserialization
$.secureEvalJSON( jsonString);
```

Figure 4.7: (De-)serialization using jQuery's JSON plugin.

```
// serialization
JSON.stringify( obj);
// deserialization
JSON.parse( jsonString);
```

Figure 4.8: (De-)serialization using JavaScript's native JSON object.

this is obsolete as this plugin only simulates the native functionality of the JSON object implemented in modern browsers, as shown in Figure 4.8.

Helios also uses the query object plugin, with functions for reading and manipulating the URL query string. It is only used by Helios to read query string GET parameters. There is no native JavaScript equivalent, but it is easy to write one using a regular expression, as we show in Appendix A.3.

Lastly, Helios uses a template engine written as a jQuery plugin. Templates are small bits of HTML intermixed with logic written in a template language that allows dynamically generating HTML content. For instance, consider the following page element, displayed while the Helios voting booth is being loaded:

```
<div id="header">
Loading election booth...
</div>
```

This element displays a default loading message. Once the voting booth has finished loading all the data it needs to operate offline, this element should be replaced with some other text. To this end, first Helios binds the page element to a template:

```
$( "#header").setTemplateURL( "header.html");
```

This call will load the resource `header.html` in the background; no further network requests will be needed. Note that the contents of this resource do not replace the previous contents of the page element just yet. The resource is simply loaded and internally bound to the page element `#header`. The document `header.html` is an HTML template that may contain placeholders, and even bits of logic, using a templating language defined by the template plugin itself. This obviously hampers static analysis considerably, since we now have to approximate the effects of another *custom* language defined by a specific plugin. As a simple example, the following code is contained in the document `header.html`:

```
<h1 id="election_name">{$T.election.name}</h1>
<p>Voting booth</p>
```

Once the election booth with all the required data has been loaded, the JavaScript code can render the template into the corresponding page element that it previously bound the template to:

```
$( "#header").processTemplate(
  { "election" : election_object}
);
```

The template engine will preprocess the template and replace any placeholders with the given data. Additionally, templates may contain small bits of logic that use this data, such as `{#if ...}...{#/if}` to render certain bits of HTML only under certain conditions, or `{#foreach ... as ...}...{#/for}` to render a certain bit of HTML several times (e.g., to generate a checkbox for each possible answer to a question). Finally, the original content of the `#header` element is replaced with the processed template.

For static analysis, this functionality was one of the greater challenges. Clearly, in static analysis we cannot consider network requests that load portions of a page dynamically. But even if we store the original template together with the main code, this does not resolve all the problems. Indeed, this content contains bits of logic in a custom templating language that is dynamically evaluated at runtime, making it almost impossible to assess the effects of this function call to the HTML DOM statically.

The original Helios paper [Adida 2008] emphasized the usage of templates as a great feature to avoid the otherwise tempting intermixing of HTML and JavaScript. However, intermixing HTML and JavaScript is exactly what this plugin does behind the scenes, so while its use eases a manual code review (if one trusts the template engine), it is problematic for an automated static analysis. To overcome this problem, we re-implemented the templates in pure JavaScript, avoiding intermixing any HTML. This was the most complex code transformation, but turned out to be best for static analysis. In the above example, to emulate the behavior of the header template, we would replace the `#header` page element in the page with the code shown in Figure 4.9. This places both the original content of the header element and the (unprocessed) template statically into the correct portion of the HTML DOM. Thus, we do not require the call to `.setTemplateURL()` any longer. Further, we replace the call to `.processTemplate()` with a call to the custom function shown in Figure 4.10, which simulates the template processing of the original template using pure JavaScript.

```html
<div id="header">
  <div id="header_unprocessed">
    Loading election booth...
  </div>
  <div id="header_processed" style="display: none;">
    <h1 id="election_name"></h1>
    <p>Voting booth</p>
  </div>
</div>
```

Figure 4.9: Page element to simulate the effects of the template plugin.

```javascript
function processTemplate_header() {
  // process template
  document.getElementById( "election_name").textContent = election_object.name;
  // make it visible
  document.getElementById( "header_unprocessed").style.display = "none";
  document.getElementById( "header_processed").style.display = "";
}
```

Figure 4.10: JavaScript function to simulate the effects of the template plugin.

The functions for other templates, such as the `question` template that displays a question, are similar, but quickly grow more complex as the logic contained in those plugins becomes more involved and more DOM manipulations have to be made with JavaScript. We detail them in Appendix A.4.

4.3.1.3 Other libraries

Helios leverages two more libraries to ease development. However, all of the features used can easily be replaced by equivalent JavaScript code. The refactorings elaborated in Appendix A.5 rid our static analysis of the highly complex Underscore library, which offers a multitude of functions in about 30 kilobytes. The other library, which allows class-style inheritance instead of JavaScript's prototype-based inheritance, is only used as syntactic sugar to define objects. Helios does *not* leverage inheritance at all, so refactoring was not difficult, and we detail it in Appendix A.6.

4.3.2 A unified model for HTML and JavaScript components

As noted previously, the intermixing of JavaScript and HTML is commonplace, but unduly hinders static analysis. In order to faithfully process and analyze all aspects of such programs, WALA integrates the HTML components into a unified JavaScript model. Intuitively, the DOM is represented as nested

```
<html>
<body>
<script>
function foo() {
  alert( "Hello World!");
}
</script>
<a onclick="foo()">Click me</a>
</body>
</html>
```

Figure 4.11: Simple JavaScript program with intermixed HTML.

functions, which can be referenced from anywhere within the program using function calls.

As an example, Figure 4.11 shows a simple JavaScript program with intermixed HTML and Figure 4.12 depicts the resulting pure JavaScript analysis model. The model consists of three parts, which are encapsulated by the window.__MAIN__ function. First, top-level JavaScript code (i.e., code not contained in functions, but on the uppermost layer) is added. In the illustrative example, this is only the definition and body of the function foo, but not the JavaScript code contained within the <a> node. As a second step, WALA rebuilds the DOM structure as a JavaScript model: The DOM tree structure is modeled by nested functions. The function make_node0 represents the outermost <html> element, containing the <body> node in form of the function make_node1. Within this function are both the <script> and <a> nodes represented as make_node2 and make_node3, respectively. However, only make_node3 contains the code triggered by onclick (as in the original HTML tag) in a function this.onclick. The code within the <script> tag, as mentioned previously, has been moved to the top-level node. The third and final component of the model is user interaction. WALA represents this as an infinite loop that continuously simulates all possible user interactions. In our example, these interactions are loading the site and clicking the button of node three.

This representation models an entire page consisting of intermixed JavaScript and HTML as a single large JavaScript program, thereby allowing us to circumvent the problems discussed in Section 4.1.3. In addition, data flows that may occur when a user clicks a certain sequence of buttons, thereby triggering associated JavaScript functions, are also modeled.

```
window.__MAIN__ = function __WINDOW_MAIN__() {
  // top-level JavaScript
  function foo() {
   alert( "Hello World!");
  }
  // build the DOM
  function make_node0( parent) {
    // construct <html> element
    // using JavaScript DOM methods
    function make_node1( parent) {
      // construct <body> node
      function make_node2( parent) {
        // construct <script> node
      };
      function make_node3( parent) {
        // construct <a> node
        this.onclick = function a_onclick( ev) { foo() };
      };
    };
  };
  // model user interaction
  while( true){
    window.onload();
    node3.onclick();
  }
}
window.__MAIN__();
```

Figure 4.12: JavaScript model resulting from the code in Figure 4.11.

4.3.3 Intermediate representation

After conversion into a pure JavaScript program, WALA uses Rhino [Mozilla 1998] to parse the JavaScript program, creating an *intermediate representation* (IR). The IR represents a method's instructions in a Java bytecode-like, static single assignment (SSA) form that eliminates stack abstraction and instead maps variables to symbolic registers. As is typical in compilers, the IR organizes instructions in a *control-flow graph* (see Section 3.2.1).

Figure 4.13 contains a running JavaScript example for this section and following sections, and Figure 4.14 illustrates the conversion of the functions `foo` and `iszero` to the WALA IR.[2] First, as can be seen in Figure 4.14, variable assignments and function calls are broken up into individual statements. Intermediate results, e.g., return values from calls or arguments, are stored in symbolic registers named `v<number>`. The call to `iszero` is realized by the `invoke` command, with `a` as a parameter. Note that WALA distinguishes between two types of function calls for technical reasons: `dispatch` is primarily used to handle *method calls*, i.e., calls to functions directly associated with an

[2]For the sake of exposition, the IR omits some details that are necessary to resolve the internal variables and the function names to the correct object, but is otherwise faithful.

```
var foo =
  function() {
    a = 3;
    b = iszero( a);
  };

var iszero =
  function( z) {
    return z == 0;
  };
```

Figure 4.13: Code of the running example.

```
a = 3;
v3 = invoke iszero a;
b = v3;
```

```
v3 = binaryop(eq) v1, 0;
ret v3;
```

Figure 4.14: WALA IR of the running example.

object, such as `B.bar()`, whereas `invoke` resolves non-method calls such as `bar()`. The function `iszero` is not associated with an object, consequently `invoke` is used.

We can set up various structures and maps from IR constructs to information that is relevant to specific analysis forms. Our main interest here lies in the so-called *system dependence graph*, which can be analyzed for illegitimate data processing using *information flow control* theory. We discuss system dependence graphs in the next section.

4.3.4 System dependence graphs

As a next step, WALA converts the IR to another program representation more suited to information flow analysis: the *system dependence graph* (SDG). SDGs are used to conservatively approximate all possible information flow within a program. As was true for the program dependence graph discussed in Section 3.3.3, a system dependence graph of a program P is a directed graph where the nodes represent P's statements and predicates, and the edges represent the dependencies between them [Horwitz *et al.* 1990]. In fact, the system dependence graph can be seen as an extension of the program dependence graph allowing for interprocedural analysis: It is partitioned into program dependence graphs that model the control and data dependencies within single functions and procedures of the complete program. The PDGs are connected at *call sites*, consisting of a call node c (i.e., a node containing an `invoke` or `dispatch` statement) that is connected with the entry node e of the called function. Parameter passing and result returning, as well as side effects of the called function, are modeled via formal *parameter* and *return* nodes and edges. For passed parameters, there exists an appropriate formal node at

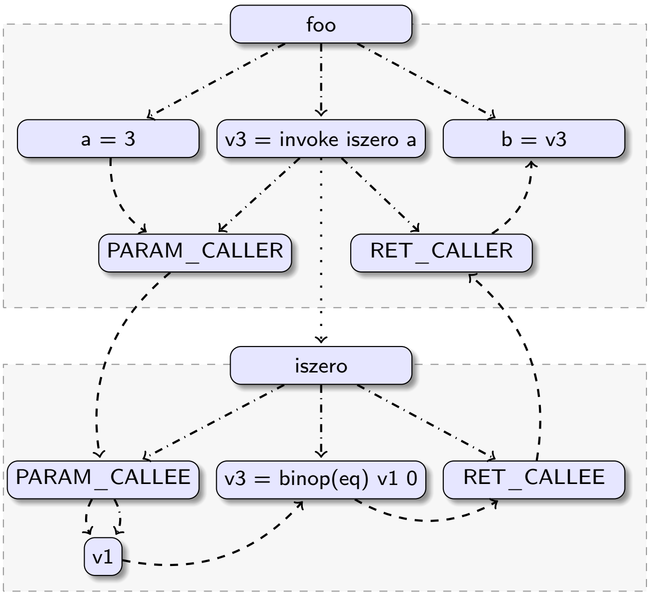

Figure 4.15: SDG of the two functions of the running example.

caller and callee sites, called `PARAM_CALLER` and `PARAM_CALLEE`, respectively. Likewise, there exist `RETURN_CALLER` and `RETURN_CALLEE` nodes for return value passing. The `PARAM_CALLER` nodes (referred to as formal-*out* nodes) are control dependent on the calling statement c, whereas the `PARAM_CALLEE` nodes (called formal-*in* nodes) are control dependent on the function entry node e. Likewise for the return nodes. This parameter passing model guarantees that all inter-procedural effects of a function are propagated via call sites. A machine-checked proof [Wasserrab & Lohner 2010] shows that the SDG is a conservative approximation to the real data and control flows in a program, i.e., it contains all actual flows.

Figure 4.15 shows the SDG of our running example. The formal-in and formal-out nodes are constructed as described above. Consistently with Chapter 3, dash-dotted arrows depict control dependencies, dashed arrows represent data dependencies, and loosely dotted arrows represent function calls.

4.3.5 Slicing

Slicing [Reps *et al.* 1994] is used to find all nodes that can be reached in the SDG from a specific *seed* node. The most important benefit of this method is to restrict the size of the graph to be analyzed as fast as possible.

Assume that in the example in Figure 4.15, we are interested in which statements can influence the value that is passed as a parameter to the function `iszero` at a specific call site. We therefore compute a backwards slice,

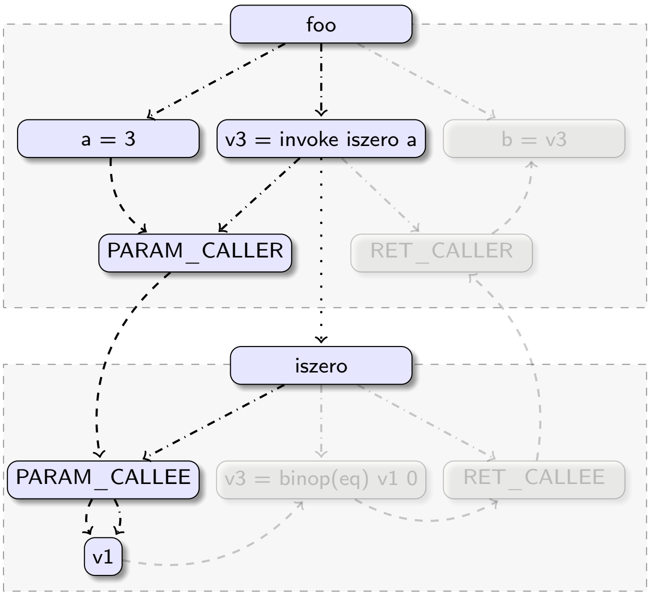

Figure 4.16: Backwards slice of `v1` of the running example.

shown in Figure 4.16. The slice contains only nodes that can be reached by traversing dependencies backwards, be they control, call or data dependencies: In Figure 4.16, nodes contained in the slice are colored blue, while nodes not contained in the slice are transparent. Beginning from `v1`, the data dependency can be followed backwards to `PARAM_CALLEE`, from where it passes out of `iszero` back into the calling function to the `PARAM_CALLER` node. Subsequently, we reach the node `a = 3`. By also including the control dependencies (indirect flow) we can also include `iszero` (from `PARAM_CALLEE`), `v3 = invoke iszero a` (from `iszero` and `PARAM_CALLER`), and `foo`.

4.3.6 Confidentiality and integrity analysis using information flow

Before we present the actual analysis for integrity and confidentiality on graph slices, we briefly summarize standard information-flow terminology. Conventionally, information flow analysis distinguishes between *explicit* and *implicit* flows. Explicit flows correspond to directly copying information, e.g., via a variable assignment such as `l = h;`, in which the value of a *secret* (or *high*) variable `h` is passed to a *public* (or *low*) variable `l`. Implicit flows appear when the *control flow* of the program, i.e., the sequence of statements that are executed, is dependent on high variables. Consider, for example, the program `if (h) l=1; else l=0;`, wherein the content of the low variable `l` will be set to either `1` or `0`, directly corresponding to the value of `h`.

Numerically, the full SDG of the Helios client consists of roughly 7 million nodes and is therefore impossible to analyze manually. By phrasing the security issues in terms of an information flow problem, we can compute appropriate slices, which only contain at most 6000 nodes. This number is further reduced since we only need to consider paths between a high and a low statement, which leaves us with only a handful of different paths containing less than 40 nodes.

4.3.6.1 Confidentiality

In terms of information flow, we can state our confidentiality problem as a *declassification* problem: Sending high, confidential data over a low, public channel without *declassifying* first by means of encryption constitutes a compromise in confidentiality (see Section 3.5). In the SDG, we observe a breach of confidentiality as a path from a high input source (e.g., a secret vote) to a low output (e.g., an `XMLHttpRequest`) without a declassification mechanism that declassifies data in-between.

Figure 4.17 shows parts of a slice resulting from slicing backwards from an `XMLHttpRequest.send()` function call in the SDG of our transformed Helios voting booth. The node `dispatch send v50` represents a call to the method `send` with the variable stored in `v50` as an argument. `v50` is computed from the previous statement `invoke v52 v4`. Following the data dependencies backwards, one eventually crosses from the callee `ajax_post` into the calling function `request_ballot_encryption`. In this function, one eventually reaches the statement `v46 = getfield answers` while following the data dependencies backwards. This statement retrieves the highly confidential votes. Since there is no declassification contained in this data dependency path, we have to consider this execution path as potentially dangerous. In Figure 4.17, the blue dashed arrows highlight the suspicious path along the data dependence edges. As in Figure 4.15, dash-dotted arrows depict control dependencies and loosely dotted arrows represent function calls.

Note that we left the definition of declassification ambiguous: There is no automatic decision procedure to decide whether a function is an *appropriate* means of declassification. Therefore we manually analyze the resulting paths for problematic flows whenever declassification is absent.

4.3.6.2 Integrity

Public, low input is passed to the JavaScript Helios voting client using GET parameters, whose values are specified in the URL. Usually, these parameters have to be *endorsed* (e.g., via a sanitization function) and handled with

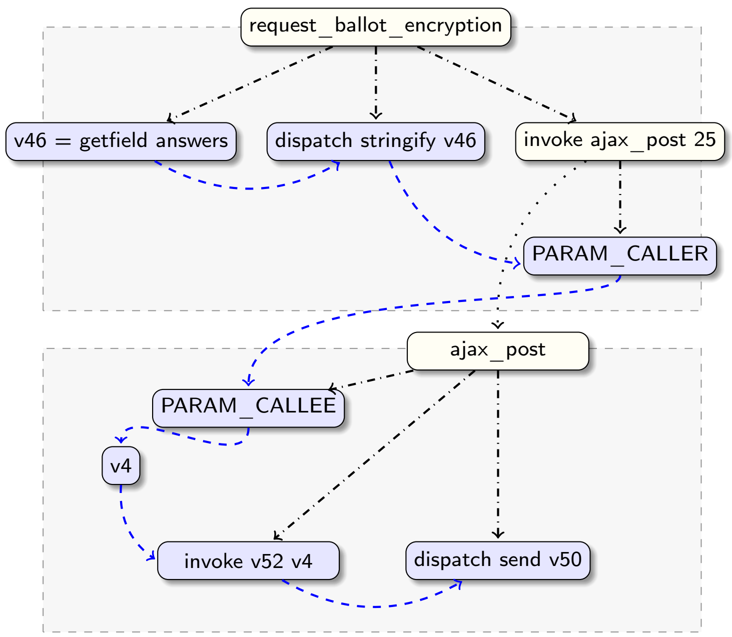

Figure 4.17: Relevant parts of a backward slice that highlights a suspicious path with `XMLHttpRequest.send()` as the seed.

great care. A breach of integrity can be observed when sanitization is absent (see Section 3.5). In terms of information flow we can phrase this problem as a forward flow from calls that retrieve GET parameter values, eventually leading to a high variable without passing through an endorsement function first.

As was the case with declassification, there is no automatic way to ascertain that a function is an *appropriate* means of endorsement. Therefore we, again, require human insight to confirm whether the functions called on a path are sufficient.

4.4 Vulnerabilities

Using the methodology described in the previous section, our automatic analysis was able to identify two flaws in the Helios client-side source code: one breach of integrity, and one breach of confidentiality. We verified that the corresponding information flows can be exploited in practice in the live version of Helios by successfully deploying corresponding exploits in a mock election, both for our transformed version of the voting booth and the original one. The breach of confidentiality results in a browser-independent vulnerability leading to arbitrary script execution, whereas the breach of confidentiality is only evident in a subset of browsers. We discuss the vulnerability originating from the breach of integrity first.

4.4.1 Arbitrary script execution

The first security flaw we discovered is a cross-site scripting attack. We notified the authors of Helios of this vulnerability and they acknowledged that it is a severe problem that they intend to fix; at the time of submission of this thesis, this flaw is still present. As we discussed in Chapter 2, cross-site scripting is considered one of the most critical and most prevalent security vulnerabilities in web applications. It occurs when poorly validated user input is executed by the browser's interpreter. In this case, we are looking at a DOM-based cross-site scripting exploit, that is, the browser itself is caused to insert a script into the DOM that it then executes.

4.4.1.1 Base XSS exploit

In this case, the DOM-based XSS vulnerability arises from a specially crafted GET parameter. Indeed, the Helios voting booth is generally loaded via a URL such as:

```
http://heliosvoting.org/booth/vote.html
          ?election_url=/helios/elections/<UUID>
```

where `<UUID>` is the election-specific identifier which has the form of a hash. This URL contains a GET parameter `election_url` accessible from the client-side JavaScript code.[3]

The client parses this parameter `election_url` in order to load election data, election metadata, and additional entropy from the server, as can be seen in the code snippet extracted from the original Helios client in Figure 4.18. With a properly formatted parameter `election_url`, the URLs used in these requests invoke the Helios server API to return JSON objects containing data to initialize the voting booth. Unfortunately, the parameter `election_url` is not sanitized on the client side. Therefore, an attacker can use it to point it towards an external resource, e.g., an attacker-controlled server, using a URL such as:

```
http://heliosvoting.org/booth/vote.html
          ?election_url=http://attacker.evil/get-corrupt-data
```

In addition, the attacker sets up a server `http://attacker.evil` to return JSON strings of the format expected by the Helios client, but with corrupted contents, allowing the attacker to inject their own election data.

[3]The parameter is usually URL-encoded, but we present it here in URL-decoded form for the sake of readability.

```
var election_url = $.query.get("election_url");
// ...
$.get(election_url,
  function(resp) {
    /* set up election data */
  });
$.getJSON(election_url + "/meta",
  function(resp) {
    /* set up election metadata */
  });
$.get(election_url + "/get-rand",
  function(resp) {
    /* add server randomness to entropy */
  });
```

Figure 4.18: XSS vulnerability in the original Helios voting client.

4.4.1.2 Circumventing the same-origin policy

The DOM-based XSS vulnerability described above will not in fact work when done naively, as the browser's *same-origin policy* prevents accessing external resources. In practice, the requests will be sent and the handler functions called, but the response variable `resp` will have the value `undefined`. However, this can be circumvented. Indeed, nowadays web applications are so complex and often composed of multiple scripts that they may intentionally want to dynamically include scripts from other locations. This can be achieved using the *cross-origin resource sharing* (CORS) mechanism. Using this mechanism, web servers may allow requests from other domains to access (some of) their resources. In the case of the attack presented in this section, it is actually the malicious content that wants to be accessed. Hence, the attacker can abuse the CORS mechanism to inject their script into the voting booth as follows. The attacker needs to set up their malicious server to allow requests from other origins to access the corrupted JSON data by sending a specially crafted HTTP header along with this data. For illustration, Figure 4.19 shows such a header allowing the resource sent in the HTTP body (i.e., the corrupted JSON data) to be accessed by scripts from any origin (for this particular attack, allowing the resource to be accessed only from the domain where the Helios voting booth resides would equally work). The corrupted JSON data will then be successfully passed to and processed by the corresponding handler functions in the JavaScript Helios voting client.

Using this approach, the attacker can manipulate the election data contained in returned JSON strings, with severe consequences: The attacker can compromise the integrity of the vote by intentionally mislabeling the answers (e.g., switching the displayed order of names) in a vote, deceiving the user into voting for the wrong candidate. Likewise, the attacker can violate vote privacy

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Access-Control-Allow-Headers: X-Requested-With,
  ↪   Origin, Content-Type, Accept
Content-Type: application/json
```

Figure 4.19: CORS header to circumvent the same-origin policy for our exploit.

by substituting their own encryption key for the authentic encryption key used to encrypt the final ballot submitted over the network.

Notably, the attack even compromises the ballot auditing process described in Section 4.2.2. This is due to the fact that the link generated by the voting booth contains the same GET parameter as the URL of the voting booth, and the ballot auditing code contains the same code snippet to load election data as the voting booth itself. In other words, the ballot auditing code contains the very same vulnerability and is equally duped by the corrupted JSON data.

Still, while the attacker can alter JSON object-specific values to their liking, they are still unable to execute arbitrary code, i.e., to completely control the voting booth.

4.4.1.3 Arbitrary script execution

It turns out that the attack can be escalated even further in the original Helios voting client (but not in our transformed client) by setting up an external server that sends JavaScript code instead of the expected JSON object. This leads to the script being loaded and executed by the Helios voting booth client.

This behavior is a consequence of how jQuery evaluates the `$.getJSON` function when it is called with an external URL as its first argument: It tries to circumvent the same-origin policy by itself. Instead of issuing a regular `XMLHttpRequest` for the resource (as our modified code does, see Section 4.3.1), jQuery creates a `<script>` tag inside the DOM's header and sets its `src` attribute to the remote URL. Since remote scripts included in this manner are intentionally exempt from the same-origin policy, this causes the browser to load and execute the retrieved content regardless of its origin. Normally, `$.getJSON` would expect to retrieve a JSON object, which does not hold executable content. By sending a JavaScript instead of a JSON object from the remote server, the attacker gains the ability to execute arbitrary code. Thus, the attacker gains complete control over the client and may, for example, hijack a voter's session, or log a voter's every activity in the voting booth, and so forth.

```
$.post(BOOTH.election_url + "/encrypt-ballot",
  { 'answers': $.toJSON(BOOTH.ballot.answers) },
  function(result) {
    /* process encrypted ballot */
  });
```

Figure 4.20: Sensitive data exposure in the original Helios voting client.

4.4.2 Leaking the vote

The second flaw uncovered by our analysis is a program path that leads to an *unencrypted ballot* being openly sent over the network. Specifically, when the voting booth's DOM is loaded, the Helios client promptly confirms whether the client supports *web workers* (scripts running in background threads). Web workers are used in Helios to perform encryption of a voter's choice and generation of zero-knowledge proofs.[4] Surprisingly, if the browser does not support web workers, the Helios voting client simply requests the server to encrypt the ballot, and sends it the plaintext ballot, as can be seen in the code snippet presented in Figure 4.20.

Clearly, this code was included on purpose, yet it comes as a complete surprise, as the voting booth does perform a network request while interacting with the voter, contradicting the premise of a single-page web application and the claim of the original paper [Adida 2008]. Sending the plaintext ballot violates all assumptions. It could be argued that when using a secured HTTPS connection, a passive attacker cannot read the secret ballot. However:

1. This means that an additional layer of encryption on top of Helios's own encryption is needed to guarantee privacy. This contradicts the claim of the original paper [Adida 2008], where the Helios protocol alone guarantees privacy.

2. The installation instructions of Helios from the original GitHub repository lead to the Helios client connecting via HTTP by default. While it is possible to run the Helios server-side code on top of an SSL/TLS terminator, additional knowledge and expertise is required. The documentation neither explains this procedure nor even mentions its necessity.

3. Even when using HTTPS, the key pair for the SSL/TLS encryption differs from the custom key pair that is generated by Helios for each new

[4]If the browser does not support web workers, it is difficult to do this efficiently. Earlier versions of Helios used a technology known as LiveConnect to implement interaction between JavaScript and the browser's Java Runtime Environment [Adida 2008], yet this technology is being faded out and not commonly supported by modern browsers, so that support for this feature has since been removed from Helios.

election. In particular, the election public key may have been jointly computed by a set of trustees (such that no single entity knows its private part), while an administrator may well have access to the server's private HTTPS key. In addition, the server's key pair may change less frequently than the custom per-election key pairs.

In summary, running the Helios client in browsers that do not support web workers leads to a clear violation of vote privacy. Such browsers include Internet Explorer 9 and earlier, as well as all available versions of Opera Mini [Deveria 2017]. Depending on the actual statistics used, the combined worldwide percentage of people using affected browsers is between 12% and 36%.[5] A secure way to deal with these browsers would be to simply disallow them completely and prompt the voter to select a different browser. At the very least, this unexpected behavior should be clearly documented and plainly pointed out to election administrators. Currently, neither is the case.

When we notified the Helios authors of this vulnerability, they stated that they were not concerned, since in Helios, some inherent trust is placed in the server anyway. While they acknowledged that the claim of a *single-page web application* from the first paper is no longer true, they argued that the alternative of not supporting outdated browsers is unacceptable for practical real-world elections (since elections must be fair). They also pointed out that, even though the election server may indeed see plaintext ballots during the election process due to this behavior, there is no single-owner *long-term* storage of plaintext ballots. In summary, their point of view is that the need for usability and the support of a wide range of versions of all major browsers outweighs the threat to vote privacy.

4.5 Discussion and Takeaways

Although our analysis was performed on a modified version of the Helios client, the attacks also apply to the original client: We confirmed these vulnerabilities in an unmodified Helios client. The converse does not hold in general: We saw in Section 4.4.1 how an attack that allowed arbitrary modification of a set of variables can be amplified to arbitrary script execution. This was possible due to jQuery's internal behavior, and does not apply to the transformed code.

Therefore, our analysis of the transformed code is *sound* in the sense that any illegal information flow found can also be reproduced in the original code. However, it is not *complete*, as there may be harmful flows that it does not uncover. While our analysis of the transformed code is useful to

[5] Statistics taken from `http://gs.statcounter.com` and `http://netmarketshare.com`.

uncover previously unknown vulnerabilities, a positive result stating that no vulnerabilities were found in the transformed code can only be applied to the original code *modulo jQuery* (and any other third-party libraries), as these libraries themselves may contain vulnerabilities that may lead to exploits that our transformed code is inherently immune to.

The original Helios paper [Adida 2008] expected auditors to closely investigate the client-side JavaScript code, and that using the jQuery library would make it easier to understand and analyze the implementation due to its abstraction layer on top of low-level JavaScript functionalities, which makes the code more concise and easier to follow. However, as actual auditors of the code, we point out that this is not necessarily the case.

From a developer's perspective, modern browsers are more compatible than ever: Standardized and well-documented APIs for DOM traversal and manipulation, event handling or server communication have been adopted by all major browsers, decreasing the demand for jQuery [Schwartz & Bloom 2014]. While it might be argued that using jQuery eases support for older web browsers such as Internet Explorer 9 or earlier, we saw in Section 4.4.2 that supporting such browsers induces other challenges and potential vulnerabilities that cannot be solved by jQuery. Therefore the need to support old browsers in a voting client, which clearly cannot afford severe implementation-level vulnerabilities, is questionable.

From an analyst's perspective, automated analysis becomes much harder in the presence of libraries. Manual analysis modulo jQuery may be slightly easier, but implies putting blind trust in the security and behavior of a third-party library. As we have seen, this trust is not necessarily justified. Generally speaking, any potential vulnerabilities in a third-party library may inadvertently lead to vulnerabilities in the code they are employed in.

We conclude that a web application that does *not* rely on jQuery is easier to inspect and trust. Most of the functionality provided by jQuery can be implemented in native JavaScript code that runs in all modern browsers. The same applies, albeit to a slightly lesser degree, for the Underscore library and the implementation of class inheritance in JavaScript. As a side effect, we significantly reduced the code complexity when removing these libraries: With all uncompressed libraries included, the original version of the Helios client code amounts to almost 500 KB and over 9000 LOC in total. Without these libraries, the client has less than 250 KB and under 4000 LOC. Keeping dependencies low is a good idea both for security reasons and conciseness of the entire codebase. Note that the code transformations that we implemented to simulate the functionality of third-party libraries used by Helios can easily be exported into an external lightweight library, allowing our analysis to be easily reproduced in future versions of Helios.

We note that at the time of publication of the Helios paper, compatibility between browsers was a greater issue and that these libraries made a lot more sense. With the rapid development of browsers, performance of modern JavaScript engines and the availability of standardized, cross-platform JavaScript APIs, the client codebase could easily be reduced to less than half its size, easing code review processes and increasing trust in its implementation.

Clearly, our approach has limitations. In particular, the provided code transformations had to be applied manually. Although most of them could be automated, doing so for the plethora of existing libraries and APIs so as to generalize the approach to the majority of web applications is a daunting task. With the increasing compatibility between browsers and the continuous development of ECMAScript and its standard library, the need for third-party libraries and their use may decrease. Side effects of the standard library may be approximated, easing analysis of JavaScript applications in the future.

Finally, we point out that the vulnerabilities we found can easily be fixed. The confidentiality problem (Section 4.4.2) is a purposefully built (though questionable) feature that can be removed. For the integrity problem (Section 4.4.1), it suffices to sanitize the parameter obtained from the URL query string to ensure that it has the expected form. We stress the fact that although these attacks are simple, no manual code review has unveiled them so far, which highlights the benefits of an automated analysis.

4.6 Related Work

4.6.1 Conceptual Attacks on Helios

A multitude of approaches have been recently proposed to automatically ascertain central security properties for electronic voting [e.g., Backes *et al.* 2008, Delaune *et al.* 2009]. The analysis of Helios in particular has received tremendous attention from the scientific community. Note that none of the attacks mentioned in this section is related to the vulnerabilities we uncovered; instead, they target Helios on a conceptual level. Several publications investigate *privacy* of ballots in Helios. In particular, the related notion of *vote independence* has given rise to considerable debate: Vote independence means that by seeing a voter's encrypted ballot, another voter should not be able to cast a meaningfully related ballot.

Cortier and Smyth show that Helios does not satisfy vote independence and exploit this fact in order to compromise vote privacy [Cortier & Smyth 2011]. They discuss a countermeasure known as *ballot weeding*, and show that their revised scheme offers vote privacy in a symbolic model. Bernhard *et al.* define vote privacy in a computational model and prove that this revised version of Helios fulfills their definition, though only under non-standard assumptions [Bernhard *et al.* 2011]. Following along that line of work, Bernhard *et al.* study pitfalls of the Fiat-Shamir heuristic for non-interactive zero-knowledge proofs, which are used in Helios, and show that a stronger variant of the heuristic leads to ballot independence in Helios at lesser computational costs [Bernhard *et al.* 2012b] than the aforementioned revised version of Helios. Later, Smyth investigates an attack on vote privacy related to the one presented earlier by Cortier and Smyth [Smyth 2012], and contrasts the two aforementioned solutions. A different look at vote independence is put forth by Desmedt and Chaidos [Desmedt & Chaidos 2012]: The authors argue that the ability to create related ballots may in fact be desirable, since it enables voters to copy ballots from voters whom they trust without forcing these trusted voters to reveal their choice. They show that ballot copying is always feasible in Helios when the voter who casts the original ballot and the voter who copies cooperate, using a *ballot blinding* technique. Finally, Bernhard *et al.* define a *measure* for vote privacy in e-voting protocols and illustrate its usefulness by using Helios as an example [Bernhard *et al.* 2012a].

Helios puts an even greater concern on *verifiability* (both individual and universal) than on privacy, and thus, the extent to which Helios fulfills this expectation has also been thoroughly investigated in the literature. Kremer *et al.* put forth a formal definition of verifiability in a symbolic model and use it to analyze the Helios protocol [Kremer *et al.* 2010]. A more fine-grained model to assess the verifiability of e-voting protocols such as Helios is presented by Küsters *et al.* [Küsters *et al.* 2012]. They show that Helios is vulnerable to so-called *clash attacks*, wherein malicious administrators could surreptitiously replace a voter's ballot, and discuss countermeasures. Bernhard *et al.* show how the pitfalls of the Fiat-Shamir heuristic mentioned earlier may be exploited by colluding election administrators to break universal verifiability in Helios [Bernhard *et al.* 2012b]. Finally, Cortier *et al.* define the notions of *weak* and *strong* verifiability—corresponding to varying degrees of trust assumptions—in a computational model [Cortier *et al.* 2014]. They provide a generic way to transform weakly verifiable election schemes into strongly verifiable ones, apply their methodology to the variant of Helios by Bernhard *et al.* mentioned above, and show that the resulting scheme is strongly verifiable.

Sturton *et al.* implement a verifiably secure voting machine [Sturton *et al.* 2009]. In contrast to our work, their focus lies on direct-recording electronic voting machines, while the Helios voting client runs in a browser within an uncontrolled environment. Moreover, they design their system with the intent of making it amenable to verification from the start, while we verify an already deployed real-life system. Variants of Helios have been proposed, e.g., using mixnets instead of homomorphic tallying [Bulens *et al.* 2011], or using a threshold encryption scheme where only a subset of the trustees may proceed to tallying [Cortier *et al.* 2013]. Besides these, some usability studies [Karayumak *et al.* 2011a, Karayumak *et al.* 2011b] have been performed on Helios and improvements thereof.

4.6.2 Practical Attacks on Helios

Actual attacks against the implementation of Helios have not been reported prior to this work, respectively the corresponding conference publication [Backes *et al.* 2016]. Instead of exposing flaws in the implementation of Helios itself, related work has demonstrated exploits in incidental components using Helios as a case study. Estehghari and Desmedt show how vulnerabilities in Adobe Reader can be exploited in order to install a malicious browser rootkit that subverts the integrity of a user's vote in Helios [Estehghari & Desmedt 2010]. Their attack does not identify a vulnerability in Helios; it is only used as a case study. Similarly, Smyth and Pironti highlight logical web application flaws that arise from using TLS in an insecure manner, and also use Helios as a case study in order to show how this can be exploited to surreptitiously cast votes on behalf of honest voters [Smyth & Pironti 2013].

4.6.3 Static Analysis of JavaScript

During the last decade, there has been extensive research into information flow violations, which can break the integrity or confidentiality of programs. The notion of *noninterference* was introduced by Goguen and Meseguer [Goguen & Meseguer 1984]. Intuitively, every statement is assigned a security level; noninterference between two security levels means that no statement of the first security level may influence a statement of the second security level. Early approaches to analyze information flow violations focused predominantly on proving noninterference using type systems: Volpano *et al.* present a type-based algorithm that certifies noninterference for both implicit and explicit paths with respect to standard programming language semantics [Volpano *et al.* 1996]. Myers' Java Information Flow (Jif) framework [Myers 1999] enables tracking of information flow using annotations in the Java source code, while Shankar presents a taint analysis for C programs using a constraint-based type-inference engine [Shankar *et al.* 2001]. However, type-based approaches tend to be excessively complex and conservative.

Snelting *et al.* [Snelting *et al.* 2006] are the first to connect the notion of noninterference with program dependence graphs by showing that, intuitively, if a statement s_1 is in the backwards slice of a statement s_2, then the security level of s_1 interferes with the security level of s_2. Hammer *et al.* leverage this observation to implement an algorithm to check noninterference for Java programs [Hammer *et al.* 2006].

Indeed, despite the obvious dangers posed by vulnerable client-side code, research on static analysis has primarily focused on other languages more suited for server-side programming, most notably Java [e.g., Tripp *et al.* 2009]. In contrast, static analysis of JavaScript code is scarce, which may be due to the numerous challenges induced by the language's dynamic nature and other obstacles, as discussed in Section 4.1. One of the earliest works in this area was presented by Vogt *et al.*, who perform a *dynamic* taint analysis inside the browser to prevent XSS attacks, but use a simple static pass through the tainted scope to improve their results [Vogt *et al.* 2007]. In the same vein, Chugh *et al.* propose a staged approach [Chugh *et al.* 2009]: First, a static analysis is applied to as much of the code as possible, then residual analysis is performed when the code is dynamically loaded. Guha *et al.* use static analysis techniques to extract a model of expected client behavior from JavaScript programs as seen from the server and use it to build an intrusion-prevention proxy for the server [Guha *et al.* 2009].

Guarnieri and Livshits use their tool Gatekeeper to detect security problems according to different security policies in JavaScript widgets using static pointer analysis [Guarnieri & Livshits 2009]. In 2011, Guarnieri *et al.* present their tool Actarus, which enables purely static taint analysis for JavaScript code [Guarnieri *et al.* 2011]. While sound, their analysis has not been shown to scale to large applications. In a follow-up paper, Tripp *et al.* present Andromeda [Tripp *et al.* 2013] and improve scalability by computing data flow propagations and potentially vulnerable information flows on demand rather than to eagerly compute a complete data flow solution. Unfortunately, none of these tools is openly available.

Jensen *et al.* model the HTML DOM and browser API [Jensen *et al.* 2011] as an extension of earlier work on type analysis [Jensen *et al.* 2009]. Richards *et al.* [Richards *et al.* 2011] dispel common myths on the use of `eval` in a large-scale study. In turn, Jensen *et al.* show how `eval` can, in certain cases, be safely removed to aid static analysis [Jensen *et al.* 2012].

For JavaScript, research has been more intensively pursued in the area of dynamic analysis. As discussed earlier, dynamic analysis is more precise than static analysis, but it typically cannot cover all possible program paths and thus cannot be used to reason about all information flows [Sabelfeld & Myers 2003]. Since dynamic analysis is not the focus of this thesis, we refer the reader to works primarily concerned with dynamic JavaScript analysis [e.g., Askarov & Sabelfeld 2009, Meyerovich & Livshits 2010, Curtsinger *et al.* 2011, Hedin & Sabelfeld 2012, Hedin *et al.* 2016] for a deeper insight into this line of work.

4.7 Summary

We performed the first *implementation-level* analysis of the Helios JavaScript voting client. This analysis is relevant from two different perspectives.

First, Helios constitutes one of the most widely deployed and analyzed remote electronic voting protocols. Although its security properties have been an active subject of research, this research focused on a conceptual level, implicitly assuming that the implementation accurately reflects the protocol's intentions. We have shown that this is not necessarily the case and uncovered two severe flaws which, despite thorough investigations, a cautious implementation, and the vulnerabilities' simplicity, had remained unnoticed thus far. Considering highly sensitive systems such as remote electronic voting schemes—which may constitute a highly attractive target for extraordinarily resourceful adversaries—it is essential to analyze such systems not only on a conceptual, but also on a concrete implementation level. Our methodology addressed the gap between real-world and statically analyzable code, and we expect approaches in the same vein can be applied in a variety of related settings, finding vulnerabilities in client implementations that were overlooked during manual audits and conceptual or algorithmic investigations.

Second, we showed how to overcome the intricate technical challenges associated with analyzing a real-world JavaScript web application with a complex set of dependencies. This kind of research is highly relevant in a world where the number of web applications increases continuously and security is a growing and serious concern. We provided code transformations, replacing functionality that cannot be analyzed using current static analysis methods with functionally equivalent code. Using state-of-the-art tools of static analysis, we then reduced a highly complex system dependence graph consisting of 7 million nodes to a handful of potentially harmful flows that may compromise both privacy and integrity of the analyzed application. We then faithfully modeled all information flows of the program as a system dependence graph. Slicing reduced this 7 million node graph to a handful of potentially harmful information flows. Further inspection revealed that these flows correspond to actual vulnerabilities: a major XSS vulnerability, which was escalated to arbitrary script execution, and a minor flaw that led to leaking the plaintext ballot.

JavaScript is one of the core web technologies to devise client-side applications. On the server side, albeit the use of JavaScript is possible using frameworks such as Node.js, other languages dominate. One of the most popular languages for developing server-side web applications is PHP. In the next chapter, we present a new framework specifically designed to allow scanning PHP applications for vulnerabilities.

Chapter 5

A Framework for Static PHP Code Analysis

Contents

The most popular and widely deployed server-side language for web applications is undoubtedly PHP. Today, it powers more than 80% of the top ten million websites [W3Techs 2017b], including some of the Web's busiest platforms such as Facebook, Wikipedia, Flickr, or Wordpress, and contributes to almost 140,000 open-source projects on GitHub [Zapponi 2017]. Yet from a security standpoint, the language is poorly designed: It typically yields a large attack surface (e.g., every PHP script on a server can potentially be used as an entry point by an attacker) and bears inconsistently designed functions with often surprising side effects [Munroe 2012], all of which a programmer must be aware of and keep in mind while developing a PHP application.

To better understand the design of PHP, it is interesting to have a brief look at its history. Not unlike JavaScript, the design of the PHP language was not laid out after careful consideration and planning. On the contrary, originally it was not even a programming language. Between 1993 and 1994, when the world of web development was still young, Rasmus Lerdorf was developing dynamic back ends for various websites, mostly using C. Since this meant recompiling the entire web server whenever he made changes, he decided to add a standard library of very common web functions that he frequently needed to the web server. Additionally, he also enriched the web server with a simple state machine that had but two states: *in-HTML* mode and *in-tag* mode. The web server used this state machine to process templates: When it hit the end of a tag, it would lookup the string that it found inside the tag and call the matching C function in the library. Ultimately, the web server would substitute the output of the function for the tag. As Lerdorf shared his framework, various people and web companies kept asking him to add more functions to suit their needs to the library, and he obliged. Eventually, partly due to differing wishes of his users, partly to the advent of different browsers which made it necessary to serve different HTML code depending on a client, Lerdorf added logical tags to the macro language. Thus, control flow came into being in PHP, and a full-fledged programming language ultimately evolved from it. As Lerdorf put it himself,

> *"I don't know how to stop it, there was never any intent to write a programming language [...] I have absolutely no idea how to write a programming language, I just kept adding the next logical step along the way."*
>
> — Rasmus Lerdorf [Lerdorf 2003]

The first version of PHP was released in December of 1994. As more people started helping in the development of PHP, its standard library grew increasingly complex. This may explain, to a degree, why PHP today bears a patchwork of fixes and inconsistently designed functions. In addition, the PHP language appears to be well-suited for beginners and hobby programmers in web development, whose programming knowledge typically comprises HTML, CSS, and JavaScript. When they want to, say, connect their application to a database, they may not desire to make an extensive paradigm shift towards server-side web development technologies such as Java EE, JSP, Ruby on Rails, Python/Django, and so forth. PHP addresses this requirement with its shallow learning curve and ease of use.

As a result of both its confusing and inconsistent APIs and a lack of expertise of some of its users, PHP applications are particularly prone to programming mistakes that may lead to web application vulnerabilities such as SQL injections and cross-site scripting. Combined with its prevalence on the Web, PHP therefore constitutes a prime target for automated security analyses to assist developers in avoiding critical mistakes and consequently improve the overall security of applications on the Web. Indeed, a considerable amount of research has been dedicated to identifying vulnerable information flows for PHP code in a machine-assisted manner [Jovanovic *et al.* 2006, Jovanovic *et al.* 2010, Dahse & Holz 2014a, Dahse & Holz 2014b]. All these approaches successfully identify different types of PHP vulnerabilities in web applications. However, all of these approaches have only been evaluated in a controlled environment of about half a dozen projects. Therefore it is unclear how scalable they are and how well they perform in much less controlled environments of very large sets of arbitrary PHP projects. (See Section 5.6 on related work for details). In addition, these approaches are hardly customizable, in the sense that they cannot be configured to look for various different kinds of vulnerabilities.

The research question of how to detect PHP application vulnerabilities at large scale in an efficient manner, whilst maintaining an acceptable precision and the ability to customize the detection process as needed, has received significantly less attention so far. Yet it is a problem that is crucial to cope with, given the rapidly increasing number of web applications. In this chapter, we present a framework that addresses this problem.

Contributions. We propose a highly scalable and flexible approach for analyzing PHP applications that may consist of millions of lines of code. To this end, we leverage the recently proposed concept of *code property graphs* [Yamaguchi *et al.* 2014]: These graphs constitute a canonical representation of code incorporating a program's syntax, control flow, control dependencies, and data dependencies in a single graph structure, which we further enrich with call edges to allow for interprocedural analysis. These graphs are then stored in a graph database that lays the foundation for efficient and easily programmable *graph traversals* amenable to identifying flaws in program code. We show that this approach is well-suited to discover vulnerabilities in high-level, dynamic scripting languages such as PHP at a large scale. In addition, it is highly flexible: The bulk work of generating code property graphs and importing them into a database is done in a fully automated manner. Subsequently, an analyst can write traversals to query the database as desired so as to find various kinds of vulnerabilities: For instance, one may look to detect common code patterns or look for specific flows from given types of attacker-controller sources to given security-critical function calls that are not appropriately sanitized; what sources, sinks, and sanitizers are to be considered may be easily specified and adapted as needed.

We show how to model typical web application vulnerabilities using such graph traversals that can be efficiently run by the database back end and evaluate our approach on a set of 1,854 open-source PHP projects on GitHub. The three main contributions of this chapter are the following:

- *Introduction of PHP code property graphs.* We are the first to employ the concept of code property graphs for a high-level, dynamic scripting language such as PHP. We implement code property graphs for PHP using static analysis techniques and additionally augment them with call edges to allow for interprocedural analysis. These graphs are stored in a graph database that can subsequently be used for complex queries. The generation of these graphs is fully automated, that is, all that users have to do to implement their own interprocedural analyses is to write such queries. We make our implementation publicly available to facilitate independent research. To the best of our knowledge, this is the first open-source framework that allows to analyze PHP code in a fully customizable way, i.e., depending on an analyst's requirements.

- *Modeling web application vulnerabilities.* We show that code property graphs can be used to find typical web application vulnerabilities by modeling such flaws as graph traversals, i.e., fully programmable algorithms that travel along the graph to find specific patterns. These patterns are undesired flows from attacker-controlled input to security-critical function calls without appropriate sanitization routines. We detail such patterns precisely for attacks targeting both server and client, such as SQL injections, command injections, code injections, arbitrary file accesses, cross-site scripting and session fixation. These graph traversals demonstrate the feasibility of our technique. In addition, more traversals may easily be written by PHP application developers and analysts to detect other kinds of vulnerabilities or patterns in program code.

- *Large-scale evaluation.* To evaluate the efficacy of our approach, we report on a large-scale analysis of 1,854 popular PHP projects on GitHub totaling almost 80 million lines of code. In our analysis, we find that our approach scales well to the size of the analyzed code. In total, we find 78 SQL injection vulnerabilities, 6 command injection vulnerabilities, 105 code injection vulnerabilities, 6 vulnerabilities allowing an attacker to access arbitrary files on the server, and one session fixation vulnerability. XSS vulnerabilities are very common and our tool generated a considerable number of reports in our large-scale evaluation for this class of attack. Inspecting only a small sample (under 2%) of these reports, we find 26 XSS vulnerabilities.

Outline. The remainder of this chapter is organized as follows: In Section 5.1, we recapitulate the concept of code property graphs and briefly discuss how we augment them with call edges to allow for interprocedural analysis. In Section 5.2, we present a conceptual overview of our approach, follow up with the necessary techniques to represent and query PHP code property graphs in a graph database, and discuss how typical classes of vulnerabilities can be modeled using traversals. Subsequently, Section 5.3 presents the implementation of our approach, while Section 5.4 presents the evaluation of our large-scale study. Following this, Section 5.5 discusses our technique, Section 5.6 presents related work, and Section 5.7 summarizes this chapter.

5.1 Code Property Graphs

Our work builds on the concept of *code property graphs*, a joint representation of a program's syntax, control flow, and data flow, first introduced by Yamaguchi *et al.* [Yamaguchi *et al.* 2014, Yamaguchi 2015] to discover vulnerabilities in C code. The key idea of this approach is to merge classic program representations (see Chapter 3) into a so-called *code property graph*. More precisely, in the original paper, syntactical properties of the code are inferred from abstract syntax trees, control flow from the control flow graph, and data flow from program dependence graphs. By combining these structures into a single graph structure, we obtain a single global view enriched with information describing this code, called the code property graph. This joint representation is well-suited to mine program code for patterns linked to vulnerabilities, whether these vulnerabilities are due to purely syntactical mistakes, arise from vulnerable control or data flows, or a combination of these (see Chapter 3 for a discussion on vulnerability types). However, it does not yet allow us to reason about vulnerabilities that arise from control or data flows across function calls. Therefore, we also merge *call graphs* into the final structure so as to enable interprocedural analysis.

For illustration, recall the running example from Chapter 3, which we extend with the definition of the called function `query` as shown in Figure 5.1. The resulting code property graph of the entire system composed of the two functions `foo` and `query` is depicted in Figure 5.2. For the sake of illustration, this example suffers from a trivial SQL injection vulnerability. Using the techniques presented in this chapter, this vulnerability can be easily found.

As can be seen in Figure 5.2, the nodes of the code property graph are the same as the nodes of the abstract syntax tree (see Section 3.1.2), with the sole exception that the *ENTRY* and *EXIT* nodes known from the control flow graph (see Section 3.2.1) have been added to the code property graph.

```php
<?php
function foo() {

  $x = $_GET["id"];

  if(isset($x)) {
    $sql = "SELECT * FROM users
            WHERE id = '$x'";
    query($sql);
  }
}
?>
```

```php
<?php
function query( $sql) {

  mysql_query($sql);
}
?>
```

Figure 5.1: Example PHP code for the code property graph in Figure 5.2.

These two nodes, as well as the AST nodes that correspond to statements or predicates, are simultaneously the nodes of the control flow graph, and are highlighted in blue in Figure 5.2. Control flow is indicated by dotted arrows labeled with ϵ, true or false as discussed in Section 3.2.1. Our actual implementation has some additional constructs for handling `foreach` loops (that is, it has control flow edges labeled with next and complete) as well as for handling exceptions (for exceptions, statements within a `try` block are connected with a control flow edge labeled `exception` to the first statement of the corresponding `catch` block). Since the nodes of the program dependence graph are the same as the nodes of the control flow graph (except for the *ENTRY* and *EXIT* nodes), they can also be connected with control and data dependence edges: Consistently with Chapter 3, control dependence edges are dash-dotted, and data dependence edges are dashed in Figure 5.2. In addition, control dependence edges are annotated with C_{true} and C_{false}, and data dependence edges are annotated with variable names, as discussed in Section 3.3.3. Notably, we also consider the *PARAM* node for the parameter `sql` of the function `query` as a node of the control flow and program dependence graphs. This is due to the fact that a parameter can be seen, in a sense, as a statement which declares a variable: Note that there is a data dependence edge from the *PARAM* node to the *CALL* node of function `mysql_query`. This edge is essential for interprocedural analysis, as we will see in Section 5.2.3.4. Lastly, a loosely dotted arrow from the call node in function `foo` is connected to the function declaration node of function `query`: This is a call edge. Using call edges allows us to map calls to the called functions, and ultimately arguments to parameters, which is of paramount importance for an interprocedural analysis. Using call edges from callers to callees, as well as data dependence edges from parameters to the statements that use them, we can trace vulnerable information flow across functions, as we discuss in Section 5.2.3.4.

We now proceed to explain our methodology for discovering vulnerabilities in PHP code using these interprocedural code property graphs.

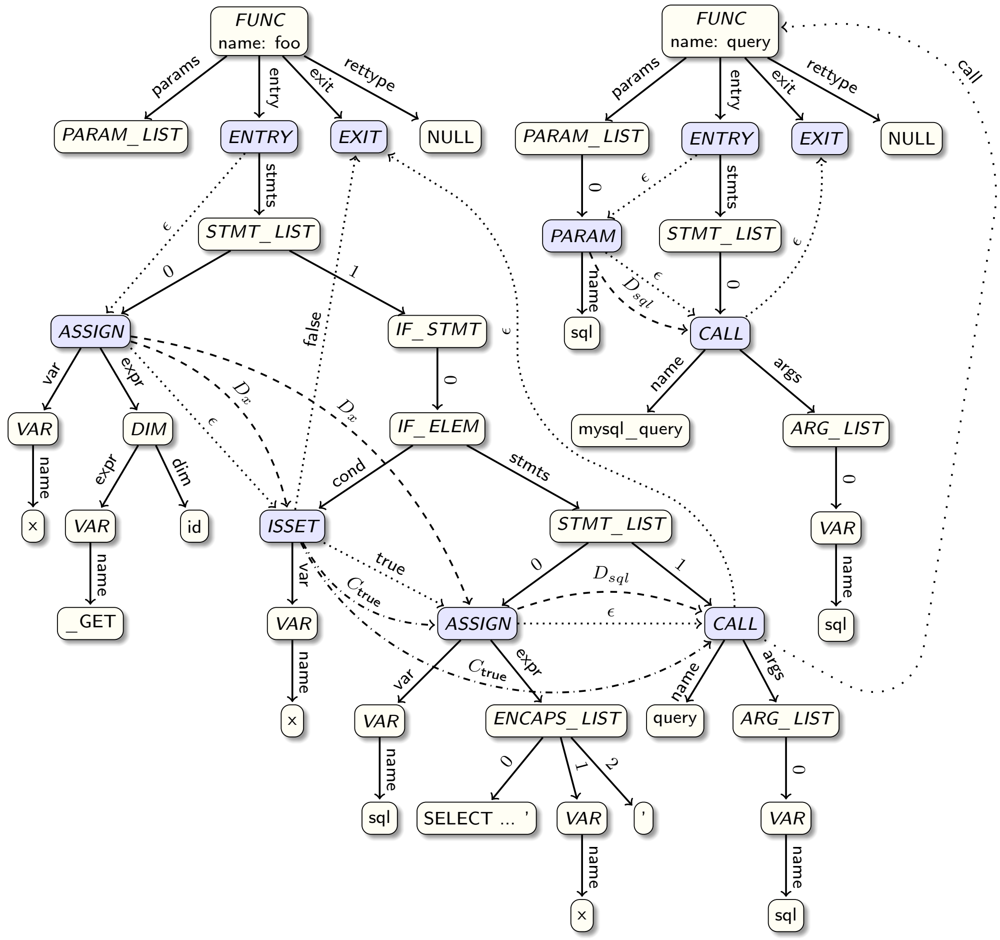

Figure 5.2: Interprocedural code property graph of the example in Figure 5.1. Nodes of the control flow and program dependence graphs are colored in blue, all other nodes are yellow. Solid arrows denote parental relationship of the abstract syntax tree. Dotted arrows indicate control flow, dash-dotted arrows indicate control dependencies and dashed arrows indicate data dependencies. The loosely dotted arrow represents a call edge.

5.2 Methodology

In this section, we present the methodology of our work. We first give a conceptual overview of our approach, discussing the representation and generation of code property graphs from PHP code. Subsequently, we discuss the viability of code property graphs for the purpose of finding web application vulnerabilities and introduce the notion of graph traversals. We then follow up with details on how different types of web application vulnerabilities can be modeled.

5.2.1 Conceptual Overview

Property graphs are a common graph structure featured by many popular graph databases such as Neo4J, OrientDB or Titan. A property graph (V, E) is a directed graph consisting of a set V of vertices (equivalently *nodes*) and a set E of edges. Every node and edge has a unique *identifier* and a (possibly empty) set of *properties* defined by a map from keys to values. In addition, nodes and edges may have one or more *labels*, denoting the type of the node or of the relationship.

Each of the structures presented in Chapter 3 captures a unique view on the underlying code. We define code property graphs as a combination of abstract syntax trees, control flow graphs, program dependence graphs and call graphs as detailed in Section 5.1. In particular, this definition extends the original definition [Yamaguchi *et al.* 2014] by incorporating call graphs to enable interprocedural analysis. Formally, a code property graph (V, E) is a property graph where the set of nodes V comprises the nodes of the abstract syntax tree as well as artificial *ENTRY* and *EXIT* nodes for each function. The set of edges E is the union of the set of edges of the abstract syntax tree, the control flow graph, the program dependence graph and the call graph. Nodes and edges are labeled and have a set of properties describing all relevant information as appropriate (we detail the relevant properties in the next sections).

The first step of our analysis is to prepare code property graphs for PHP code. This involves parsing the code and generating ASTs, then CFGs, then PDGs and finally call graphs. Next, the property graph is imported into a graph database. Subsequently, vulnerabilities can be described as patterns formulated as queries to the graph database. Sending these queries to the graph database outputs a set of suspicious paths which an analyst may then inspect. In this section, we describe the process of the generation of the code property graph in more detail, before we turn our attention to the graph database queries in the next section.

5.2.1.1 Parsing and Generation of Abstract Syntax Trees

Abstract syntax trees constitute the first step in our graph generation process. In order to model the code of an entire PHP project with syntax trees, we start by recursively scanning the directory for any PHP files. For each identified file, PHP's own internal parser [PHP Group 2014] is used to generate an AST representing the file's PHP code. The parse process is *robust* in the sense that no other resources besides this file are needed. In particular, if a PHP file imports other PHP files (i.e., using `include` or `require`), then the imported files are not needed to generate the AST of the file being parsed. Hence, even when only parts of the source code are known, our analysis can be performed only on these known parts (as opposed to a compiler or interpreter which requires the entire source code to build a binary or run a program).

Each node of the AST generated by the parser is a node of the property graph that we aim to generate: It is labeled as an `AST` node and has a set of properties. The first of these properties is a particular AST node type: For instance, there is a type for representing assignments, for function call expressions, for function declarations, etc. In all, there is a total of 103 different node types. For the sake of completeness, these are listed in Appendix B.1. Another property is a set of flags, e.g., to specify modifiers of a method declaration node. Further properties include a line number denoting the location of the corresponding code, and—in the case of leaf nodes—a property denoting the constant value of a particular node (such as the contents of a hardcoded string), as well as a few other technical properties that we omit here for simplicity. Edges of the AST bear the label `PARENT_OF`.

Additionally, a file node is created for the parsed file and connected to its AST's root node, and directory nodes are created and connected to each other and to file nodes in such a way that the resulting graph mirrors the project's filesystem hierarchy. File and directory nodes are labeled as `Filesystem` nodes with a a property storing their path and a flag to distinguish files and directories. Edges are labeled as `DIRECTORY_OF` when the source node is a directory, and as `FILE_OF` to connect a file node to a file's AST root node.

Finally, note that control flow graphs and program dependence graphs, which we want to generate next, are defined *per function* only (see Chapter 3). Yet PHP is a scripting language and commonly contains *top-level code*, i.e., there may be code in a PHP file that is not wrapped in a function, but executed directly by the PHP interpreter when loading the file. In order to be able to construct CFGs and PDGs for this code as well, we create an artificial *top-level function* AST node for each file during AST generation, holding that file's top-level code. This top-level function node constitutes the root node of any PHP file's syntax tree.

Similarly, since PHP is object-oriented, some code may be declared on the top-level scope of a given class (normally, this code contains field and method declarations). Such code belongs neither to the top-level code of a file, nor to any other explicitly declared function. Therefore, we also create artificial function nodes for classes to contain such code.

5.2.1.2 Control Flow Graphs

The next step before generating control flow graphs is to extract the individual function subtrees from the abstract syntax trees of the parsed files. Function subtrees in these ASTs may exist side by side, or may be nested within each other: For instance, a file's artificial top-level function may contain a particular function declaration, which in turn may contain a closure declaration, etc. We thus built a *function extractor* that extracts the appropriate subtrees for control flow graph and program dependence graph generation and is able to cope with nested functions: Whenever a function subtree contains another function subtree, the function extractor returns a subtree for each of the two functions: The subtree of the outer function contains the inner function's root node, but not its body. The inner function contains both its root node and its body. In the end, all PHP code is contained in a function, suitable for CFG and PDG generation. These subtrees are then individually processed by the CFG and PDG generating routines. Essentially, CFGs and PDGs are generated for all types of functions, and nested functions are properly handled.

To generate a control flow graph from an abstract syntax tree of a function, we first identify those AST nodes that are also CFG nodes, i.e., nodes that represent statements or predicates (see Figure 5.2). Control flow graphs can then be calculated from the AST by providing semantic information about all program statements that allow a programmer to alter control flow. Calculation is performed by defining translation rules from elementary abstract syntax trees to corresponding control flow graphs, and applying these to construct a preliminary control flow graph for a function. This preliminary control flow graph is subsequently corrected to account for unstructured control flow statements (see Section 3.2.1 for details). In addition, control flow graph generation also generates artificial *ENTRY* and *EXIT* nodes for each function and incorporates them in the code property graph as illustrated in Figure 5.2. These nodes are labeled as `Artificial` nodes and have a set of properties, such as a flag to distinguish *ENTRY* and *EXIT* nodes, the node id of the function node that they belong to, the function name and a few other (rather technical) properties. Edges of the CFG bear the label `FLOWS_TO` and have a property to indicate the condition of the flow.

5.2.1.3 Program Dependence Graphs

As explained in Section 3.3, program dependence graphs can be generated with the help of the control flow graph. For control dependencies, the post-dominator tree must be computed from the control flow graph first. This can be done generically without providing any additional semantic information, as shown in Section 3.3.1. Data dependencies are slightly more involved, as they require us to run a *use/def analysis* on the individual nodes of the control flow graph to determine, for each statement or predicate, which variables are used and which variables are (re-)defined. This clearly requires additional semantic information. As a simple example, we know that the variable on the left of an assignment node is *defined* in the assign statement, whereas all variables on the right of the assignment node are *used* in the assign statement. Once this information has been calculated for each CFG node, that information is propagated backwards along the control flow edges to solve the reaching definitions problem, as detailed in Section 3.3.2. Its solution gives rise to the data dependence edges of the program dependence graph. The generated edges bear the labels `CONTROLS` for control dependencies and `REACHES` for data dependencies, with properties to indicate the condition of a control dependence, respectively the name of the variable in the case of a data dependence.

5.2.1.4 Call Graphs

The final step in our graph generation process is the generation of call graphs. This must be done at the end since we need to first store all function nodes to be able to map all call nodes appropriately. More precisely, at the time of generation of the abstract syntax trees, we keep track of all call nodes that we encounter, as well as of all function declaration nodes. Once we finish the parsing process for all files of a particular project (and we can thus be confident that we have collected all function declaration nodes), those call nodes are connected to the corresponding function declaration nodes with call edges (labeled `CALLS`). We naturally resolve namespaces (`namespace X`), imports (`use X`) and aliases (`use X as Y`) at parse time. Function names are resolved within the scope of a given project, i.e., we do not need to analyze `include` or `require` statements, which are often only determined at runtime; instead, all functions declared within the scope of a project are known during call graph generation. Note that there are four types of calls in PHP: function calls (`foo()`), static method calls (`A::foo()`), constructor calls (`new A()`), and dynamic method calls (`$a->foo()`). The first three types are mapped unambiguously. For the last type, we only connect a call node to the corresponding method declaration if the called method's name is unique within the project, or if the object reference is `$this` (as in `$this->foo()`), since these references can be resolved statically without ambiguities. If several methods with the same name are known from different classes and a reference different from `$this` is used, we do not construct a call edge, as that would require a highly involved type inference process for PHP that is out of the scope of this project (and indeed, since PHP is a dynamically typed language and because of its ability for reflection, it is not even possible to statically infer every object's type). However, looking at the empirical study conducted on 1,854 projects that we present in Section 5.4, we can report that this approach allowed us to correctly map 78.9% of all dynamic method call nodes. Furthermore, out of a total of 13,720,545 call nodes, there were 30.6% function calls, 54.2% dynamic method calls, 6.4% constructor calls, and 8.8% static method calls. This means that 88.6% of all call nodes were successfully mapped in total.

5.2.1.5 Combined Code Property Graph

The final graph represents the entire codebase including the project's structure, syntax, control flow, and data dependencies as well as interprocedural calls. It is composed of `AST`, `Filesystem`, and `Artificial` nodes (where the majority of nodes are AST nodes, while filesystem nodes represent files and directories, and artificial nodes are used for entry and exit nodes of functions). Some of the AST nodes (namely, those AST nodes representing statements or predicates) are simultaneously CFG and PDG nodes. Additionally, the code property graph has seven types of edges: directory edges, file edges, syntax tree edges, control flow edges, control dependence edges, data dependence edges, and call edges. This graph is the foundation of our analysis.

5.2.2 Graph Traversals

Code property graphs can be used in a variety of ways to identify vulnerabilities in applications. For instance, they may be used to identify common code patterns known to contain vulnerabilities on a syntactical level, while abstracting from formatting details or variable names; to identify control-flow type vulnerabilities, such as failure to release locks; or to identify taint-style type vulnerabilities, such as attacker-controlled input that flows into security-critical function calls, etc.; see Chapter 3 for a detailed discussion.

Graph databases are optimized to contain heavily connected data in the form of graphs and to efficiently process graph-related queries. As such, they are an ideal candidate to contain our code property graphs. Then, finding vulnerabilities is only a matter of writing meaningful database queries that identify particular patterns and control/data flows an analyst is interested in.

Such database queries are written as *graph traversals*, i.e., fully programmable algorithms that travel along the graph to collect, compute, and output desired information as specified by an analyst. Graph databases make it easy to implement such traversals by providing a specialized graph traversal API.

Apart from logic bugs, most of the vulnerabilities which occur in web applications can be abstracted as *information-flow problems* violating *confidentiality* or *integrity* of the application, as we discussed in Section 3.5. A breach of confidentiality occurs when secret information, e.g., database credentials, leaks to a public channel, and hence to an attacker. In contrast, attacks on integrity are data flows from an untrusted, attacker-controllable source, such as an HTTP request, to a security-critical sink. To illustrate the use of code property graphs to identify vulnerabilities, we focus on information-flow vulnerabilities threatening the integrity of applications. Given a specific application for which we can determine what data should be kept secret, finding breaches of confidentiality is equally possible with this technique. However, for doing so at scale, the core problem is that it is hard or even impossible to define *in general* what data of an application should be considered confidential and must therefore be protected. Thus, to find information-flow vulnerabilities violating confidentiality would require us to take a closer look at each application and identify confidential data—such as we did in Chapter 4, where we considered the variable containing the votes cast by a voter in the Helios voting client as confidential: This required a manual inspection of the source code to identify the confidential variable. In contrast, it is generally much easier to determine what data originates from an untrusted source and to identify several types of generally security-critical sinks. We discuss these sources and security-critical function calls in the context of PHP code in Section 5.2.3. Since we are interested in performing a large-scale analysis, we concentrate on threats targeting the integrity of an application.

Before we proceed to more complex traversals to find information flows, we implement *utility traversals* that are aware of our particular graph structure as well as the information it contains and define typical travel paths that often occur in this type of graph. These utility traversals are used as a base for more complex traversals. For instance, we define utility traversals to travel from an AST node to its enclosing statement, its enclosing function, or its enclosing file, traversals to travel back or forth along the control or the data flow, and so forth. We refer the reader to the work by Yamaguchi *et al.* [Yamaguchi *et al.* 2014] for a more detailed discussion of utility traversals.

5.2.3 Modeling Vulnerabilities

As discussed before, although our methodology can be applied to detect confidentiality breaches, we cannot do this for large-scale analyses, due to the inherent lack of a notion of *secret data*. Hence, we focus on threats to the integrity of an application. Even though we are conducting an analysis of server-side PHP code, we are not limited to the discovery of vulnerabilities resulting in attacks which target the server side (e.g., SQL injections or command injections). For example, cross-site scripting and session fixation can be caused by insecure server-side code, but clearly target clients. Our analysis allows us to detect both attacks, i.e. attacks targeting the server and attacks targeting the client. In the remainder of this section, we first discuss sources which are directly controllable by an attacker. Subsequently, we follow up with discussions of attacks targeting the server and attacks targeting the client, all of which we aim to discover in our large-scale case study in Section 5.4. We finish by describing the process of detecting illicit flows.

5.2.3.1 Attacker-Controllable Input

In the context of a web application, all data which is directly controllable by an attacker must be transferred in an HTTP request. For the more specific case of PHP, this data is contained in multiple global associative arrays. Among these, the most important ones are [PHP Group 2017b]:

- `$_GET`: This array contains all GET parameters, i.e., a key/value representation of parameters passed in the URL. Although the name might suggest otherwise, this array is also present in POST requests, containing the URL parameters.
- `$_POST`: All data which is sent in the *body* of a POST request is contained in this array. Similarly to `$_GET`, this array contains decoded key/value pairs, which were sent in the POST body.
- `$_COOKIE`: Here, PHP stores the parsed cookie data contained in the request. This data is sent to the server in the `Cookie` header.
- `$_REQUEST`: This array contains the combination of all the above. The behavior in cases of collisions can be configured, such that, e.g., `$_GET` is given precedence over `$_COOKIE`.
- `$_SERVER`: This array contains different server-related values, e.g., the server's IP address. More interestingly, all headers transferred by the client are accessible via this array, e.g., the user agent. For our analysis, we consider accesses to this array for which the key starts with `HTTP_`,

since this is the default prefix for parsed HTTP request headers, as well as accesses for which the key equals `QUERY_STRING`, which contains the query string used to access the page.

- `$_FILES`: Since PHP is a web programming language, it naturally accepts file uploads. This array contains information on and content of uploaded files. Since, e.g., MIME type and file name are attacker-controllable, we also consider this as a source for our analysis.

We also consider some legacy variables to account for PHP code written for older versions of the PHP interpreter. The exact sources that we consider for each vulnerability type we are interested in are listed in Appendix B.2.

The values of all of these variables can be controlled or at least influenced by an attacker. In the case of GET and POST parameters, an attacker may even cause an innocuous victim to call a PHP application with an input of their choice (e.g., using forged links), while the attacker can usually only modify their own cookies. Yet all of them can be used by an attacker to call an application with unexpected input, allowing them to trigger contained vulnerabilities.

5.2.3.2 Attacks Targeting the Server

For server-side attacks, a multitude of vulnerability classes has to be considered. In the following, we recall each of the classes that we are interested in from Chapter 2. More importantly, we detail, for the specific case of PHP, the corresponding security-critical function calls that may induce a vulnerability when used improperly. We also detail specific sanitizers which ensure (when used properly) that a flow cannot be exploited. The full list of sanitizers that we consider in the context of each vulnerability can be found in Appendix B.2.

SQL Injections (cf. Section 2.2.1) are vulnerabilities in which an attacker exploits a flaw in the application to inject SQL commands of their choosing. While, depending on the database, the exact syntax is slightly different, the general concept is the same for all database engines. Here, we look for three major sinks, namely `mysql_query`, `pg_query`, and `sqlite_query`. For each of these, specific sanitizers exist in PHP, such as for instance `mysql_real_escape_string`, `pg_escape_string` or `sqlite_escape_string`.

Command Injection (cf. Section 2.2.2) is a type of attack in which the goal is to execute commands on the shell. More specifically, PHP offers different ways of running an external program: A programmer may use `popen` to execute a program and pass arguments to it, or they can use `shell_exec`, `passthru`, or backtick operators to invoke a shell command. PHP provides the functions `escapeshellcmd` and `escapeshellarg`, which can be used to sanitize commands and arguments, respectively.

Code Injection (cf. Section 2.2.3) attacks occur when an adversary is able to force the application to execute PHP code of their choosing. Due to its dynamic nature, PHP allows the evaluation of code at runtime using the language construct `eval`. In cases where user input is used in an untrusted manner in invocations of `eval`, this can be exploited to execute arbitrary PHP code. As the necessary payload depends on the exact nature of the flawed code, there is no general sanitizer which may be used to thwart all these attacks.

In addition, PHP applications might be susceptible to *file inclusion* attacks. In these, if an attacker can control the values passed to `include` or `require`, which read and interpret the passed file, PHP code of their choosing can also be executed. If the PHP interpreter is configured accordingly, even remote URLs may be used as arguments, resulting in the possibility to load and execute remote code. However, even when the PHP interpreter is configured to evaluate local files only, vulnerabilities may arise: For instance, if a server is shared by several users, a malicious user might create a local PHP file with malicious content, make it world-readable and exploit another user's application to read and execute that file. Another scenario would be that a PHP file already exists that, when included in the wrong environment, results in a vulnerability.

Arbitrary File Reads/Writes (cf. Section 2.2.4) can result when some unchecked, attacker-controllable input flows to a call to `fopen`. Based on the applications and the access mode used in this call, an attacker can therefore either read or write arbitrary files. In particular, an attacker may use `..` in their input to traverse upwards in the directory tree to read or write files unexpected by the developer. These vulnerabilities are often defended against by using regular expressions, which aim to remove, e.g., dots from the input.

5.2.3.3 Attacks Targeting the Client

Apart from the previously discussed attacks which target the server, we review two additional classes of flaws which affect the client, discussed in detail in Chapter 2. More specifically, these are cross-site scripting and session fixation. We now outline the corresponding security-critical function calls for the specific case of PHP.

For these types of vulnerabilities, cookies are not a critical source. This is due to the fact that an attacker cannot modify the cookies of their victim (without having exploited the XSS in the first place). Rather, they can forge HTML documents which force the victim's browser to send GET or POST requests to the flawed application. In Appendix B.2, we list the exact sources and sanitizers that we deem appropriate in the context of each vulnerability.

Cross-Site Scripting (XSS) (cf. Section 2.3.1) is an attack in which the attacker is able to inject JavaScript code in an application. More precisely, the goal is to have this JavaScript code execute in the browser of a desired victim. Since JavaScript has full access to the document currently rendered, this allows the attacker to control the victim's browser in the context of the vulnerable application. In the specific case of PHP, a reflected cross-site scripting attack may occur when input from the client is *reflected* back in the response using the calls `echo` or `print`. For these attacks, PHP also ships built-in sanitizers. We consider these, such as `htmlspecialchars`, `htmlentities`, or `strip_tags`, as valid sanitizers in our analysis.

Session Fixation (cf. Section 2.3.2) is the last vulnerability we consider. The attack here is a little less straightforward compared to those previously discussed. In essence, an attacker obtains a valid session identifier for a website (e.g., they browse to the website and use the one assigned to them), then tricks their victim into using that session identifier. By default, PHP uses cookies to manage sessions. Hence, if there is a flaw which allows overwriting the session cookie in the victim's browser, this can be exploited by the adversary. To successfully impersonate their victim, the attacker forcibly sets the session cookie of their victim to their own. If the victim now logs in to the application, the attacker also gains the same privileges. To find such vulnerabilities, we analyze all data flows into the PHP function call `setcookie`. As there is no generic way to protect against this attack, we cannot model a specific sanitizer to filter benign flows.

5.2.3.4 Detection Process

After having discussed the various types of flaws we consider, we now outline the graph traversals used to find flaws in applications. To optimize efficiency, we in fact perform two consecutive queries for each class of vulnerabilities that we are interested in.

Indexing critical function calls. The first query returns a list of identifiers of all AST nodes that correspond to a given security-critical function call. For instance, it finds all nodes that correspond to call expressions to the function `mysql_query`. The reason for doing so is that we may then work with this index for the next, much more complex traversal, which attempts to find flows to these nodes from attacker-controllable inputs, instead of having to touch every single node in the graph. As an example, Figure 5.3 shows the Cypher query (see Section 5.3) that we use to identify all nodes representing `echo` and `print` statements. (It is straightforward, since `echo` and `print` are language constructs in PHP, i.e., they have a designated node type). If done right, such an index can be generated by the graph database back end in a highly efficient

```
MATCH (node:AST)
USING INDEX node:AST(type)
WHERE node.type IN ["AST_ECHO", "AST_PRINT"]
RETURN node.id;
```

Figure 5.3: Sample indexing query in Cypher.

manner, as we will see in Section 5.4. The Cypher queries for all security-critical function calls we are interested in can be found in Appendix B.2.

Identifying critical data flows. The second query is more complex. Its main idea is depicted in Figure 5.4. Its purpose is to find critical data flows that end in a node corresponding to a security-critical function call.

For each node in the index generated by the previous traversal, the function `init` is called, a recursive function whose purpose is to find such data flows even across function borders. It first calls the function `visit`, which starts from the given node and travels backwards along the data dependence edges defined by the PDG using the utility traversal `sources`; it only travels backwards those data dependence edges for variables which are not appropriately sanitized in a given statement. It does so in a loop until it either finds a low source, i.e., an attacker-controllable input, or a function parameter. Clearly, there may be many paths that meet these conditions; they are all handled in parallel, as each of the utility traversals used within the function `visit` can be thought of as a *pipe* which takes a set of nodes as input and outputs another set of nodes. The loop emits only nodes which either correspond to a low source or a function parameter. Finally, for each of the nodes emitted from the loop, the step `path` outputs the paths that caused these nodes to be emitted. Each of these paths corresponds to a flow from either a parameter or a low source to the node given as argument to the function. Note that since we travel *backwards*, the head of each path is actually the node given as argument, while the last element of each path is a parameter or low source.

Back in the function `init`, the list of returned paths is inspected. Those paths whose last element is not a parameter (but a low source) are added to the final list of reported flows. For those paths whose last element is indeed a parameter, we perform an interprocedural jump in the function `jumpToCallSiteArgs`: We travel back along all call edges of the function defining this parameter to the corresponding call expression nodes, map the parameter to the corresponding argument in that call expression, then recursively apply the overall traversal to continue traveling back along the data dependence edges from that argument for each call expression that we traveled to. After the recursion, the returned paths are connected to the found paths in the called function. For the sake of presentation, the simplified code in

```
def init( Vertex node) {

  finalflows = [];

  varnames = getUsedVariables( node); // get USEs in node
  flows = visit( node, varnames); // get list of flows

  for( path in flows) {

    if( path.last().type == TYPE_PARAM) {

      callSiteArgs = jumpToCallSiteArgs( path.last());

      callingFuncFlows = [];
      for( Vertex arg in callSiteArgs) {
        callingFuncFlows.addAll( init( arg)); // recursion
      }
      // connect the paths
      for( List callingFuncFlow : callingFuncFlows) {
        finalflows.add( path + callingFuncFlow);
      }
    }
    else {
      finalflows.add( path);
    }
  }

  return finalflows;
}

def visit( Vertex sink, List varnames) {

  sink
  .statements() // traverse up to CFG node
  .as('datadeploop')
    .sources( varnames)
    .sideEffect{ varnames = getUnsanitizedVars( it) }
    .sideEffect{ foundsrc = containsLowSource( it) }
  .loop('datadeploop'){ !foundsrc && it.type != TYPE_PARAM }
  .path()
}

def jumpToCallSiteArgs( Vertex param) {

  param
  .sideEffect{ paramNum = it.childnum }
  .function() // traverse to enclosing function
  .functionToCallers() // traverse to callers
  .callToArgumentList() // traverse to argument list
  .children().filter{ it.childnum == paramNum }
}
```

Figure 5.4: (Simplified) path-finding traversal in Gremlin.

```php
<?php
function foo() {
  $a = $_GET['a'];
  $b = $_GET['b'];
  bar( $a, $b);
}

function bar( $a, $b) {
  $c = $_GET['c'];
  echo $a.$c;
}
?>
```

Figure 5.5: PHP code illustrating the graph traversal for XSS vulnerabilities.

Figure 5.4 glosses over some technicalities, such as ensuring termination in the context of circular data dependencies or recursive function calls, or tackling corner cases such as sanitizers used directly within a security-critical function call, but conveys the general idea. The interested reader may find the complete function in Appendix B.2.

The end result output by the path-finding traversal is a set of interprocedural data dependence paths (i.e., a set of lists of nodes) starting from a node dependent on an attacker-controllable source and ending in a security-critical function call, with no appropriate sanitizer being used along the way. These flows correspond to potential vulnerabilities and can then be investigated by a human expert in order to either confirm that there is a vulnerability, or determine that the flow cannot actually be exploited in practice.

As an example, consider the PHP code in Figure 5.5. Starting from the `echo` statement, the traversal travels the data dependence edges backwards both to the assignment of `$c` and to the parameter `$a` of function `bar`. The assignment of `$c` uses a low source without an appropriate sanitizer, hence this flow is reported. In the case of the parameter `$a`, the traversal travels to the call expression of function `bar` in function `foo` and from there to argument `$a`, then recursively calls itself starting from that argument. Since `$a` likewise originates from a low source without sanitization, this flow is reported too. Note that even though variable `$b` also originates from a low source and is passed as an argument to function `bar`, the parameter `$b` does not flow into the `echo` statement and hence, no flow is reported in this case.

5.3 Implementation

To generate ASTs for PHP code, we leverage a PHP extension[1] which exposes the PHP ASTs internally generated by the PHP 7 interpreter as part of the compilation process to PHP userland. Our parser utility generates ASTs for PHP files, then exports those ASTs to a CSV format. As described in Section 5.2.1, it also scans a directory for PHP files and generates file and directory nodes reflecting a project's structure. Using PHP's own internal parser to generate ASTs, instead of, say, writing an ANTLR grammar ourselves, means that AST generation is well-tested and reliable. Additionally, we inherently support the new PHP 7 version including all language features. At the same time, parsing PHP code written in older PHP versions works as well. Some PHP features have been removed in the course of time, and executing old PHP code with a new interpreter may cause runtime errors—however, such code can still be parsed, and the non-existence of a given function (for example) in a newer PHP version does not impede our analysis.

For our database back end, we leverage Neo4J, a popular open-source graph database written in Java. The CSV format output by our parser utility can be directly imported into a Neo4J database using a fast batch importer for huge datasets shipped with Neo4J. This allows us to efficiently access and traverse the graph and to take advantage of the server's advanced caching features for increased performance.

In order to generate CFG, PDG, and call edges, we implemented a fork of Joern [Yamaguchi *et al.* 2014], which builds similar code property graphs for C. We extended Joern with the ability to import the CSV files output by our PHP parser and map the ASTs that they describe to the internal Joern representation of ASTs, extending or modifying that representation where necessary. We then extended the CFG and PDG generating code in order to handle PHP ASTs. Next, we implemented the ability to generate call graphs in Joern. Finally, we added an export functionality that outputs the generated CFG, PDG, and call edges in CSV format. These edges can thus be imported into the Neo4J database simultaneously with the CSV files output by our parser.

The flow-finding graph traversals described in Section 5.2.3.4 are written in the graph traversal language Gremlin,[2] which builds on top of Groovy, a JVM language. In addition to Gremlin, Neo4J also supports Cypher, an SQL-like query language for graph databases which is geared towards simpler queries, but is also more efficient for such simple queries. We use Cypher for the indexing query of security-critical function calls described in the previous

[1] `https://github.com/nikic/php-ast`

[2] `http://tinkerpop.incubator.apache.org`

section. Both Gremlin and Cypher scripts are sent to the Neo4J server's REST API endpoint and the queries' results are processed using a thin Python wrapper.

Our tool is free open-source software and has been integrated into the Joern framework, available at:

https://github.com/octopus-platform/joern

5.4 Evaluation

In this section, we evaluate our implemented approach. We first present the dataset used and follow up with a discussion of the findings targeting both server and client.

5.4.1 Dataset

Our aim was to evaluate the efficacy of our approach on a large set of projects in a fully automated manner, i.e., without any kind of preselection by a human. We used the GitHub API in order to randomly crawl for projects that are written in PHP and have a rating of at least 100 stars to ensure that the projects we analyze enjoy a certain level of interest from the community.

As a result, we obtained a set consisting of 1,854 projects. We then applied our tool to build code property graphs for each of these projects, and imported all of these code property graphs into a single graph database that we subsequently ran our analysis on.

As a final step before the actual analysis, we proceeded to create an *index* of AST node types in the graph database. An index is a redundant copy of information in the database with the purpose of making the retrieval of that information more efficient. Concretely, it means that we instructed the database back end to create an index that maps each of the 103 different AST node types to a list of all node identifiers that have the given type. We can thus efficiently retrieve all AST nodes of any given type. This approach makes the identification of nodes that correspond to security-critical function calls (i.e., the first query as explained in Section 5.2.3.4) more efficient by several orders of magnitude.

On such a large scale, it is interesting to see how well our implementation behaves in terms of space and time. We performed our entire analysis on a machine with 32 physical 2.60 GHz Intel Xeon CPUs with hyperthreading and 768 GB of RAM. The time measurements for graph generation and the final size of the database are given in Table 5.1.

Statistics on database generation	
AST generation	40m 30s
CFG, PDG, and call edge generation	5h 10m 33s
Graph database import	52m 11s
AST node type indexing	3h 1m 32s
Database size (before indexing)	56 GB
Database size (after indexing)	66 GB

Table 5.1: Statistics on database generation.

Upon inspection of the crawled dataset, we judged that it would be sensible to distinguish two subsets of projects with respect to our analysis:

- $\mathcal{C}$: Among the crawled 1,854 projects, we found that 4 were explicitly vulnerable software for educational purposes, or web shells. In this set, we expect a large number of unsanitized flows, as these projects contain such flows *on purpose*. Therefore, this set of projects can be seen as a sanity check for our approach to find unsanitized flows: If it works well, we should see a large number of reports. We show that this is indeed the case.
- $\mathcal{P}$: This is the set of the remaining 1,850 projects. Here we expect a proportionally smaller set of unsanitized flows, as such flows may correspond to actually exploitable vulnerabilities.

In Table 5.2, we present statistics concerning the size of the projects and the resulting code property graphs in the two sets $\mathcal{P}$ and $\mathcal{C}$. All in all, the total number of lines of code that we analyze amounts to almost 80 million, with the smallest project consisting of only 22 lines of code, and the largest consisting of 2,985,451 lines of code. The complete list of projects can be found in Appendix B.3. To the best of our knowledge, this is the largest collection of PHP code that has been scanned for vulnerabilities in a single study.

The resulting code property graphs consist of over 300 million nodes, with about 26 million CFG edges, 15 million PDG edges, and 4 million call edges. The number of AST edges plus the number of files equals the number of AST nodes, since each file's AST is a tree. Evidently, there are many more AST edges than CFG or PDG edges, since control flow and data dependence edges only connect AST nodes that correspond to statements or predicates.

Concerning the time needed by the various traversals as reported in the remainder of this section, we note that on the one hand, a large number of CPUs is not necessarily of much help, since a traversal is hard to parallelize automatically for the graph database server. The presence of a large memory, on the other hand, enables the entire graph database to live in memory; we expect this to yield a great performance increase, although we have no direct

	$\mathcal{P}$	$\mathcal{C}$
# of projects	1,850	4
# of PHP files	428,796	952
# of LOC	77,722,822	356,400
# of AST nodes	303,105,896	1,955,706
# of AST edges	302,677,100	1,954,754
# of CFG edges	25,447,193	197,656
# of PDG edges	14,459,519	187,785
# of call edges	3,661,709	25,747

Table 5.2: Dataset and graph sizes.

time measurements to compare to, as we did not run all our traversals a second time with a purposefully small heap only to force I/O operations.

5.4.2 Findings

In this section, we present the findings of our analysis. As detailed in Section 5.2.3, our approach aims to find vulnerabilities which can be used to attack either server or client, and we discuss these in Sections 5.4.2.1 and 5.4.2.2, respectively. For every type of security-critical function call, we consider different sets of sanitizers as valid (see Section 5.2.3). However, for all of them, we consider the PHP functions `crypt`, `md5`, and `sha1` as a sufficient transformation of attacker-controlled input to safely embed it into a security-critical function call. Additionally, we accept `preg_replace` as a sanitizer; this is fairly generous, yet since we evaluate our approach on a very large dataset, we want to focus on very general types of flows. (In contrast, when using our framework for a specific project, it could be fine-tuned to find very specific flows, e.g., we could consider `preg_replace` as a sanitizer only in combination with a given set of regular expressions).

5.4.2.1 Attacks Targeting the Server

SQL Injection. For SQL injections, we ran our analysis separately for each of the security-critical function calls `mysql_query`, `pg_query`, and `sqlite_query`. The large majority of our findings was related to calls to `mysql_query`. Our findings for `mysql_query` and `pg_query` are summarized in Tables 5.3 and 5.4. In the case of `sqlite_query`, our tool discovered 202 calls in total, but none of these were dependent on attacker-controllable inputs, hence we omit a more detailed discussion for this function.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 19s	
Pathfinder traversal	34m 32s	
`mysql_query` calls	3,098	963
Sinks (Flows)	322 (2,023)	171 (244)
Vulnerabilities	74	-

Table 5.3: Evaluation for `mysql_query`.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 16s	
Pathfinder traversal	3m 42s	
`pg_query` calls	326	55
Sinks (Flows)	6 (6)	5 (7)
Vulnerabilities	4	-

Table 5.4: Evaluation for `pg_query`.

Tables 5.3 and 5.4 show the time needed for the indexing query to find all function calls to `mysql_query` and `pg_query`, respectively, and for the traversals to find flows from attacker-controllable inputs to these calls. Furthermore, they show the total number of function calls found in both the sets $\mathcal{P}$ and $\mathcal{C}$, i.e., the number of nodes output by the indexing query. Then, they show the total number of *sinks*, i.e., the size of the subset of these function calls which do indeed depend on attacker-controllable input without using an appropriate sanitization routine. The number in parentheses denotes the total number of flows, that is, the number of paths reported by the pathfinder traversal which have one of these sinks as an endpoint. Finally, the tables show the number of *vulnerabilities*: We investigated all reports from our tool and counted the number of actually exploitable vulnerabilities. Here, a vulnerability is defined as a sink for which there exists at least one exploitable flow. Thus, the number of vulnerabilities should be compared to the number of reported sinks, as multiple exploitable flows into the same sink are only counted as a single vulnerability. We do not report on vulnerabilities in $\mathcal{C}$ due to the fact that these projects are intentionally vulnerable. However, we analyzed these reports and confirmed that they do indeed point to exploitable flows in most cases. In those cases where the flows are not exploitable, input is checked against a whitelist or regular expression, or sanitized using custom routines.

As a result of our manual inspection, we found that 74 out of 322 sinks for `mysql_query` were indeed exploitable by an attacker, which yields a good hit rate of 22.9%. For `pg_query`, we performed even better: We found that 4 out of 6 sinks were indeed vulnerable.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	2m 28s	
Pathfinder traversal	13m 14s	
`shell_exec` / `popen` calls and backtick operators	1,598	270
Sinks (Flows)	19 (47)	64 (1,483)
Vulnerabilities	6	-

Table 5.5: Evaluation for `shell_exec`, `popen`, and the backtick operator.

Among the flows that we deemed non-critical, we found that many could be attributed to *trusted areas* of web applications, i.e., areas that only a trusted, authenticated user, such as an administrator or moderator, can access in the first place. Such flows may still result in exploitable vulnerabilities if, for instance, an attacker manages to get an authenticated administrator to click on some forged link. For our purposes, however, we make the assumption that such an area is inaccessible, and hence, focus on the remaining flows instead.

A smaller subset of the flows that we considered non-critical was found in install, migrate, or update scripts which are usually deleted after the process in question is finished (or made inaccessible in some other way). However, if a user is supposed to take such measures manually, and forgets to do so, these flows may—unsurprisingly—also result in exploitable vulnerabilities. Our interest, however, lies more in readily exploitable flaws, so these flaws are not within our scope.

Lastly, several flows were non-critical for a variety of reasons. For instance, programmers globally sanitize arrays such as `$_GET` or `$_POST` before using their values at all. We also observed that many programmers sanitized input by using ad-hoc sanitizers, such as matching them against a whitelist, or casting them to an integer. An analyst interested in a specific project could add such sanitizers to the list of acceptable sanitizers to improve the results.

Command Injection. The results of our traversals for finding command injections are summarized in Table 5.5.

Here it is nice to observe that the ratio of sinks to the total number of calls is much higher in the set $\mathcal{C}$ (i.e., $^{64}/_{270} = 0.24$) than it is in the set $\mathcal{P}$ ($^{19}/_{1598} = 0.012$). Indeed, for web shells in particular, unsanitized flows from input to shell commands are to be expected. This observation confirms that our approach works well to find such flows. In $\mathcal{P}$, we are left with only 19 sinks (originating from 47 flows), of which we confirmed 6 to be vulnerable, yielding a hit rate of $^{6}/_{19} = 0.32$, i.e., 32%. For the others, we find that these flows use the low input as part of a shell command and cast it to an integer or check that it is an integer before executing the command, or check it against a whitelist.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	3s	
Pathfinder traversal	48m 41s	
`eval` statements	5,111	255
Sinks (Flows)	19 (2,404)	115 (147)
Vulnerabilities	5	-

Table 5.6: Evaluation for `eval`.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	5s	
Pathfinder traversal	1d 2h 5m 41s	
`include`, `include_once`, `require`, `require_once` statements	199,169	1,792
Sinks (Flows)	455 (1,292)	50 (100)
Vulnerabilities	100	-

Table 5.7: Evaluation for `include` / `require`.

Code Injection. A large class of vulnerabilities are code injections. Since this can occur by either having control over a string passed to `eval` or over the URL passed to `include` or `require`, we focus on both of these classes in our analysis. We summarize our findings in Tables 5.6 and 5.7. We first discuss the results for `eval`, then turn to `include` and `require`.

For `eval`, as was true for command injection, it is nice to observe yet again that the ratio of sinks to total number of statements is significantly higher in $\mathcal{C}$ ($^{115}/_{255} = 0.6$) than it is in $\mathcal{P}$ ($^{19}/_{5111} = 0.004$). As expected, code injection is much more common in web shells or intentionally vulnerable software than in other projects, confirming once more that our approach works well to find such flows. The indexing query is very efficient in this case (3 seconds), which can be explained by the fact that `eval` is actually a PHP construct that corresponds to a distinguished AST node type: Hence, the database only needs to return all nodes of that particular type, whereas in the case of `mysql_query` for example, the database needs to check a constellation of several AST nodes in order to identify the calls to this function.

There is no universal sanitizer for the `eval` construct. When evaluating input from low sources, allowable input very much depends on the context. Upon inspection, we find that many flows are not vulnerable because an attacker-controllable source first flows into a database request, and then the result of that database request is passed into `eval`. In other cases, whitelists or casts to ints are used. We do, however, find 5 sinks where an attacker can inject code, i.e., exploitable code injection flaws. This yields a hit rate of $^{5}/_{19} = 0.26$.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 18s	
Pathfinder traversal	1h 47m 35s	
`fopen` calls	11,288	949
Sinks (Flows)	265 (667)	357 (1,121)
Vulnerabilities	6	-

Table 5.8: Evaluation for `fopen`.

Lastly, we also investigated the reason for there being so many flows with so few sinks in the case of `eval`: In one of the projects, an `eval` is frequently performed on the results of various processed parts of several database requests. These database queries often use several variables from low sources (properly sanitized). The various combinations of the different sources, the different database requests and the processed parts of the results account for the high number of flows, which eventually flow into only a handful of sinks.

In the case of the PHP language constructs `include` / `require`, there is no universal standard on how to sanitize input variables either. Accordingly, we do find 100 vulnerabilities where an attacker is indeed able to inject strings of their choosing into a filename included by an `include` or `require` statement. However, in the vast majority of these cases, the attacker can only control a part of the string. A fixed prefix hardly hurts an attacker since they may use the string `..` to navigate the directory hierarchy, but a fixed suffix is harder to circumvent: It requires an attacker to be able to upload a file to the server or remote file inclusion to be enabled, as discussed in Section 2.2.3. This is a limitation of the type of attack per se, rather than of our approach.

Arbitrary File Reads/Writes. For vulnerabilities potentially resulting in file content leaks or corruptions, we look at the function call `fopen`, used to access files. We report on our findings in Table 5.8.

Yet again and as expected, we observe that the ratio of sinks to calls is greater in $\mathcal{C}$ ($^{357}/_{949} = 0.38$) than in the set $\mathcal{P}$ ($^{265}/_{11288} = 0.023$): Arbitrary files are much more commonly opened from low input on purpose in $\mathcal{C}$.

As was the case for `include` / `require`, there is no standard sanitizer in this case. Upon inspecting the flows, we again find whitelists, database requests or casts to integers that prevent us from exploiting the flow. Even when an attacker does indeed have some influence on the opened file—unintended by the programmer—this does not necessarily induce a vulnerability: In many cases, the file is opened and processed internally only, without being leaked and with no harm to the program. This explains why we find only 6 vulnerabilities out of a total of 265 sinks.

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	25s	
Pathfinder traversal	5d 7h 57m 8s	
`echo` statements and `print` expressions	946,170	36,077
Sinks (Flows)	15,972 (45,298)	2,788 (5,550)
Sample	726 (852)	-
Vulnerabilities	26	-

Table 5.9: Evaluation for `echo` / `print`.

5.4.2.2 Attacks Targeting the Client

Cross-Site Scripting (XSS). After having discussed attacks which target the server, we now turn to flaws which allow an attack against the client. The results for cross-site scripting are shown in Table 5.9.

At first glance it may seem astounding that there are so many instances of `echo` and `print` nodes in our graph. This, however, is to be expected if we think about the nature of PHP: PHP is a web-based language that focuses on producing HTML output. We also note that, when HTML code is intermixed with PHP code, i.e., when there is code outside of `<?php ... ?>` tags, it is treated like an argument for an `echo` statement by the PHP AST parser. Additionally, the inline echo tags `<?= $var; ?>` also produce `echo` nodes in the AST. Finally, passing several arguments to `echo` as in `echo exrp1, expr2;` produces a distinct `echo` node for each argument. The time taken by the pathfinder traversal is quite high. Indeed, the running time of this traversal grows linearly in the number of nodes it has to process. It averages to 4 minutes and 9 seconds for each of the 1,854 projects.

Since echoing input from the user is a common scenario in PHP, several standard sanitizers exist: We consider `htmlspecialchars`, `htmlentities`, and `strip_tags`. Still, we can observe here that the number of remaining flows in $\mathcal{P}$ is very high (45,298). However, it must also be noted that they result from a set of 1,850 projects, thus averaging to only about 24 flows per project. Hence, inspecting the flows when analyzing a single project appears perfectly feasible; it is clear that the number of flows grows linearly in the number of projects. Yet in our large-scale study, we cannot inspect all of the reports in a reasonable amount of time. Therefore, we sampled 1,000 flows at random, 852 of which fell into the set $\mathcal{P}$ and ended in 726 distinct sinks spread across 116 different projects.

Upon inspection of the sample, we find many uncritical paths that use sanitizers in the form of whitelists or casts to integers. In other cases, even though HTML and JavaScript code could be injected, the PHP script explicitly

	$\mathcal{P}$	$\mathcal{C}$
Indexing query	1m 17s	
Pathfinder traversal	8m 28s	
`setcookie` calls	1,403	403
Sinks (Flows)	158 (507)	63 (95)
Vulnerabilities	1	-

Table 5.10: Evaluation for `setcookie`.

sets the content type of the response, e.g., to `application/json`. This way, browsers are forced to disable their content sniffing and interpret the result as JSON [Zalewski 2011]. In such cases, the HTML parser and the JavaScript engine are not invoked; hence, such a flow cannot be exploited.

Still, we do find 26 exploitable XSS vulnerabilities in 16 different projects, e.g., in the popular software LimeSurvey.[3] By projecting the ratio of 16 vulnerable projects to the reported 116 projects (13.7%), we expect that about 255 of the 1,850 projects are vulnerable to XSS attacks, which validates the fact that XSS vulnerabilities are the most common application-level vulnerability on the Web [WhiteHat 2015]. Hence, the fact that we obtain a high number of flows must also be attributed to the fact that we analyze a very large number of projects and that such vulnerabilities are, indeed, very common. These facts should be kept in mind when considering the large number of reported flows.

Session Fixation. As we discussed in Section 5.2.3, session fixation attacks can be conducted when an attacker can arbitrarily set a cookie for their victim. Therefore, to find such vulnerabilities, we focused on function calls to `setcookie`, the results of which are shown in Table 5.10.

There is no standard sanitizer for `setcookie`. Upon inspecting the flows, we find only one vulnerability among the 158 sinks. This is mainly due to the following fact: In many of these cases, an attacker is indeed able to control the value of the cookie. However, for an exploitable session fixation vulnerability, the attacker needs to control *both* the name and the value of the cookie, an opportunity which turns out not to be very common.

5.5 Discussion and Limitations

The main goal of our evaluation was to evaluate the efficacy and applicability of our approach to a large amount of PHP projects without hand-selecting these projects first, i.e., in a fully automated manner (the entire process of crawling for projects, parsing them, generating code property graphs, importing

[3]We reported this and other bugs to the developers. The vulnerability in LimeSurvey has since been acknowledged and fixed.

them into a graph database and running our traversals requires scant human interaction). To the best of our knowledge, such a large-scale analysis of PHP projects has not been performed before. The final inspection of the reported flows cannot be automated; it requires contextual information and human intelligence to decide whether some flow does indeed lead to an exploitable vulnerability in practice.

In the end, our approach performed better for some types of vulnerabilities than for others. In the case of code injection, we obtained a good hit rate of about 25%, whereas in the case of cross-site scripting, only about 4% of the reported data flows were indeed exploitable. Considering that PHP is a highly dynamic scripting language and the analysis was performed on a large scale, we believe that these numbers are still within reason. As far as efficiency is concerned, the combined computing time was a little under a week for the 1,854 projects. However, the lion's share of the time (over 5 days) was consumed by the traversal looking for cross-site scripting vulnerabilities. This is explained by the fact that flows from low sources to `echo` statements are very common in PHP. All in all, our approach appears to scale well, and it could be further improved by parallelizing the traversals.

We have considered the most widespread and relevant types of vulnerabilities (cf. Chapter 2), but we envision our tool could be used to find more specific types of flaws that we did not consider here, such as magic hashes (see Section 3.1.2). In this case, all code matching the syntactical property of the result of a hash function (`hash`, `md5`, ...) being compared to another value with the == operator could be easily queried from a code property graph database and coupled with other conditions, e.g., that the hashed value depends on a public input, using similar techniques as the ones presented in this chapter. The strength of our approach lies in the expressiveness and flexibility of graph traversals, i.e., users of our framework can use it as needed in the context of a given application.

Clearly, there are also flows which are impossible to discover using static analysis. For instance, we cannot reconstruct the control or data flow yielded by PHP code evaluated within an `eval` construct. Another interesting example is PHP's capability for reflection. Consider for example the code snippet `$a = source(); $b = $$a; sink($b);`: Here, the variable passed into the sink is the variable whose *name* is the same as the *value* of the variable `$a`. Since the value of `$a` cannot be determined statically, but depends on runtime input, this scenario can only be covered by dynamic analysis. To tackle this case with static analysis, we have two options: we can either *over-approximate* or *under-approximate*, i.e., we can either assume that *any* variable which is present in the current context could flow into the sink, or assume that no other variable was written by an adversary. On the one hand, over-approximating will result in a higher number of false positives, i.e., flows will be detected that turn out not to be harmful in practice. On the other hand, under-approximating will result in a higher number of false negatives, meaning that some vulnerable flows will remain undetected. Here, we decided to under-approximate so as to reduce false positives.

Global variables also represent a hard problem: If, during analysis, the input to a security-critical function can be traced back to a global variable, then it is not clear whether this global variable should be considered as tainted, since that depends on what other functions which manipulate the same variable may have executed earlier, or which files manipulating this variable may have included the file containing the code currently analyzed, but this information is usually only available at runtime, i.e., it is statically unknown.

Although we evaluated our tool once on a single crawl of a large amount of GitHub projects, we envision that it could be useful in other scenarios. In particular, it can potentially be useful to companies with large and fast-evolving codebases when run recurrently in order to find newly introduced security holes quickly. Clearly, such a use case could be interesting for Wordpress platforms or online shops. Here, the flexibility and customizability of our tool are particularly effective.

5.6 Related Work

We review the two most closely-related areas of previous research, i.e., the discovery of vulnerabilities in PHP code, and flaw detection based on query languages and graphs.

5.6.1 Discovery of Vulnerabilities in PHP Code

The detection of security vulnerabilities in PHP code has been in the focus of research for over ten years. One of the first works to address the issue of static analysis in the context of PHP was produced by Huang *et al.* [Huang *et al.* 2004a], who presented a lattice-based algorithm derived from type systems and typestate to propagate taint information. Subsequently, they presented another technique based on bounded model checking [Huang *et al.* 2004b] and compared it to their first technique. A significant fraction of PHP files were rejected due to the applied parser (about 8% in their experiments). In contrast, by using PHP's own internal parser, we are inherently able to parse any valid PHP file and will even be able to parse PHP files in the future as new language features are added. If such a language feature alters control flow or re-defines variables, we will be able to parse it, but we will have to slightly correct control flow graph and/or program dependence graph generation to avoid introducing imprecisions.

In 2006, Xie and Aiken [Xie & Aiken 2006] addressed the problem of statically identifying SQL injection vulnerabilities in PHP applications. At the same time, Jovanovic *et al.* presented *Pixy* [Jovanovic *et al.* 2006], a tool for static taint analysis in PHP. Their focus was specifically on cross-site scripting bugs in PHP applications. In total, they analyzed six different open-source PHP projects. In these, they rediscovered 36 known vulnerabilities (with 27 false positives) as well as an additional 15 previously unknown flaws with 16 false positives. Wasserman and Su presented two works focused on statically finding both SQL injections and cross-site scripting [Wassermann & Su 2007, Wassermann & Su 2008]. Additional work in this area has been conducted on the correctness of sanitization routines [Balzarotti *et al.* 2008, Yu *et al.* 2010]. As a follow-up on their work *Pixy*, Jovanovic *et al.* extended their approach to also cover SQL injections [Jovanovic *et al.* 2010]. While all these tools were pioneers in the domain of automated discovery of vulnerabilities in PHP applications, they focused on very specific types of flaws only, namely, cross-site scripting and SQL injections. In this work, we cover a much wider array of different kinds of vulnerabilities.

Most recently, Dahse and Holz [Dahse & Holz 2014a] presented *RIPS*, which covers a similar range of vulnerabilities as we do in this work. RIPS builds control flow graphs and then creates block and function summaries by simulating the data flow for each basic block, which allows to conduct a precise taint analysis. In doing so, the authors discovered previously unknown flaws in osCommerce, HotCRP, and phpBB2. Compared to our work, they only evaluated their tool on a handful of selected applications, but did not conduct a large-scale analysis. Since RIPS uses a type of symbolic execution to build block and function summaries, it is unclear how well it would scale to large quantities of code. Instead of symbolic execution, we efficiently build program dependence graphs to conduct taint analysis; to the best of our knowledge, we are the first to actually build program dependence graphs for PHP. Moreover, RIPS lacks the flexibility and the programmability of our graph traversals: It is able to detect a hard-coded, pre-defined set of vulnerabilities. In contrast, our tool is a framework which allows developers to program their own traversals. It can be used to model various types of vulnerabilities, in a generic way (as we demonstrate in this work) or geared towards a specific application.

Dahse and Holz followed up on their work by detecting second-order vulnerabilities, e.g., persistent cross-site scripting, identifying more than 150 vulnerabilities in six different applications [Dahse & Holz 2014b]. Follow-up work inspired by them was presented in 2015, when Olivo *et al.* [Olivo *et al.* 2015] discussed a static analysis of second-order denial-of-service vulnerabilities.

They analyzed six applications, which partially overlap with the ones analyzed by previous work, and found 37 vulnerabilities, accompanied by 18 false positives. These works can be considered as orthogonal to ours.

In summary, while there has been a significant amount of research on the subject of static analysis for PHP, these works focused on a small set of (the same) applications. In contrast, our work is not aimed towards analyzing a single application in great detail. Instead, our goal was to implement an approach which would scale well to scanning large quantities of code and would be flexible enough to add support for additional vulnerability types with minimal effort. Unfortunately, a direct comparison of results between our tool and other tools is difficult, due to the fact that we do not usually have access to the implemented prototypes on the one hand, and the limited detail of the reports on the other. This difficulty has also been noticed by other authors [Jovanovic *et al.* 2010, Dahse & Holz 2014a]. Usually, only the number of detected vulnerabilities is reported, but not the vulnerabilities as such. Even comparing the numbers is not straightforward, as there is no universally agreed-upon standard on how vulnerabilities should be counted. For instance, when there exist several vulnerable data flows into the same security-critical function call, it is not clear whether each flow should be counted as a vulnerability, or whether it should count as a single vulnerability, or anything in-between (e.g., depending on the similarity of the different flows). In this work, we explained precisely how we counted vulnerabilities, and we make our tool publicly available on GitHub both for researchers and developers.

5.6.2 Flaw Detection Using Query Languages and Graphs

Our work uses queries for graph databases to describe vulnerable program paths, an approach closely related to defect detection via query languages, as well as static program analysis using graph-based program representations.

The concept of using query languages to detect security and other bugs has been considered by several researchers in the past [Paul & Prakash 1994, Hallem *et al.* 2002, Lam *et al.* 2005, Martin *et al.* 2005, Goldsmith *et al.* 2005]. In particular, Martin *et al.* [Martin *et al.* 2005] proposed the *Program Query Language* (PQL), an intermediary representation of programs. With this representation, they are able to identify violations of design rules, to discover functional flaws and security vulnerabilities in a program. Livshits and Lam [Livshits & Lam 2005] used PQL to describe typical instances of SQL injections and cross-site scripting in Java programs, and successfully identified 29 flaws in nine popular open-source applications.

Graph-based program analysis has a long history, ranging back to the seminal work by Reps [Reps 1998] on program analysis via graph reachability, and the introduction of the program dependence graph by Ferrante *et al.* [Ferrante *et al.* 1987]. Following along this line of research, Kinloch and Munro [Kinloch & Munro 1994] present the Combined C Graph, a data structure specifically designed to aid in graph-based defect discovery, while Yamaguchi *et al.* [Yamaguchi *et al.* 2014] present the code property graph for vulnerability discovery. Their work, which inspired the work presented in this chapter, first employed a graph representation of code properties to detect vulnerabilities in C code. The work presented here notably extends their work by first demonstrating that similar techniques can be employed to identify vulnerabilities in high-level, dynamic scripting languages, making it applicable for the identification of vulnerabilities in web applications, and second by adding call graphs, allowing for interprocedural analysis.

Their idea was picked up by Alrabaee *et al.* [Alrabaee *et al.* 2015], who use a graph representation to detect code reuse. Their specific goal in this is to ease the task of reverse engineers when analyzing unknown binaries. The concept of using program dependence graphs is also used by Johnson *et al.* [Johnson *et al.* 2015], who built their tool PIDGIN for Java. Specifically, they create the graphs and run queries on them, in order to check security guarantees of programs, enforce security during development, and create policies based on known flaws. Besides this more specific use in finding flaws, several works have looked at PDGs for information-flow control, such as [Hammer 2009, Graf 2010, Snelting *et al.* 2014].

5.7 Summary

Given the pervasive presence of PHP as a web programming language, our aim was to develop a flexible and scalable analysis tool to detect and report potential vulnerabilities in a large set of web applications. To this end, we built code property graphs, i.e., a combination of syntax trees, control flow graphs, program dependence graphs, and call graphs for PHP, and demonstrated that they work well to identify vulnerabilities in high-level, dynamic scripting languages. This implies that the approach is well suited for the analysis of a large number of web applications.

We modeled several typical kinds of vulnerabilities arising from exploitable flows in PHP applications as traversals on these graphs. We crawled 1,854 popular PHP projects on GitHub, built code property graphs representing those projects, and showed the efficacy and scalability of our approach by running our flow-finding traversals on this large dataset. We were able to observe that the number of reported flows in a small selected subset of these projects, consisting of purposefully vulnerable software, was tremendously higher than in the other projects, thus confirming that our approach works well to detect such flows. Additionally, we also discovered well over a hundred unintended vulnerabilities in the other projects.

We demonstrated that it is possible to find vulnerabilities in PHP applications on a large scale in a reasonable amount of time. Our code property graphs lay the foundation to build many more sophisticated traversals to find other classes of vulnerabilities by writing appropriate graph traversals, be they generic or specific to an application. The framework presented in this chapter is publicly available to give that possibility to researchers and developers alike.

Chapter 6
Conclusion

Today, we naturally use web applications on a daily basis: JavaScript fuels the client-side logic of almost every website, and PHP is the most prevalent server-side programming language. Moreover, the number of web applications is constantly growing. Since human error in the development of applications can never be wholly avoided, providing developers and security experts with viable means to efficiently and effectively analyze web applications in a machine-assisted manner is a vital area of research.

Static analysis techniques have been investigated by scientific literature for a long time, and they have been successfully employed to analyze (in a security context) a number of languages that lend themselves well to static methods, such as Java or C [e.g., Hammer 2009, Yamaguchi 2015]. Yet static analysis techniques have not been thoroughly studied in the context of more dynamic languages extensively used in web applications such as those investigated in this thesis; the body of literature concerned with dynamic analysis techniques (as opposed to static ones), particularly in the case of JavaScript, is more voluminous. While dynamic analysis techniques have a number of advantages, particularly access to runtime information accompanied by a higher precision in detecting vulnerabilities, they also exhibit disadvantages. In particular, they are less efficient since they require that the program be actually run (or executed symbolically) at the same time, and covering all paths through a program is a hard problem. Hence, while static analysis suffers from a higher number of false positives, it is a more lightweight and scalable technique that can find vulnerabilities which may remain overseen by dynamic analysis.

In this thesis, we investigated the effectiveness of static analysis for two of the core languages for web applications, namely, JavaScript and PHP. As can be expected, using static analysis for such dynamic languages is a highly challenging task.

For JavaScript, the fact that the language is highly dynamic is not the only hurdle: The rich environment in which JavaScript code is typically executed and the commonplace heavy usage of highly complex third-party libraries encumber static analysis significantly. In Chapter 4, we overcame these challenges by applying a series of program transformations and leveraging WALA to build a unified model of the HTML DOM and JavaScript contained in a website as a single large JavaScript program. Using backwards slicing on the program dependence graph of this program, we were able to identify two previously unknown and severe vulnerabilities in the popular Helios voting client, which, despite having been the focus of years of thorough study and investigation in scientific literature, had remained unnoticed thus far.

In Chapter 5, we built a framework for analyzing PHP applications by leveraging the recently proposed concept of code property graphs. These graphs constitute a joint representation of a program that incorporate relevant information for performing vulnerability discovery, namely, its syntax, its control flow, and its control and data dependencies. The clear separation between the generation of these graphs on the one hand and the analysis run on these graphs on the other makes it easy for developers and analysts to use our framework to perform their own analyses. Indeed, the generation of these graphs is fully automated: Our framework takes a PHP project as input and outputs a graph database suitable to be loaded into the popular graph database system Neo4J (other graph database back ends can be easily supported by implementing appropriate output modules). Vulnerabilities can then be modeled as graph traversals using standard graph database query languages such as Cypher or Gremlin. We modeled the most common vulnerabilities in PHP applications using these languages and used these models to run the largest case study (to the best of our knowledge) of vulnerability discovery in PHP applications to this day. We found several hundred vulnerabilities and demonstrated the feasibility and effectiveness or our approach.

Both these contributions show that static analysis aimed at vulnerability discovery in web applications written in highly dynamic languages is feasible. However, there are also inherent limitations that are hard or even impossible to overcome with static analysis techniques. Most prominently, constructs such as *eval* and variants which allow to dynamically evaluate the contents of strings as code cannot be reliably approximated in the general case. For our case studies, the Helios voting client itself fortunately did not make use of these constructs. Although third-party libraries used by Helios did use it, we were able to get rid of these libraries using a series of code transformations. While this allowed us to emulate the libraries accurately enough for our analysis, it required human work and insight to implement the transformations. Albeit the code transformations can be automated once the replacement APIs have been written, writing them in the first place must be done manually, and doing so for the large number of existing libraries seems like a daunting task. As far as our PHP framework is concerned, as we discussed, we chose to under-approximate, that is, we only treat the variables which directly flow into an *eval* as used by that expression, and we do not treat *eval* expressions as defining any variables. This approach reflects our desire to keep the number of false positives as low as possible, but also implies that we may miss critical information flows. For our large scale study, we additionally explicitly considered any input that flows into *eval* and its variants as suspicious and investigated the corresponding reports manually. The fact that code evaluated in *eval* expressions cannot be reliably approximated also means that a developer could purposefully hide a vulnerability from our analysis, e.g., by writing a backdoor as text and then evaluating it dynamically under certain circumstances.

An interesting avenue of research to cope with such issues inherent to static analysis may be the combination of static and dynamic analysis techniques. In particular for JavaScript, where a considerable existing body of work investigates dynamic analysis techniques for the purpose of vulnerability discovery, it may be possible to get the best of both worlds, i.e., efficiency and precision: One could use static analysis to perform a lightweight analysis first to discover suspicious paths, then use dynamic analysis techniques to improve the precision of the results by focusing on these paths. However, doing so is certainly a non-trivial problem that will require further research and deeper insights.

In summary, research in vulnerability discovery for web applications is clearly an important field, as these applications have gained a tremendous importance in our everyday lives over the past two decades; and this trend continues. Yet it is a hard problem, particularly considering the nature of the languages that have been widely adopted to write such applications in practice. While JavaScript and PHP have many features that make them comfortable and easy to use for application developers, they are notably difficult to analyze using automated techniques. Still, such techniques are needed in order to help developers and security experts cope with with the increasing amount and complexity of code circulating on the Web. In this thesis, we demonstrated that such analysis is feasible and practicable. We look forward to future work in the area, which provides ample room for further fascinating and meaningful research.

Appendix A

Refactoring JavaScript Libraries for the Helios Voting Booth

A.1 Canonical Refactorings

Elements in a document are accessed via a call to jQuery's `$()` function. For example, one can access an element with id `foo`, or a set of elements with class `bar`, like so:

```
$('#foo');
$('.bar');
```

Each of these calls creates a jQuery wrapper object that is associated with the matching element(s), and provides a myriad of functions to manipulate these elements. Calls to these functions are typically chained after the initial call to `$()`. JavaScript provides native equivalents that return native JavaScript objects:

```
document.getElementById('foo');
document.getElementsByClassName('bar');
```

Since these native JavaScript objects do not have the same functions as the jQuery objects, we need to fix these calls as well. In the following, `$(elem)` refers to some jQuery object, and `elem` refers to a native object.

For instance, Helios uses some jQuery functions to manipulate the visibility of elements, manipulate an element's class list or its attributes, retrieve or manipulate an element's inner HTML code, or select (highlight) it in the browser:

```
$(elem).show();
$(elem).hide();
$(elem).addClass('someclass');
$(elem).removeClass('someclass');
$(elem).attr('somename', 'somevalue');
$(elem).html('some html code');
$(elem).select();
```

All of these have native equivalents, so replacing these calls is straightforward:

```
elem.style.display='';
elem.style.display='none';
elem.classList.add('someclass');
elem.classList.remove('someclass');
elem.setAttribute('somename', 'somevalue');
elem.innerHTML = 'some html code';
elem.select();
```

A jQuery object may be associated with a set of elements, and in this case a chained function (usually) implicitly affects all matching elements. In native JavaScript, we have to use an explicit iteration.

jQuery also implements an event handler that is triggered as soon as a page's DOM is fully constructed (but before, e.g., images are fully loaded). The code inside the following function corresponds to the Helios client's entry point:

```
$(document).ready(
  function() {
    // code executed once the DOM is ready
  });
```

Luckily, modern browsers implement a native equivalent:

```
document.addEventListener('DOMContentLoaded',
  function() {
    // code executed once the DOM is ready
  });
```

Another jQuery trick used by Helios is to escape an HTML string as follows:

```
return $('<div/>').text('<i>Hi!</i>').html();
// Outputs: &lt;i&gt;Hi!&lt;/i&gt;
```

This creates a dummy element, then uses jQuery's `.text()` function to set that element's text contents (meaning that HTML is escaped), and finally retrieves the resulting element's contents. There are various ways to HTML-escape a string in JavaScript, but to faithfully model the behavior of the Helios client, the following code comes closest:

```
var dummyDiv = document.createElement('div'),
    dummyText = document.createTextNode('<i>Hi!</i>');
dummyDiv.appendChild( dummyText);
return dummyDiv.innerHTML;
```

Indeed, jQuery's `.text()` method internally uses the native `createTextNode()` function to create a text node, whose contents are implicitly escaped by the browser.

jQuery can not only create wrapper objects on top of new HTML elements as in the last example, but even on top of plain JavaScript objects, such as arrays. Helios uses this feature to enable some of jQuery's utility functions, e.g., an iterator:

```
$(['1','2','3']).each(
  function( index, value) {
    // function executed once per array element
  }
```

Since ECMAScript 5, the `Array` prototype implements its own native iterator:

```
['1','2','3'].forEach(
  function( value, index) {
    // function executed once per array element
  });
```

Another utility provided by jQuery is to search for an element in an array, and return the index of the last matching element:

```
var index = $(arr).index(value);
```

Note that in more recent jQuery versions, `.index()` returns the *first* matching element instead, but jQuery 1.2.2, used by Helios, returns the last one. While there is no native JavaScript equivalent for this function, its behavior is trivial to re-implement:

```
function arr_index( arr, value) {
  var res = -1;
  for( var i = 0; i < arr.length; i++) {
    if( arr[i] == value)
      res = i;
  }
  return res;
}
var index = arr_index( arr, value);
```

A.2 Modeling jQuery Ajax Functions

A.2.1 $.get

jQuery's `$.get` function sends a GET request to a specified URL and calls a success handler function upon success.

Original code:

```
$.get( url,
  function( response) {
    /* process answer */
  });
```

Transformed code:

```
var request = new XMLHttpRequest();
request.open( 'GET', url, true);
request.onload =
  function() {
    if( request.status >= 200 && request.status < 400) {
      var response = request.responseText;
      /* process answer */
    }
  };
request.send();
```

A.2.2 $.getJSON

jQuery's `$.getJSON` function sends a GET request to a specified URL. Upon success, it parses the answer as a JSON string and calls a success handler function.
Original code:

```
$.getJSON( url,
  function( response) {
    /* process answer */
  });
```

Transformed code:

```
var request = new XMLHttpRequest();
request.open( 'GET', url, true);
request.onload =
  function() {
    if( request.status >= 200 && request.status < 400) {
      var response = JSON.parse(request.responseText);
      /* process answer */
    }
  };
request.send();
```

A.2.3 $.post

jQuery's `$.post` function sends a POST request to a specified URL, along with some data to be processed by the server. The data to be submitted is encoded as a URL query string by jQuery using its internal `$.param` function, which we consequently included to obtain a faithful model. Upon success, a success handler function is called.

Original code:

```
$.post( url, data,
  function( response) {
    /* process answer */
  });
```

Transformed code:

```
function serialize_to_query_string(data) {
  var s = [];
  if( data.constructor == Array)
    data.forEach( function( value, index) {
      s.push( encodeURIComponent( index) + "=" + encodeURIComponent( value) );
    });
  else
    for( var j in data)
      if( data[j] && data[j].constructor == Array)
        data[j].forEach( function( value, index) {
          s.push( encodeURIComponent( j) + "=" + encodeURIComponent( value));
        });
      else
        s.push( encodeURIComponent( j) + "=" + encodeURIComponent( data[j]));
  return s.join("&").replace( /%20/g, "+");
}

var request = new XMLHttpRequest();
request.open( 'POST', url, true);
request.setRequestHeader( 'Content-Type', 'application/x-www-form-urlencoded; charset=UTF-8');
request.onload =
  function() {
    if( request.status >= 200 && request.status < 400) {
```

```
      var response = request.responseText;
      /* process answer */
    }
  };
request.send( serialize_to_query_string(data));
```

A.3 Query Object Plugin

Given a URL

`http://example.com/?apples=1&bananas=2`

the following jQuery call will return the string '2':

```
$.query.get('bananas');
```

There is no native JavaScript equivalent, but it is easy to write one using a regular expression:

```
function getParameterByName( name) {
  var match = RegExp('[?&]' + name + '=([^&]*)').exec(location.search);
  return match ? decodeURIComponent(match[1].replace(/\+/g,' ')) : "";
}
getParameterByName('bananas');
```

A.4 Modeling the jQuery Templating Plugin

Various templates are used in the Helios voting booth, as explained in Section 4.3.1.2. These templates implement their own logic which is interpreted by the template plugin. We translate these templates to pure HTML and JavaScript as follows.

A.4.1 The Header Template

Original template code:

```
<h1 id="election_name">{$T.election.name}</h1>
<p> </p>
```

Modified template code:

```
<h1 id="election_name"></h1>
<p> </p>
```

Custom header template processing function:

```
function processTemplate_header() {
  // process template
  document.getElementById('election_name').textContent = BOOTH.election.name;
}
```

A.4.2 The Question Template

Original template code:

```
<form onsubmit="return false;" class="prettyform" id="answer_form">
<input type="hidden" name="question_num" value="{$T.question_num}" />

<p>
<br />
<b>{$T.question.question}</b>
<br />
<span style="font-size: 0.6em;">#{$T.question_num + 1} of {$T.last_question_num + 1} &mdash;
 vote for
{#if $T.question.min && $T.question.min > 0}
{#if $T.question.max}
{$T.question.min} to {$T.question.max}
{#else}
at least {$T.question.min}
{#/if}
{#else}
{#if $T.question.max}
{#if $T.question.max > 1}up to {#/if}{$T.question.max}
{#else}
as many as you approve of
{#/if}
{#/if}
</span>
</p>

{#foreach $T.question.answers as answer}
<div id="answer_label_{$T.question_num}_{$T.answer_ordering[$T.answer$index]}"><input type="checkbox" class="ballot_answer" id="answer_{$T.question_num}_{$T.answer_ordering[$T.answer$index]}" name="answer_{$T.question_num}_{$T.answer_ordering[$T.answer$index]}" value="yes" onclick="BOOTH.click_checkbox({$T.question_num}, {$T.answer_ordering[$T.answer$index]}, this.checked);" /> {$T.question.answers[$T.answer_ordering[$T.answer$index]]}

{#if $T.question.answer_urls && $T.question.answer_urls[$T.answer_ordering[$T.answer$index]] && $T.question.answer_urls[$T.answer_ordering[$T.answer$index]] != ""}
  
<span style="font-size: 12pt;">
[<a target="_blank" href="{$T.question.answer_urls[$T.answer_ordering[$T.answer$index]]}">more info</a>]
</span>
{#/if}
</div>
{#/for}

<div id="warning_box" style="color: green; text-align:center; font-size: 0.8em; padding-top:10px; padding-bottom: 10px; height:50px;">
</div>


{#if $T.show_reviewall}
```

```
<div style="float: right;">
<input type="button" onclick="BOOTH.validate_and_confirm({$T.question_num});"
    value="Proceed" />
</div>
{#/if}

{#if $T.question_num != 0}
<input type="button" onclick="BOOTH.previous({$T.question_num})" value="Previous" />
 
{#/if}

{#if $T.question_num < $T.last_question_num}
<input type="button" onclick="BOOTH.next({$T.question_num})" value="Next" />
 
{#/if}

<br clear="both" />

</form>
```

Modified template code:

```
<form onsubmit="return false;" class="prettyform" id="answer_form">
<input id="question_num" type="hidden" name="question_num" value="" />

<p>
<br />
<b id="question_question"></b>
<br />
<span id="question_subtext" style="font-size: 0.6em;">
</span>
</p>

<div id="answer_labels">
</div>

<div id="warning_box" style="color: green; text-align:center; font-size: 0.8em;
    padding-top:10px; padding-bottom: 10px; height:50px;">
</div>

<div id="question_proceed_div" style="float: right; display: none;">
<input id="question_proceed_button" type="button" value="Proceed" />
</div>

<input id="question_previous_button" type="button" value="Previous" style="display: none;"
    />
 

<input id="question_next_button" type="button" value="Next" style="display: none;" />
 

<br clear="both" />

</form>
```

Custom question template processing function:

```
function processTemplate_question(question_num) {
  // process template
  document.getElementById('question_num').value = question_num;
  document.getElementById('question_question').textContent =
    BOOTH.election.questions[question_num].question;
```

```

var subtext = (question_num + 1) + " of " + BOOTH.election.questions.length + " &mdash; vote for ";
if( BOOTH.election.questions[question_num].min && BOOTH.election.questions[question_num].min > 0) {
  if( BOOTH.election.questions[question_num].max) {
    subtext += BOOTH.election.questions[question_num].min + " to " + BOOTH.election.questions[question_num].max;
  }
  else {
    subtext += "at least " + BOOTH.election.questions[question_num].min;
  }
}
else {
  if( BOOTH.election.questions[question_num].max) {
    if( BOOTH.election.questions[question_num].max > 1) {
      subtext += "up to ";
    }
    subtext += BOOTH.election.questions[question_num].max;
  }
  else {
    subtext += "as many as you approve of";
  }
}
document.getElementById('question_subtext').innerHTML = subtext;

document.getElementById( 'answer_labels').innerHTML = "";
BOOTH.election.questions[question_num].answers.forEach( function( value, index) {
  var div = document.createElement('div');
  div.id = "answer_label_" + question_num + "_" + BOOTH.election.question_answer_orderings[question_num][index];

  var input = document.createElement('input');
  input.type = "checkbox";
  input.classList.add("ballot_answer");
  input.id = "answer_" + question_num + "_" + BOOTH.election.question_answer_orderings[question_num][index];
  input.name = "answer_" + question_num + "_" + BOOTH.election.question_answer_orderings[question_num][index];
  input.value = "yes";
  input.onclick = function() { BOOTH.click_checkbox( question_num, BOOTH.election.question_answer_orderings[question_num][index], this.checked); };

  var answer = document.createTextNode( BOOTH.election.questions[question_num].answers[BOOTH.election.question_answer_orderings[question_num][index]]);

  div.appendChild( input);
  div.appendChild( answer);

  if( BOOTH.election.questions[question_num].answer_urls &&
      BOOTH.election.questions[question_num].answer_urls[BOOTH.election.question_answer_orderings[question_num][index]] &&
      BOOTH.election.questions[question_num].answer_urls[BOOTH.election.question_answer_orderings[question_num][index]] != "") {
    var nbsp = document.createTextNode("  ");
    var span = document.createElement('span');
    span.style.fontSize = "12pt";
    var lbrack = document.createTextNode("[");
    var a = document.createElement('a');
    a.target = "_blank";
    a.href = BOOTH.election.questions[question_num].answer_urls[BOOTH.election.question_answer_orderings[question_num][index]];
```

```
      a.textContent = "more info";
      var rbrack = document.createTextNode("]");
      span.appendChild( lbrack);
      span.appendChild( a);
      span.appendChild( rbrack);
      div.appendChild( nbsp);
      div.appendChild( span);
    }

    document.getElementById( 'answer_labels').appendChild( div);
  });

  if( BOOTH.all_questions_seen) {
    document.getElementById('question_proceed_div').style.display = "";
    document.getElementById('question_proceed_button').onclick = function() { BOOTH.validate_and_confirm( question_num); };
  }

  if( question_num != 0) {
    document.getElementById('question_previous_button').style.display = "";
    document.getElementById('question_previous_button').onclick = function() { BOOTH.previous( question_num); };
  }
  else {
    document.getElementById('question_previous_button').style.display = "none";
  }

  if( question_num < BOOTH.election.questions.length - 1) {
    document.getElementById('question_next_button').style.display = "";
    document.getElementById('question_next_button').onclick = function() { BOOTH.next( question_num); };
  }
  else {
    document.getElementById('question_next_button').style.display = "none";
  }
}
```

A.4.3 The Seal Template

Original template code:

```
{#if $T.election_metadata.use_advanced_audit_features}
<div style="float: right; background: lightyellow; margin-left: 20px; padding: 0px 10px 10px 10px; border: 1px solid #ddd; width:200px;">
<h4><a onclick="$('#auditbody').slideToggle(250);" href="#">Audit</a> <span style="font-size: 0.8em; color: #444">[optional]</span></h4>
<div id="auditbody" style="display:none;">
<p>
If you choose, you can audit your ballot and reveal how your choices were encrypted.
</p>
<p>
You will then be guided to re-encrypt your choices for final casting.
</p>
<input type="button" value="Verify Encryption" onclick="BOOTH.audit_ballot();" class="pretty" />
</p>
</div>
</div>
{#/if}

```

```
<h3>Review your Ballot</h3>


<div style="padding: 10px; margin-bottom: 10px; background-color: #eee; border: 1px #ddd solid; max-width: 340px;">
{#foreach $T.questions as question}

<b>Question #{$T.question$index + 1}: {$T.question.short_name}</b><br>
{#if $T.choices[$T.question$index].length == 0}
<div style="margin-left: 15px;">&#x2610; <i>No choice selected</i></div>
{#/if}
{#foreach $T.choices[$T.question$index] as choice}
<div style="margin-left: 15px;">&#x2713; {$T.choice}</div>
{#/for}
{#if $T.choices[$T.question$index].length < $T.question.max}
[you under-voted: you may select up to {$T.question.max}]
{#/if}
[<a onclick="BOOTH.show_question({$T.question$index}); return false;" href="#">edit responses</a>]
{#if !$T.question$last}<br><br>{#/if}
{#/for}
</div>


<p><p>Your ballot tracker is <b><tt style="font-size: 11pt;">{$T.encrypted_vote_hash}</tt></b>, and you can <a onclick="BOOTH.show_receipt(); return false;" href="#">print</a> it.<br /><br /></p>

<p>
Once you click "Submit", the unencrypted version of your ballot will be destroyed, and only the encrypted version will remain.  The encrypted version will be submitted to the Helios server.</p>

<button id="proceed_button" onclick="BOOTH.cast_ballot();">Submit this Vote!</button><br />
<div id="loading_div"><img src="loading.gif" id="proceed_loading_img" /></div>



<form method="POST" action="{$T.cast_url}" id="send_ballot_form" class="prettyform">
<input type="hidden" name="election_uuid" value="{$T.election_uuid}" />
<input type="hidden" name="election_hash" value="{$T.election_hash}" />
<textarea name="encrypted_vote" style="display: none;">
{$T.encrypted_vote_json}
</textarea>
</form>
```

Modified template code:

```
<div id="auditbox" style="float: right; background: lightyellow; margin-left: 20px; padding: 0px 10px 10px 10px; border: 1px solid #ddd; width:200px; display: none;">
<h4><a onclick="document.getElementById('auditbody').style.display='';" href="#">Audit</a> <span style="font-size: 0.8em; color: #444">[optional]</span></h4>
<div id="auditbody" style="display:none;">
<p>
If you choose, you can audit your ballot and reveal how your choices were encrypted.
</p>
<p>
You will then be guided to re-encrypt your choices for final casting.
</p>
<input type="button" value="Verify Encryption" onclick="BOOTH.audit_ballot();" class="pretty" />
</p>
```

```
</div>
</div>

<h3>Review your Ballot</h3>


<div id="reviewbox" style="padding: 10px; margin-bottom: 10px; background-color: #eee; border: 1px #ddd solid; max-width: 340px;">
</div>


<p><p>Your ballot tracker is <b><tt id="seal_div_vote_hash" style="font-size: 11pt;"></tt></b>, and you can <a onclick="BOOTH.show_receipt(); return false;" href="#">print</a> it.<br /><br /></p>

<p>
Once you click "Submit", the unencrypted version of your ballot will be destroyed, and only the encrypted version will remain.  The encrypted version will be submitted to the Helios server.</p>

<button id="proceed_button" onclick="BOOTH.cast_ballot();">Submit this Vote!</button><br />
<div id="loading_div"><img src="loading.gif" id="proceed_loading_img" /></div>



<form method="POST" action="" id="send_ballot_form" class="prettyform">
<input type="hidden" id="send_ballot_form_election_uuid" name="election_uuid" value="" />
<input type="hidden" id="send_ballot_form_election_hash" name="election_hash" value="" />
<textarea id="send_ballot_form_encrypted_vote" name="encrypted_vote" style="display: none;">
</textarea>
</form>
```

Custom seal template processing function:

```
function processTemplate_seal() {
  // process template
  if( BOOTH.election_metadata.use_advanced_audit_features) {
    document.getElementById('auditbox').style.display = "";
  }

  document.getElementById( 'reviewbox').innerHTML = "";
  BOOTH.election.questions.forEach( function( question, qindex) {
    var b = document.createElement('b');
    b.textContent = "Question #" + (qindex + 1) + ": " + question.short_name;
    document.getElementById( 'reviewbox').appendChild( b);
    document.getElementById( 'reviewbox').appendChild( document.createElement('br'));

    var choices = BALLOT.pretty_choices(BOOTH.election, BOOTH.ballot);
    if( choices[qindex].length == 0) {
      var div = document.createElement('div');
      div.style.marginLeft = "15px";
      var text = document.createTextNode("\u2610 ");
      var i = document.createElement('i');
      i.textContent = "No choice selected";
      div.appendChild( text);
      div.appendChild( i);
      document.getElementById( 'reviewbox').appendChild( div);
    }
    choices[qindex].forEach( function( choice, cindex) {
      var div = document.createElement('div');
      div.style.marginLeft = "15px";
      var text = document.createTextNode("\u2713 " + choice);
```

```
      div.appendChild( text);
      document.getElementById( 'reviewbox').appendChild( div);
    });
    if( choices[qindex].length < question.max) {
      var text = document.createTextNode( "[you under-voted: you may select up to " + question.max + "]");
      document.getElementById( 'reviewbox').appendChild( text);
    }

    var lbrack = document.createTextNode("[");
    var a = document.createElement('a');
    a.href = "#";
    a.onclick = function() { BOOTH.show_question(qindex); return false; };
    a.textContent = "edit responses";
    var rbrack = document.createTextNode("]");
    document.getElementById( 'reviewbox').appendChild( lbrack);
    document.getElementById( 'reviewbox').appendChild( a);
    document.getElementById( 'reviewbox').appendChild( rbrack);

    // not last iteration?
    if( qindex < BOOTH.election.questions.length - 1) {
      document.getElementById( 'reviewbox').appendChild( document.createElement('br'));
      document.getElementById( 'reviewbox').appendChild( document.createElement('br'));
    }
  });

  document.getElementById('seal_div_vote_hash').textContent = BOOTH.encrypted_ballot_hash;
  document.getElementById('send_ballot_form').action = BOOTH.election.cast_url;
  document.getElementById('send_ballot_form_election_uuid').value = BOOTH.election.uuid;
  document.getElementById('send_ballot_form_election_hash').value = BOOTH.election_hash;
  document.getElementById('send_ballot_form_encrypted_vote').value = BOOTH.encrypted_vote_json;
}
```

A.4.4 The Audit Template

Original template code:

```
<h3>Your audited ballot</h3>

<p>
<b><u>IMPORTANT</u></b>: this ballot, now that it has been audited, <em>will not be tallied</em>.<br />
To cast a ballot, you must click the "Back to Voting" button below, re-encrypt it, and choose "cast" instead of "audit."
</p>

<p>
<b>Why?</b> Helios prevents you from auditing and casting the same ballot to provide you with some protection against coercion.
</p>

<p>
<b>Now what?</b> <a onclick="$('#audit_trail').select(); return false;" href="#">Select your ballot audit info</a>, copy it to your clipboard, then use the <a target="_blank" href="single-ballot-verify.html?election_url={$T.election_url}">ballot verifier</a> to verify it.<br />
Once you are satisfied, click the "back to voting" button to re-encrypt and cast your ballot.
</p>
```

```

<form action="#">
<textarea name="audit_trail" id="audit_trail" cols="80" rows="10" wrap="soft">
{$T.audit_trail}
</textarea>
<br /><br />
Before going back to voting,<br />
you can post this audited ballot to the Helios tracking center so that others might double-check the verification of this ballot.
<br /><br />
<b>Even if you post your audited ballot, you must go back to voting and choose "cast" if you want your vote to count.</b>
<br /><br />
<input type="button" value="back to voting" onclick="BOOTH.reset_ciphertexts();BOOTH.seal_ballot();" class="pretty" />
    
<input type="button" value="post audited ballot to tracking center" onclick="$(this).attr('disabled', 'disabled');BOOTH.post_audited_ballot();" id="post_audited_ballot_button" class="pretty" style="font-size:0.8em;"/>

</form>
```

Modified template code:

```
<h3>Your audited ballot</h3>

<p>
<b><u>IMPORTANT</u></b>: this ballot, now that it has been audited, <em>will not be tallied</em>.<br />
To cast a ballot, you must click the "Back to Voting" button below, re-encrypt it, and choose "cast" instead of "audit."
</p>

<p>
<b>Why?</b> Helios prevents you from auditing and casting the same ballot to provide you with some protection against coercion.
</p>

<p>
<b>Now what?</b> <a onclick="document.getElementById('audit_trail').select(); return false;" href="#">Select your ballot audit info</a>, copy it to your clipboard, then use the <a id="single_ballot_verify" target="_blank" href="">ballot verifier</a> to verify it.<br />
Once you are satisfied, click the "back to voting" button to re-encrypt and cast your ballot.
</p>

<form action="#">
<textarea name="audit_trail" id="audit_trail" cols="80" rows="10" wrap="soft">
</textarea>
<br /><br />
Before going back to voting,<br />
you can post this audited ballot to the Helios tracking center so that others might double-check the verification of this ballot.
<br /><br />
<b>Even if you post your audited ballot, you must go back to voting and choose "cast" if you want your vote to count.</b>
<br /><br />
<input type="button" value="back to voting" onclick="BOOTH.reset_ciphertexts();BOOTH.seal_ballot();" class="pretty" />
    
```

```
<input type="button" value="post audited ballot to tracking center" onclick="this.setAttribute('disabled', 'disabled');BOOTH.post_audited_ballot();" id="post_audited_ballot_button" class="pretty" style="font-size:0.8em;"/>

</form>
```

Custom audit template processing function:

```
function processTemplate_audit() {
  // process template
  document.getElementById('single_ballot_verify').href = "single-ballot-verify.html?election_url=" + BOOTH.election_url;
  document.getElementById('audit_trail').value = BOOTH.audit_trail;

  // re-enable button in case it was disabled earlier; before, this
  // was sort of implicit when the jQuery template got re-processed,
  // since this completely re-generated the contents of #seal_div
  document.getElementById('post_audited_ballot_button').removeAttribute('disabled');
}
```

A.4.5 The Footer Template

Original template code:

```
<span style="float:right; padding-right:20px;">
<a target="_new" href="mailto:{$T.election_metadata.help_email}?subject=help%20with%20election%20{$T.election.name}&body=I%20need%20help%20with%20election%20{$T.election.uuid}">help!</a>
</span>
{#if $T.election.BOGUS_P}
The public key for this election is not yet ready. This election is in preview mode only.
{#else}
Election Fingerprint: <span id="election_hash" style="font-weight:bold;">{$T.election.hash}</span>
{#/if}
```

Modified template code:

```
<span style="float:right; padding-right:20px;">
<a id="footer_email" target="_new" href="">help!</a>
</span>
<div id="footer_BOGUS_P" style="display: none;">
The public key for this election is not yet ready. This election is in preview mode only.
</div>
<div id="footer_NOT_BOGUS_P" style="display: none;">
Election Fingerprint: <span id="election_hash" style="font-weight:bold;"></span>
</div>
```

Custom footer template processing function:

```
function processTemplate_footer() {
  // process template
  document.getElementById('footer_email').setAttribute('href', "mailto:" + BOOTH.election_metadata.help_email + "?subject=help%20with%20election%20" + BOOTH.election.name + "&body=I%20need%20help%20with%20election%20" + BOOTH.election.uuid);
  if( BOOTH.election.BOGUS_P) {
    document.getElementById('footer_BOGUS_P').style.display = "";
  }
  else {
    document.getElementById('election_hash').textContent = BOOTH.election.hash;
    document.getElementById('footer_NOT_BOGUS_P').style.display = "";
  }
}
```

A.5 Underscore

Underscore is a utility library for JavaScript that provides, for example, some tools to manipulate JavaScript arrays or objects, of which Helios uses a few. Fortunately, they are easy to rewrite in native JavaScript code.

For instance, Helios uses yet another iterator for arrays, a map function, a function that decides whether a given array contains a specified element, and a function to enumerate all names of an object's properties:

```
_(arr).each( function(value, index) { /* code */ });
_(arr).map( function(value, index) { /* code */ });
_(arr).include( v);
_.keys(obj);
```

All of these have native equivalents:

```
arr.forEach( function(value, index) { /* code */ });
arr.map( function(value, index) { /* code */ });
arr.indexOf(v) != -1;
Object.keys(obj);
```

Another feature used by Helios is the removal of all `null` elements of an array:

```
var results = _.reject( arr, _.isNull);
```

This is easily done in native JavaScript:

```
var results = [];
arr.forEach( function(value) {
  if( value !== null) {
    results[results.length] = value;
  }
});
```

Lastly, a function that extends one object with all the properties of another object, resulting in a union of the two objects, is used by Helios:

```
_.extend(obj, obj2);
```

It is again straightforward to do this in native JavaScript:

```
function underscore_extend(obj1, obj2){
    for(var key in obj2)
        if(obj2.hasOwnProperty(key))
```

```
            obj1[key] = obj2[key];
    return obj1;
};
underscore_extend(obj, obj2);
```

These simple modifications allow us to get rid of the Underscore library, a highly complex library with a multitude of functions, approximately 30 kilobytes in size, for our static analysis.

A.6 Class Inheritance

Finally, Helios uses John Resig's implementation of class inheritance,[1] which implements a technique to create classical objects with constructors and simulates classical inheritance in JavaScript. This library declares an object `Class` that can be used like so to declare a "class" `Person`:

```
var Person =
  Class.extend({
    init:
      function( isDancing) {
        this.dancing = isDancing;
      },
    dance:
      function() {
        return this.dancing;
      }
  });

var p = new Person( true);
var p2 = new Person( false);

p.dance(); // Outputs: true
p2.dance(); // Outputs: false
```

Here, the function `init` is a special function that is being used as a constructor: It is called when a new object `Person` is created via the `new` keyword. The library also implements class inheritance (i.e., we could use `Person.extend` in the above code to declare a new class that "inherits" from `Person`). However, the Helios client does not use these inheritance capabilities. Therefore, since Helios exclusively uses this library to declare class-like objects, we can easily

[1] http://ejohn.org/blog/simple-javascript-inheritance

do the same thing in pure JavaScript. Indeed, in JavaScript, functions are already first-class objects, so the following code achieves the same thing as the above:

```
var Person =
  function( isDancing) {
    this.dance =
      function() {
        return this.dancing;
      }

    this.dancing = isDancing;
  };

var p = new Person( true);
var p2 = new Person( false);

p.dance(); // Outputs: true
p2.dance(); // Outputs: false
```

We need to move the “constructor” of the object (i.e., the code that was declared in the special `init` function) to the end of the function declaration, since it may want to invoke some of its internally declared functions.

Appendix B

PHP Code Property Graphs: Definitions and Queries

B.1 AST Node Types

Here, we list all AST node types used to represent the entire PHP language. There are two types of nodes: nodes with a fixed number of children, and nodes with an arbitrary number of children. The majority of nodes has a fixed number of children. For these, each child has a specific role, which we also specify here. Some nodes can have an arbitrary number of children. These are typically list-type nodes, such as the AST node representing a statement list. See Section 3.1.2 for a more detailed discussion.

B.1.1 Nodes with a fixed number of children

Nodes with exactly 0 children (leaf nodes). These are the leaf nodes of the AST. These use properties to reflect the final content, e.g., a node of type `integer` has a property to hold the concrete integer.

Node	Children	Description	Example
`NULL`	-	Used when an interior node with a fixed number of children does not require a given child.	-
`integer`	-	Integer.	`42`
`double`	-	Double.	`3.14`
`string`	-	String (also used to hold identifiers, such as the name of a called function).	`"Hello World"`
`MAGIC_CONST`	-	Magic constant name.	`__FILE__`
`TYPE`	-	PHP type hint such as a parameter type or a function return type.	`function foo( array $bar) : callable {}`

Nodes with exactly 1 child.

Node	Children	Description	Example
TOPLEVEL	1. stmts	Special node to hold the top-level code of a file or of a class.	-
NAME	1. name	Used to identify names in PHP code that may be qualified, such as for example the name of a class that a class declaration extends.	`class Foo extends \B\Bar {}`
CLOSURE_VAR	1. name	Special node holding a variable that occurs within the `use` language construct.	`function() use ($foo) {};`
VAR	1. name	Variable.	`$foo`
CONST	1. name	Constant.	`FOO`
UNPACK	1. expr	Unpack operator (also known as the splat operator) useful in conjunction with variadic functions.	`foo( ...$traversable)`
UNARY_PLUS	1. expr	Unary plus.	`+$x`
UNARY_MINUS	1. expr	Unary minus.	`-$x`
CAST	1. expr	Cast expression.	`(string)42`
EMPTY	1. expr	`empty` language construct.	`empty($foo)`
ISSET	1. var	`isset` language construct.	`isset($foo)`
SILENCE	1. expr	Silence operator.	`@foo()`

Node	Children	Description	Example
SHELL_EXEC	1. expr	Shell command execution expression.	`ls`
CLONE	1. expr	clone language construct.	clone($foo)
EXIT	1. expr	exit language construct.	exit($foo);
PRINT	1. expr	Print expression.	print($foo)
INCLUDE_OR_EVAL	1. expr	Include or eval expression.	include 'foo.php' or eval("$evil")
UNARY_OP	1. expr	Unary operations.	!$foo
PRE_INC	1. var	Pre-increment operation.	++$i
PRE_DEC	1. var	Pre-decrement operation.	--$i
POST_INC	1. var	Post-increment operation.	$i++
POST_DEC	1. var	Post-decrement operation.	$i--
YIELD_FROM	1. expr	yield from expression (to delegate work to other generators).	yield from foo()
GLOBAL	1. var	global statement.	global $bar;
UNSET	1. var	unset statement.	unset($foo);
RETURN	1. expr	Return statement.	return 42;
LABEL	1. name	Label declaration.	here:
REF	1. var	Variable reference.	&$foo
HALT_COMPILER	1. offset	__halt_compiler statement.	__halt_compiler();
ECHO	1. expr	Echo statement.	echo $foo;

Node	Children	Description	Example
THROW	1. expr	Throw statement.	`throw new Exception();`
GOTO	1. label	Goto statement.	`goto here;`
BREAK	1. depth	Break statement.	`break 2;`
CONTINUE	1. depth	Continue statement.	`continue 3;`

Nodes with exactly 2 children.

Node	Children	Description	Example
DIM	1. expr 2. dim	Array indexing expression.	`$foo[42]`
PROP	1. expr 2. prop	Property access expression.	`$foo->bar`
STATIC_PROP	1. class 2. prop	Static property access expression.	`Foo::$bar`
CALL	1. expr 2. args	Call expression.	`foo($bar)`
CLASS_CONST	1. class 2. const	Class constant access expression.	`Foo::BAR`
ASSIGN	1. var 2. expr	Assignment expression.	`$foo = 42`
ASSIGN_REF	1. var 2. expr	Assignment by reference expression.	`$foo =& $bar`
ASSIGN_OP	1. var 2. expr	Assignment expression with operation.	`$foo += 42`
BINARY_OP	1. left 2. right	Binary operation expression.	`$x + $y`

Node	Children	Description	Example
GREATER	1. left 2. right	*Greater than* expression.	`$x > $y`
GREATER_EQUAL	1. left 2. right	*Greater than or equal to* expression.	`$x >= $y`
AND	1. left 2. right	*Boolean and* expression.	`$x && $y`
OR	1. left 2. right	*Boolean or* expression.	`$x || $y`
ARRAY_ELEM	1. value 2. key	Individual elements of an `array` expression.	`array("somekey" => 42)`
NEW	1. class 2. args	`new` expression.	`new Foo($bar)`
INSTANCEOF	1. expr 2. class	`instanceof` expression.	`$foo instanceof Bar`
YIELD	1. value 2. key	`yield` expression.	`yield $somekey => bar()`
COALESCE	1. left 2. right	Coalesce expression.	`$foo ?? "bar"`
STATIC	1. var 2. default	Static variable declaration statement.	`static $foo = 42;`
WHILE	1. cond 2. stmts	While loop.	`while(true) {};`
DO_WHILE	1. stmts 2. cond	Do-while loop.	`do {} while(true);`
IF_ELEM	1. cond 2. stmts	Individual `if`/`elseif`/`else` block of an if-statement.	`if($foo) {}`

Node	Children	Description	Example
SWITCH	1. cond 2. stmts	Switch-statement.	`switch ($i) {}`
SWITCH_CASE	1. cond 2. switchlist	Case of a switch-statement.	`case "foo":`
DECLARE	1. declares 2. stmts	`declare` statement.	`declare (ticks=1) {}`
PROP_ELEM	1. name 2. default	Element of a property declaration list.	`public $foo, $bar = 42;`
CONST_ELEM	1. name 2. value	Element of a constant declaration list.	`const FOO = "", BAR = 42;`
USE_TRAIT	1. traits 2. adaptations	Trait use statement.	`use Foo {}`
TRAIT_PRECEDENCE	1. method 2. insteadof	Trait precedence statement.	`use Foo, Bar { Bar::baz insteadof Foo; }`
METHOD_REFERENCE	1. class 2. method	Method reference in a trait use statement.	`use Foo { Foo::bar as protected; }`
NAMESPACE	1. name 2. stmts	Namespace statement.	`namespace Foo {}`
USE_ELEM	1. name 2. alias	Element of a `use` statement.	`use Foo as Bar, Baz as Qux;`
TRAIT_ALIAS	1. method 2. alias	Trait alias statement.	`use Foo { Foo::bar as protected baz; }`
GROUP_USE	1. prefix 2. uses	Group `use` statement.	`use Foo\{Bar as B, Baz as C};`

Nodes with exactly 3 children.

Node	Children	Description	Example
`CLASS`	1. `extends` 2. `implements` 3. `stmts`	Class declaration.	`class Foo extends Bar implements Baz {}`
`METHOD_CALL`	1. `expr` 2. `method` 3. `args`	Method call expression.	`$foo->bar($baz)`
`STATIC_CALL`	1. `class` 2. `method` 3. `args`	Static method call expression.	`Foo::bar($baz)`
`CONDITIONAL`	1. `cond` 2. `trueexpr` 3. `falseexpr`	Ternary conditional operator.	`$cond ? "foo" : "bar"`
`TRY`	1. `trystmts` 2. `catchlist` 3. `finalstmts`	Try statement.	`try {} catch(Ex $e) {} finally {}`
`CATCH`	1. `exception` 2. `var` 3. `stmts`	Catch statement.	`catch(Ex $e) {}`
`PARAM`	1. `type` 2. `name` 3. `default`	Function parameter.	`function foo(int $bar = 42) {}`

Nodes with exactly 4 children.

Node	Children	Description	Example
`FUNC`	1. `params` 2. `uses` 3. `stmts` 4. `returntype`	Function declaration.	`function foo() : int {}`

Node	Children	Description	Example
`CLOSURE`	1. `params` 2. `uses` 3. `stmts` 4. `returntype`	Closure declaration.	`function() use ($foo) : int {};`
`METHOD`	1. `params` 2. `uses` 3. `stmts` 4. `returntype`	Method declaration.	`class Foo { function foo() : int {} }`
`FOR`	1. `init` 2. `cond` 3. `loop` 4. `stmts`	For-loop.	`for ($i = 0; $i < 3; $i++) {}`
`FOREACH`	1. `expr` 2. `value` 3. `key` 4. `stmts`	Foreach-loop.	`foreach ($foo as $key => $val) {}`

B.1.2 Nodes with an arbitrary number of children

As previously mentioned, some nodes have a list character and can have an arbitrary number of children. These are the following.

Node	Children	Description	Example
`ARG_LIST`	At least 0.	List of arguments.	`foo($bar, $baz)`
`LIST`	At least 1.	`list` language construct.	`list($a, $b) = array(3, 42)`
`ARRAY`	At least 0.	`array` language construct.	`array(3, 42)`
`ENCAPS_LIST`	At least 1.	Used for strings with encapsulated variables.	`"Hello $foo"`
`EXPR_LIST`	At least 1.	Holds a list of expressions.	`for ($i = 0, $j = 0; $i < 3; $i++) {}`

Node	Children	Description	Example
`STMT_LIST`	At least 0.	Holds a list (i.e., a *block*) of statements.	`{}`
`IF`	At least 1.	Holds a number of `if`/`elseif`/`else` blocks.	`if($foo) {} elseif($bar) {} else {}`
`SWITCH_LIST`	At least 0.	Holds a number of switch blocks.	`switch ($i) { case "foo": case "bar": break; }`
`CATCH_LIST`	At least 0.	Holds a number of catch blocks.	`try {} catch(Foo $f) {} catch(Bar $b) {}`
`PARAM_LIST`	At least 0.	List of function parameters.	`function foo($bar, $baz) {}`
`CLOSURE_USES`	At least 1.	List of variables to import into a closure.	`function() use ($foo,$bar) {};`
`PROP_DECL`	At least 1.	List of property declarations.	`public $foo, $bar = 42;`
`CONST_DECL`	At least 1.	List of constant declarations.	`const FOO = "", BAR = 42;`
`CLASS_CONST_DECL`	At least 1.	List of class constant declarations.	`class Baz { const FOO = "", BAR = 42; }`
`NAME_LIST`	At least 1.	Holds a list of names, such as a list of interfaces a class implements.	`class Foo implements Bar, Baz {}`
`TRAIT_ADAPTATIONS`	At least 1.	Holds a list of trait use and trait precedence statements.	`use Foo, Bar { Bar::baz insteadof Foo; Foo::qux as protected; }`
`USE`	At least 1.	Holds a number of `use` elements.	`use Foo as Bar, Baz as Qux;`

B.2 Graph Traversals

B.2.1 Indexing queries

As described in Section 5.2.3.4, the first step in our detection process is the indexing of security-critical function calls. We first create an index which maps each AST node type to the set of AST node ids with the given type. This makes all subsequent queries to index AST nodes with a given type and a given set of properties significantly more efficient:

```
CREATE INDEX ON :AST(type);
```

In the following, we list all Cypher queries to detect security-critical function calls pertaining to different classes of vulnerabilities.

SQL Injections. We identify calls to the built-in functions `mysql_query`, `pg_query`, and `sqlite_query`. We do so in three different queries, as different sanitizers apply and we therefore run three distinct analyses.

```
MATCH
  ↪ (node:AST)-[:PARENT_OF]->(expr:AST)-[:PARENT_OF]->(name:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_CALL'
  AND expr.type = 'AST_NAME'
  AND name.code = 'mysql_query'
RETURN node.id;
```

```
MATCH
  ↪ (node:AST)-[:PARENT_OF]->(expr:AST)-[:PARENT_OF]->(name:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_CALL'
  AND expr.type = 'AST_NAME'
  AND name.code = 'pg_query'
RETURN node.id;
```

```
MATCH
  ↪ (node:AST)-[:PARENT_OF]->(expr:AST)-[:PARENT_OF]->(name:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_CALL'
  AND expr.type = 'AST_NAME'
  AND name.code = 'sqlite_query'
RETURN node.id;
```

Command Injection. Shell commands can be executed using either the backtick operator or the PHP function calls `shell_exec` and `popen`. We collect all corresponding nodes using the following query.

```
MATCH (node:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_SHELL_EXEC'
RETURN node.id
UNION
MATCH (node:AST)-[:PARENT_OF]->(expr:AST)-[:PARENT_OF]->(name:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_CALL'
  AND expr.type = 'AST_NAME'
  AND name.code IN ['shell_exec','popen']
RETURN node.id;
```

Code Injection. Code can be injected either directly if an attacker can control the argument passed to the PHP construct `eval`, or indirectly if an attacker can control the argument passed to `include`, `require`, `include_once`, or `require_once`. We use the following queries to identify these two types of constructs, respectively.

```
MATCH (node:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_INCLUDE_OR_EVAL'
  AND 'EXEC_EVAL' IN node.flags
RETURN node.id;
```

```
MATCH (node:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_INCLUDE_OR_EVAL'
  AND NOT 'EXEC_EVAL' IN node.flags
RETURN node.id;
```

Arbitrary File Reads/Writes. This type of vulnerability may occur if an attacker can control input passed to the function `fopen`. We identify these calls using the following query.

```
MATCH (node:AST)-[:PARENT_OF]->(expr:AST)-[:PARENT_OF]->(name:AST)
USING INDEX node:AST(type)
```

```
WHERE node.type = 'AST_CALL'
  AND expr.type = 'AST_NAME'
  AND name.code = 'fopen'
RETURN node.id;
```

Cross-Site Scripting (XSS). Reflecting user input (or, more generally, outputting anything) in PHP normally involves using either the `echo` or `print` language constructs, both of which have a dedicated node type. Note that text outside of `<?php ... ?>` tags in a PHP file is simply interpreted as a string passed to `echo` by the parser.

```
MATCH (node:AST)
USING INDEX node:AST(type)
WHERE node.type IN ['AST_ECHO', 'AST_PRINT']
RETURN node.id;
```

Session Fixation. We identify calls to the PHP function `setcookie` using the following query.

```
MATCH
 ↪ (node:AST)-[:PARENT_OF]->(expr:AST)-[:PARENT_OF]->(name:AST)
USING INDEX node:AST(type)
WHERE node.type = 'AST_CALL'
  AND expr.type = 'AST_NAME'
  AND name.code = 'setcookie'
RETURN node.id;
```

B.2.2 Vulnerability-detection queries

The general methodology for finding suspicious data flows has been explained in Section 5.2.3.4. In essence, the function for identifying vulnerable data flows remains the same independently of the particular vulnerability we are looking for. What changes is the definition of what we consider a security-critical function call (i.e., the sink) and what we consider an appropriate sanitizer for the sink in question. The advantage of our framework is that these definitions—and in fact, the whole traversal—can be rewritten or implemented from scratch depending on the analysis needs in a given context. In this section, we give the sources, sinks, and sanitizers which we used for our evaluation presented in Section 5.4, as well as the complete implementation of the function sketched in Section 5.2.3.4.

Sources. We consider additional sources than those mentioned in Section 5.2.3 for compatibility reasons (older versions of PHP used different variables). The complete definition of sources for vulnerabilities potentially resulting in server-side attacks is as follows.

```
def isLowSource( Neo4j2Vertex it) {

  if( it.type == TYPE_VAR) {
    return getVarName(it) in [
      // modern variables (>= PHP 4.1)
      "_GET", "_POST", "_COOKIE", "_REQUEST", "_FILES",
      // variants used up to PHP 4.1, deprecated since PHP 4.2
      "HTTP_GET_VARS", "HTTP_POST_VARS", "HTTP_COOKIE_VARS", "HTTP_POST_FILES",
      // deprecated since PHP 5.6, removed as of PHP 7.0
      "HTTP_RAW_POST_DATA",
      // really old (prior to PHP 4.1) variables that were available as global variables
      // in addition to being available as keys in $HTTP_SERVER_VARS (and later, $_SERVER)
      "HTTP_ACCEPT", "HTTP_ACCEPT_CHARSET", "HTTP_ACCEPT_ENCODING", "HTTP_ACCEPT_LANGUAGE",
      "HTTP_CONNECTION", "HTTP_HOST", "HTTP_REFERER", "HTTP_USER_AGENT",
      "REQUEST_URI", "QUERY_STRING"];
  }
  else if( it.type == TYPE_DIM) {
    Neo4j2Vertex var = getDimVar(it);
    return (var.type == TYPE_VAR &&
            getVarName(var) in ["_SERVER", "HTTP_SERVER_VARS"] &&
            getDimKey(it).code ==~ /HTTP_.*|REQUEST_URI|QUERY_STRING/);
  }

  return false;
}
```

The definition for vulnerabilities resulting in client-side attacks is almost identical, except that we do not consider cookies a viable attack avenue, as discussed in Section 5.2.3.

```
def isLowSource( Neo4j2Vertex it) {

  if( it.type == TYPE_VAR) {
    return getVarName(it) in [
      // modern variables (>= PHP 4.1)
      "_GET", "_POST", "_REQUEST", "_FILES",
      // variants used up to PHP 4.1, deprecated since PHP 4.2
      "HTTP_GET_VARS", "HTTP_POST_VARS", "HTTP_POST_FILES",
      // deprecated since PHP 5.6, removed as of PHP 7.0
      "HTTP_RAW_POST_DATA",
      // really old (prior to PHP 4.1) variables that were available as global variables
      // in addition to being available as keys in $HTTP_SERVER_VARS (and later, $_SERVER)
      "HTTP_ACCEPT", "HTTP_ACCEPT_CHARSET", "HTTP_ACCEPT_ENCODING", "HTTP_ACCEPT_LANGUAGE",
      "HTTP_CONNECTION", "HTTP_HOST", "HTTP_REFERER", "HTTP_USER_AGENT",
      "REQUEST_URI", "QUERY_STRING"];
  }
  else if( it.type == TYPE_DIM) {
    Neo4j2Vertex var = getDimVar(it);
    return (var.type == TYPE_VAR &&
            getVarName(var) in ["_SERVER", "HTTP_SERVER_VARS"] &&
            getDimKey(it).code ==~ /HTTP_.*|REQUEST_URI|QUERY_STRING/);
  }

  return false;
}
```

Sinks and sanitizers. The definition of a valid sanitizer depends on the sink in question. We therefore present the exact sanitizers considered appropriate for each security-critical function call that we analyze in our evaluation presented in Section 5.4.

For mysql_query:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["mysql_real_escape_string", "mysql_escape_string", "addslashes", "crypt", "md5", "sha1"]);
}
```

For pg_query:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["pg_escape_string", "addslashes", "crypt", "md5", "sha1"]);
}
```

For sqlite_query:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["sqlite_escape_string", "addslashes", "crypt", "md5", "sha1"]);
}
```

For shell_exec, the backtick operator and popen:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["escapeshellarg", "escapeshellcmd", "crypt", "md5", "sha1"]);
}
```

For eval:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["crypt", "md5", "sha1"]);
}
```

For include, require, include_once and require_once:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["crypt", "md5", "sha1"]);
}
```

For fopen:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["crypt", "md5", "sha1"]);
}
```

For echo and print:

```
def isSanitizer( Neo4j2Vertex it) {
  if( it.type == TYPE_CALL) {
    String calledFunc = getCalledFuncName(it);
    if( calledFunc in ["htmlentities", "strip_tags", "crypt", "md5", "sha1"]) {
      return true;
    }
    else if( calledFunc == "htmlspecialchars") {
      // make sure ENT_QUOTES is given within the second argument
      if( getArgCount( it) >= 2) {
        Neo4j2Vertex secondArg = it.ithArguments(1).next();
        return secondArg.match{ it.type == TYPE_CONST && getConstName(it) == "ENT_QUOTES" }.count() > 0;
      }
    }
  }

  return false;
}
```

For setcookie:

```
def isSanitizer( Neo4j2Vertex it) {
  return (it.type == TYPE_CALL &&
          getCalledFuncName(it) in ["crypt", "md5", "sha1"]);
}
```

Identifying critical data flows. For the sake of completeness, we present the complete traversal which we sketched and explained a simplified version of in Section 5.2.3.4. It tackles some technicalities, but its fundamental idea is exactly the same.

```
def init( Vertex node) {

  List finalflows = outputPaths( interprocSearchFrom( node, 0));
  return finalflows;
}

/**
   Takes a sink and performs an interprocedural backwards data
   dependence analysis from that sink. A sink can be any AST node.

   Returns an array list of paths.
   In other words, returns an array list of array lists of vertices.
 */
def interprocSearchFrom( Neo4j2Vertex sink, int recdepth) {
  List finalflows = new ArrayList();

  // consider at most maxdepth "function jumps"
  if( recdepth > maxdepth) {
    // return empty list
    return [];
  }

  else if( containsSanitizer( sink)) {
    // return empty list
    return [];
  }
```

```
  else if( containsLowSource( sink)) {
    // no need to traverse, we have a low source -- return the single-element "list" of a single-element "path"
    return [[sink]];
  }

  else {

    // no low source and no sanitizer -- we traverse back along data dependence edges

    // for the given sink, we first find the variables it uses,
    // because we will only want to travel back the data dependence
    // edges for these variables
    List<String> varnames = getVarNamesForSink(sink).toList();
    List flows = visit(sink, varnames).toList();

    int i = 0;
    for( List path in flows) {

      // for the paths that begin in a parameter, we recurse
      if( path.size() > 0 && path.last().type == TYPE_PARAM) {
        ArrayList callSiteArgs = jumpToCallSiteArgs(path.last()).toList();
        List callingFuncPaths = new ArrayList();
        for( Neo4j2Vertex arg in callSiteArgs) {
          callingFuncPaths.addAll( interprocSearchFrom(arg, recdepth + 1)); // recursion!
        }
        // add all combined paths from calling functions to flows that we found (if any) to sinks
        for( List callingFuncPath : callingFuncPaths) {
          finalflows.add( path + callingFuncPath);
        }
      }
      else {
        // if it's not a parameter, it must have been a low source,
        // because those are the only two things that visit() emits---add it
        finalflows.add( path);
      }
    }
  }

  return finalflows;
}

/**
   For a given sink, finds all variables used within that sink and
   returns their names as a Gremlin Groovy pipeline.
 */
def getVarNamesForSink( Neo4j2Vertex sink) {
  sink
  .match{ it.type == TYPE_VAR }
  .varToName()
}

/**
 * Traverses a function backwards along the data dependence edges from
 * a given node to be considered as a sink.
 */
def visit( Neo4j2Vertex sink, List<String> varnames) {
  sink
  // traverse to enclosing CFG node;
  .statements()
```

```
  // define a set that will contain "traversed nodes" from this statement
  .sideEffect{ seen_nodes = []; firstiteration = true; }

  .as('loopbegin')

  // only iterate over source nodes we have not seen yet
  .filter{ !(it in seen_nodes) }
  // save current node as seen
  .sideEffect{ seen_nodes << it }

  // traverse data dependence edges backwards
  // in the very first iteration, we only travel back those
  // data dependence edges specified in the list of varnames used in this sink
  .ifThenElse{ firstiteration }
  {
    it
    .inE(DATA_FLOW_EDGE).filter{ it.var in varnames }.outV()
  }
  { it.sources() }
  .sideEffect{ firstiteration = false; }

  // eliminate those source statements that contain sanitizers
  .filter{ !containsSanitizer(it) }

  // check whether we found a low source and save that to a boolean
  .sideEffect{ foundlowsource = containsLowSource(it) }

  // loop until:
  // - it.object becomes null (no more sources),
  // - or we have iterated more than thirty times,
  // - or we already found a low source anyway (-> condition for looping: !foundlowsource)
  // - or we found a parameter
  // emit only statements that contain low source variables (foundlowsource) or that are
  ↪   parameters
  .loop('loopbegin'){ it.object != null && it.loops <= 30 && !foundlowsource &&
  ↪   it.object.type != TYPE_PARAM }{ foundlowsource || it.object.type == TYPE_PARAM }

  // finally, return the found paths
  .path.dedup
}

def jumpToCallSiteArgs( Neo4j2Vertex param) {
  param
  .sideEffect{ paramNum = it.childnum }
  // traverse to enclosing function
  .functions()
  // traverse to callers
  .functionToCallers()
  // get the paramNum'th argument
  .callToArgumentList()
  .children().filter{ it.childnum == paramNum }
}

// decides whether a node's subtree contains a sanitizer
def containsSanitizer( it) {
  return it.match{ isSanitizer(it) }.count() > 0;
}

// decides whether a node's subtree contains a low source
def containsLowSource( it) {
  return it.match{ isLowSource(it) }.count() > 0;
}
```

B.3 List of scanned projects

In this section we give the complete list of the 1,854 projects that we used for our large-scale evaluation in Section 5.4. All of these can be found at https://github.com/<PROJECT>.

The set $\mathcal{C}$ of 4 pieces of explicitly vulnerable software or web shells is the following:

Audi-1/sqli-labs
pfsense/pfsense
RandomStorm/DVWA
tennc/webshell

The set $\mathcal{P}$ of the remaining 1,850 projects that we scanned for vulnerabilities is the following.

10up/wp_mock
12meses12katas/Enero-String-Calculator
12meses12katas/Marzo-FizzBuzz
1up-lab/OneupUploaderBundle
2b3ez/FileManager4TinyMCE
320press/wordpress-bootstrap
320press/wordpress-foundation
a1phanumeric/PHP-MySQL-Class
a2lix/TranslationFormBundle
aarondunn/bugkick
aaronpk/Google-Voice-PHP-API
abenzer/represent-map
Abhoryo/APYDataGridBundle
abraham/twitteroauth
Abstrct/Schemaverse
ACCORD5/TrellisDesk
achingbrain/php5-akismet
adamfisk/LittleProxy
adamgriffiths/ag-auth
adldap/adLDAP
adoy/PHP-OAuth2
adriengibrat/torrent-rw
advocaite/Travianx
afragen/github-updater
afreiday/php-waveform-png
afterlogic/webmail-lite
ahmadnassri/restful-zend-framework
akalongman/sublimetext-codeformatter
akanehara/ginq
akDeveloper/Aktive-Merchant
akeneo/pim-community-dev
akrabat/zf2-tutorial
AKSW/OntoWiki
akuzemchak/laracon-todo-api
alanhogan/lessnmore
alchemy-fr/Zippy
alexbilbie/MongoQB
AliasIO/Swiftlet
alistairstead/MageTool
alixaxel/ArrestDB
allegro/php-protobuf
alleyinteractive/wordpress-fieldmanager
alliswell/Less
AlloVince/eva-engine
AlloVince/EvaThumber
alloyphp/alloy
allynbauer/statuspanic
alxlit/coffeescript-php
amal/AzaThread
amazonwebservices/aws-sdk-for-php
amnuts/opcache-gui
ampache/ampache
amphp/artax
amphp/thread
anahitasocial/anahita
Anahkiasen/flatten
Anahkiasen/polyglot
Anahkiasen/underscore-php
anandkunal/ToroPHP
anantgarg/Qwench
anchepiece/statuspanic
anchorcms/anchor-cms
andacata/HybridIgniter
andraskende/cakephp-shopping-cart
andres-montanez/Magallanes
andrespagella/Making-Isometric-Real-time-Games
andrewbiggart/latest-tweets-php-o-auth
Andrewsville/PHP-Token-Reflection
angelleye/paypal-php-library
angeloskath/php-nlp-tools
Annotum/Annotum
antecedent/patchwork
anthonyshort/Scaffold
antimattr/GoogleBundle
antonraharja/playSMS
AntonTerekhov/OrientDB-PHP
AOEpeople/Aoe_Profiler
AOEpeople/Aoe_Scheduler
Aoiujz/ThinkSDK
ApiGen/ApiGen
apotropaic/parse.com-php-library
appleseedproj/appleseed
appserver-io/appserver
aramk/crayon-syntax-highlighter
Arara/Process
ariok/codeigniter-boilerplate
aristath/bootstrap-admin
arnaud-lb/MtHaml
arshaw/phpti
artdarek/oauth-4-laravel
aschroder/Magento-SMTP-Pro-Email-Extension
asimlqt/php-google-spreadsheet-client
asm89/Rx.PHP
asm89/twig-cache-extension
astorm/Pulsestorm
Athari/YaLinqo
atk4/atk4
Atlantic18/DoctrineExtensions
atoum/atoum
atrauzzi/laravel-doctrine
atrilla/nlptools
auraphp/Aura.Di
auraphp/Aura.Marshal
auraphp/Aura.Router
auraphp/Aura.Sql
Austinb/GameQ
authy/authy-php
Automattic/babble
Automattic/batcache
Automattic/camptix
Automattic/Co-Authors-Plus
Automattic/custom-metadata
Automattic/developer
Automattic/Edit-Flow

Automattic/_s
Automattic/vip-scanner
Automattic/WP-Job-Manager
avalanche123/AvalancheImagineBundle
avalanche123/Imagine
avstudnitz/AvS_FastSimpleImport
Awilum/monstra-cms
aws/aws-sdk-php
aws/aws-sdk-php-laravel
Azure/azure-sdk-for-php
badphp/dispatch
bainternet/PHP-Hooks
bainternet/Tax-Meta-Class
banks/kohana-email
barbushin/dater
barbushin/php-imap
BarrelStrength/Craft-Master
barryvdh/laravel-dompdf
barryvdh/laravel-elfinder
barryvdh/laravel-ide-helper
barryvdh/laravel-migration-generator
barryvdh/laravel-vendor-cleanup
basho/riak-php-client
bastianallgeier/gantti
bastianallgeier/kirby
bastianallgeier/kirbycms
bastianallgeier/kirbycms-extensions
bastianallgeier/kirbycms-panel
bcit-ci/CodeIgniter
bcosca/fatfree
bearsunday/BEAR.Sunday
beberlei/AcmePizzaBundle
beberlei/assert
beberlei/DoctrineExtensions
beberlei/litecqrs-php
beberlei/metrics
beberlei/zf-doctrine
Behat/Behat
Behatch/contexts
Behat/CommonContexts
Behat/MinkExtension
Behat/Symfony2Extension
benbalter/wordpress-to-jekyll-exporter
benedmunds/codeigniter-cache
benedmunds/CodeIgniter-Ion-Auth
BenExile/Dropbox
benkeen/generatedata
bergie/dnode-php
bergie/phpflo
bernardphp/bernard
berta-cms/berta
BeSimple/BeSimpleI18nRoutingBundle
bfintal/bfi_thumb
bilalq/Tranquillity-Editor
billerickson/Core-Functionality
billerickson/display-posts-shortcode
BIOSTALL/CodeIgniter-Google-Maps-V3-API-Library
bjankord/Categorizr
bjyoungblood/BjyAuthorize
BKWLD/croppa
blind-coder/rcmcarddav
bllim/laravel4-datatables-package
blongden/hal
blt04/doctrine2-nestedset
bmidget/kohana-formo
bobthecow/Faker
bobthecow/mustache.php
bobthecow/psysh
bobthecow/Ruler
boldperspective/Whiteboard-Framework
bolt/bolt
booruguru/UserPie
bootsz/wp-advanced-search
borisrepl/boris
bortuzar/PHP-Mysql---Apple-Push-Notification-Server
box/bart
boxbilling/boxbilling
box-project/box2
braincrafted/bootstrap-bundle
braintree/braintree_php
bramus/router
brandonsavage/Upload
brandonwamboldt/utilphp
briancray/phpA-B
briancray/PHP-URL-Shortener
brianhaveri/Underscore.php
brianium/paratest
brianlmoon/GearmanManager
briannesbitt/Carbon
briannesbitt/Slim-ContextSensitiveLoginLogout
browscap/browscap
browscap/browscap-php
brunogaspar/laravel-starter-kit
bshaffer/oauth2-demo-php
bshaffer/oauth2-server-php
bstrahija/assets-ci
bstrahija/l4-site-tutorial
btroia/basis-data-export
bueltge/WordPress-Admin-Style
buggedcom/phpvideotoolkit-v2
burzum/cakephp-file-storage
burzum/cakephp-imagine-plugin
c9s/CLIFramework
CaerCam/WPMedium
cakebaker/openid-component
CakeDC/migrations
CakeDC/search
CakeDC/tags
CakeDC/TinyMCE
CakeDC/users
CakeDC/utils
cakephp/api_generator
cakephp/cakephp
cakephp/cakephp-codesniffer
cakephp/datasources
cakephp/debug_kit
cakephp/localized
calvinfroedge/codeigniter-payments
calvinfroedge/PHP-Payments
campaignmonitor/createsend-php
canni/YiiMongoDbSuite
canton7/fuelphp-casset
caouecs/Laravel-lang
CaptainRedmuff/UIColor-Crayola
captn3m0/ifttt-webhook
cartalyst/sentry
cashmusic/platform
catfan/Medoo
cboden/Ratchet-examples
cbschuld/Browser.php
ccampbell/chromephp
ccoenraets/angular-cellar
ccoenraets/backbone-directory
ccoenraets/offline-sync
ccoenraets/wine-cellar-php
cdhowie/Bitcoin-mining-proxy
cdukes/bones-for-genesis-2-0
ceesvanegmond/minify
centurion-project/Centurion
Cerdic/CSSTidy
chanmix51/Pomm
charlesportwoodii/CiiMS
charliesome/Fructose
chekun/DiliCMS
cheshirecats/CuriousWall
CHH/pipe
chibimagic/WebDriver-PHP
chnm/anthologize
chobie/php-sundown
chregu/GoogleAuthenticator.php
chrisboulton/php-diff
chrisboulton/php-resque
chrisboulton/php-resque-scheduler
chriskacerguis/codeigniter-restserver
chriskite/phactory
chriso/klein.php
chrissimpkins/tweetledee
christiaan/InlineStyle
christian-putzke/Roundcube-CardDAV
christianreber/kirbycms-knowledge-base
christophervalles/Zend-Framework-Skeleton
chyrp/chyrp
ci-bonfire/Bonfire
Cilex/Cilex
Circa75/dropplets
citelao/Spotify-for-Alfred
citricsquid/httpstatus.es
civicrm/civicrm-core

ClassPreloader/ClassPreloader
claudehohl/Stikked
claviska/SimpleImage
claviska/simple-php-captcha
clevertech/YiiBoilerplate
clue/graph
clue/graph-composer
clue/phar-composer
cmall/LocalHomePage
cmoore4/phalcon-rest
CobreGratis/boletophp
cobub/razor
cocur/slugify
Codeception/Codeception
codeguy/Slim-Extras
CodeMeme/Phingistrano
CoderKungfu/php-queue
CodeScaleInc/ffmpeg-php
CodeSleeve/asset-pipeline
Codiad/Codiad
codin/dime
codin/roar
coen-hyde/Shanty-Mongo
coinbase/coinbase-php
colinmollenhour/Cm_Cache_Backend_Redis
colinmollenhour/Cm_Diehard
colinmollenhour/Cm_RedisSession
colinmollenhour/credis
colinmollenhour/magento-lite
colinmollenhour/magento-mongo
colinmollenhour/mongodb-php-odm
collegeman/coreylib
colshrapnel/safemysql
composer/composer
composer/installers
composer/packagist
composer/satis
concrete5/concrete5-legacy
consolibyte/quickbooks-php
contao/core
cosenary/Instagram-PHP-API
cosenary/Simple-PHP-Cache
Courseware/buddypress-courseware
CpanelInc/xmlapi-php
cpliakas/git-wrapper
craue/CraueFormFlowBundle
creocoder/yii2-nested-sets
crew-cr/Crew
Crinsane/LaravelShoppingcart
crisu83/yii-app
crisu83/yii-auth
crisu83/yiistrap
crodas/ActiveMongo
crodas/Haanga
crodas/LanguageDetector
croogo/croogo
crowdfavorite/wp-capsule
crowdfavorite/wp-post-formats
crowdfavorite/wp-social
csscomb/csscomb
csscomb/csscomb-for-sublime
cviebrock/eloquent-sluggable
cweiske/phorkie
Cybernox/AmazonWebServicesBundle
dallasgutauckis/parcelabler
daneden/Basehold.it
danielbachhuber/dictator
danielboendergaard/laravel-3-ide-helper
danielmewes/php-rql
dannyvankooten/AltoRouter
dannyvankooten/PHP-Router
danslo/ApiImport
dapphp/securimage
dasmurphy/tinytinyrss-fever-plugin
Datawalke/Coordino
datawrapper/datawrapper
DaveChild/Text-Statistics
davedevelopment/phpmig
davejamesmiller/laravel-breadcrumbs
davemo/end-to-end-with-angularjs
davesloan/mysql-php-migrations
davideme/libphonenumber-for-PHP
davidpersson/media
davidwinter/wordpress-with-git
daylerees/dependency-injection-example
daylerees/kurenai
daylightstudio/FUEL-CMS
dbtlr/php-airbrake
dcooney/wordpress-ajax-load-more
dcramer/wp-lifestream
ddeboer/data-import
ddeboer/imap
debuggable/php_arrays
deliciousbrains/sqlbuddy
deliciousbrains/wp-amazon-s3-and-cloudfront
deliciousbrains/wp-migrate-db
deployphp/deployer
derekallard/BambooInvoice
dereuromark/cakephp-tools
derickr/xdebug
desandro/windex
DevGrow/jQuery-Mobile-PHP-MVC
devinsays/options-framework-plugin
devinsays/options-framework-theme
devinsays/portfolio-post-type
DevinVinson/WordPress-Plugin-Boilerplate
Devristo/phpws
devster/ubench
dflydev/dflydev-doctrine-orm-service-provider
dflydev/dflydev-markdown
dg/dibi
dg/ftp-deployment
dg/ftp-php
dgrundel/woo-product-importer
dg/twitter-php
diem-project/diem
digitalbazaar/php-json-ld
digitalnature/php-ref
dignajar/nibbleblog
DirectoryLister/DirectoryLister
discourse/wp-discourse
disqus/DISQUS-API-Recipes
disqus/disqus-php
dizda/CloudBackupBundle
dmolsen/Detector
docopt/docopt.php
doctrine/annotations
doctrine/cache
doctrine/common
doctrine/couchdb-odm
doctrine/data-fixtures
doctrine/dbal
doctrine/doctrine1
doctrine/doctrine2
doctrine/DoctrineBundle
doctrine/DoctrineFixturesBundle
doctrine/DoctrineMigrationsBundle
doctrine/DoctrineModule
doctrine/DoctrineMongoDBBundle
doctrine/DoctrineORMModule
doctrine/migrations
doctrine/mongodb
doctrine/mongodb-odm
doctrine/orientdb-odm
doctrine/phpcr-odm
doctrine/rest
doctrine/search
dodgepudding/wechat-php-sdk
doitlikejustin/amazon-wish-lister
Dolibarr/dolibarr
domnikl/DesignPatternsPHP
domnikl/statsd-php
dompdf/dompdf
donatj/PhpUserAgent
donjakobo/A3M
download-monitor/download-monitor
dphiffer/wp-json-api
dready92/PHP-on-Couch
drewjoh/phpPayPal
drewkennelly/foundation7
drewsymo/Foundation
dropbox/dropbox-sdk-php
Dropbox-PHP/dropbox-php
drslump/Protobuf-PHP
drupal/drupal
drush-ops/drush
dsyph3r/symblog
dtompkins/fbcmd

dustin10/VichUploaderBundle
dwisetiyadi/CodeIgniter-PHP-QR-Code
dzuelke/HadooPHP
e107inc/e107
easychen/LazyPHP
easychen/LazyREST
easychen/TeamToy
easydigitaldownloads/Easy-Digital-Downloads
EcomDev/EcomDev_PHPUnit
eddiejaoude/Zend-Framework--Doctrine-ORM--PHPUnit--Ant--Jenkins-CI--TDD-
educoder/pest
edvinaskrucas/notification
egeloen/IvoryCKEditorBundle
egeloen/ivory-google-map
egeloen/IvoryGoogleMapBundle
egil/php-markdown-extra-extended
ego008/youbbs
egulias/EmailValidator
ekino/EkinoNewRelicBundle
elastic/elasticsearch-php
electerious/Lychee
Element-34/php-webdriver
elfet/console
elfet/silicone-skeleton
Elgg/Elgg
elidickinson/php-export-data
elliotcondon/acf
elliotcondon/acf-field-type-template
elliothaughin/codeigniter-facebook
elliothaughin/codeigniter-twitter
Emagister/zend-form-decorators-bootstrap
Emerson/Sanity-Wordpress-Plugin-Framework
emoncms/emoncms
emposha/PHP-Shell-Detector
endroid/QrCode
entomb/slim-json-api
envato/envato-wordpress-toolkit
enygma/shieldframework
e-oz/Memory
ericbae/XTA2
ericbarnes/ci-minify
ericbarnes/codeigniter-simpletest
ericmann/Redis-Object-Cache
ericmann/wp-session-manager
ErikDubbelboer/phpRedisAdmin
erikuus/Yii-Playground
ErisDS/Migrate
erusev/parsedown
eryx/php-framework-benchmark
esotalk/esoTalk
essence/essence
etrepat/baum
etsy/ab
etsy/feature
etsy/phpunit-extensions
etsy/TryLib
evan108108/RESTFullYii
EvanDotPro/EdpSuperluminal
evantahler/PHP-DAVE-API
evernote/evernote-sdk-php
everzet/jade.php
Exeu/Amazon-ECS-PHP-Library
Exeu/apai-io
exflickr/flamework
expressodev/ci-merchant
eymengunay/php-passbook
ezimuel/PHP-design-patterns
ezimuel/PHP-Secure-Session
ezsystems/ezpublish-community
ezsystems/ezpublish-kernel
ezyang/htmlpurifier
FabianBeiner/PHP-IMDB-Grabber
fabpot-graveyard/Pirum
fabpot-graveyard/yaml
fabpot/symfony
Fabrik/fabrik
facebookarchive/facebook-php-sdk
facebookarchive/wordpress
facebook/FBMock
facebook/open-graph-protocol
facebook/php-webdriver
Facens/wpbootstrap
faisalman/simple-excel-php
Falicon/BitlyPHP
farinspace/wpalchemy
FbF/Instafilter
fedecarg/apify-library
feelinglucky/php-readability
felixge/cakephp-authsome
fennb/phirehose
fenom-template/fenom
fguillot/picoFeed
fideloper/TrustedProxy
fieryprophet/php-sandbox
fightbulc/moment.php
filamentgroup/quickconcat
FileZ/FileZ
filipstefansson/bootstrap-3-shortcodes
filp/whoops
fire015/flintstone
firebase/php-jwt
firephp/firephp-core
fkooman/php-oauth-as
fkooman/php-oauth-client
flack/createphp
flint/flint
flint/Stampie
flourishlib/flourish-classes
flourishlib/flourish-old
FluentDOM/FluentDOM
fluent/fluent-logger-php
fluxbb/fluxbb
FlyersWeb/angular-symfony
Flynsarmy/PHPWebSocket-Chat
Flyspray/flyspray
fmalk/codeigniter-phpunit
fmbiete/Z-Push-contrib
focuslabllc/ee-master-config
FokkeZB/TiCons-Server-PHP
FoolCode/SphinxQL-Query-Builder
forkcms/forkcms
formers/former
fortrabbit/slimcontroller
franzose/ClosureTable
frapi/frapi
fre5h/DoctrineEnumBundle
FreshRSS/FreshRSS
friendica/friendica
friendica/red
FriendsOfCake/CakePdf
FriendsOfCake/crud
FriendsOfPHP/Goutte
FriendsOfPHP/PHP-CS-Fixer
FriendsOfPHP/Sami
FriendsOfPHP/security-advisories
FriendsOfPHP/Sismo
FriendsOfSymfony/FOSCommentBundle
FriendsOfSymfony/FOSElasticaBundle
FriendsOfSymfony/FOSFacebookBundle
FriendsOfSymfony/FOSJsRoutingBundle
FriendsOfSymfony/FOSMessageBundle
FriendsOfSymfony/FOSOAuthServerBundle
FriendsOfSymfony/FOSRestBundle
FriendsOfSymfony/FOSTwitterBundle
FriendsOfSymfony/FOSUserBundle
FriendsOfSymfony/oauth2-php
Froxlor/Froxlor
Frug/AJAX-Chat
fruux/sabre-dav
fruux/sabre-vobject
fruux/sabre-xml
fuchaoqun/colaphp
fuel/core
fuel/fuel
fuel/oil
fuel/orm
fuelphp/fuelphp
fujimoto/php-skype
funkatron/inspekt
fusonic/chive
fyaconiello/wp_plugin_template
fzaninotto/Faker
fzaninotto/Streamer
gabordemooij/redbean
gabrielbull/php-browser
galen/PHP-Instagram-API
gallery/gallery3
gallery/gallery3-contrib

GaretJax/phpbrowscap
Garfield-fr/Symfony2Project
garo/bigpipe
gaspaio/gearmanui
gboudreau/Greyhole
gboudreau/nest-api
gburtini/Learning-Library-for-PHP
GeekPress/WP-Quick-Install
genemu/GenemuFormBundle
geocoder-php/BazingaGeocoderBundle
geocoder-php/Geocoder
getsentry/raven-php
GetSimpleCMS/GetSimpleCMS
gharlan/alfred-github-workflow
ghedipunk/PHP-Websockets
ghost1227/historical-redux2
ghunti/HighchartsPHP
giggsey/libphonenumber-for-php
gilbitron/Arrest-MySQL
gilbitron/PHP-SimpleCache
gilbitron/PIP
gilbitron/WordPress-Settings-Framework
gimler/symfony-rest-edition
giorgiosironi/phpunit-selenium
gitonomy/gitlib
gitonomy/gitonomy
gizburdt/cuztom
gjedeer/celery-php
glamorous/TMDb-PHP-API
gleez/cms
goaop/framework
gocart/GoCart
goodby/csv
googleglass/mirror-quickstart-php
GordonLesti/Lesti_Fpc
gorhill/PHP-FineDiff
GotCms/GotCms
Goteo/Goteo
GovHub/CensusShapeConverter
gpbmike/PHP-YUI-Compressor
Grandt/PHPePub
Graphite-Tattle/Tattle
graulund/tweetnest
gree/Orion
gregdingle/genetify
greggilbert/recaptcha
Gregwar/Captcha
Gregwar/CaptchaBundle
Gregwar/Image
Gregwar/ImageBundle
GSA/data.gov
guilhermeblanco/zendframework1-doctrine2
guzzle/guzzle
habari/habari
habari/system
hafriedlander/php-peg
harrydeluxe/php-liquid
haschek/PubwichFork
haseydesign/flexi-auth
hearsayit/HearsayRequireJSBundle
hellogerard/jobby
henrikbjorn/Lurker
Herzult/HerzultForumBundle
Herzult/php-ssh
Herzult/SimplePHPEasyPlus
heyitspavel/fitbitphp
hfcorriez/pagon
hightman/pspider
hightman/scws
hightman/xunsearch
hipchat/hipchat-php
hlashbrooke/WordPress-Plugin-Template
hoaproject/Console
hoaproject/Ruler
hoaproject/Websocket
horde/horde
hownowstephen/php-foursquare
hugodias/cakeStrap
humanmade/backupwordpress
humanmade/Colors-Of-Image
humanmade/Custom-Meta-Boxes
humanmade/WPThumb
hunk/Magic-Fields
hwi/HWIOAuthBundle
hybridauth/hybridauth
hypery2k/owncloud
hzlzh/Alfred-Workflows
iamcal/oembed
iamcal/php-emoji
ianckc/CodeIgniter-Instagram-Library
iandunn/WordPress-Plugin-Skeleton
ichikaway/cakephp-mongodb
idiot/Spiffing
idno/Known
ifsnop/mysqldump-php
IgnitedDatatables/Ignited-Datatables
igorw/ConfigServiceProvider
igorw/doucheswag
igorw/evenement
igorw/IgorwFileServeBundle
igorw/yolo
igstan/learn-you-some-erlang-kindle
iliaal/php_excel
illuminate/database
illuminate/html
imanee/imanee
imbo/imbo
immobiliare/ApnsPHP
impressivewebs/CSS3-Click-Chart
impresspages/ImpressPages
Incenteev/ParameterHandler
indeyets/appserver-in-php
indeyets/pake
indieteq/PHP-MySQL-PDO-Database-Class
infinitas/infinitas
inspirer/mibew
intaro/pinboard
interconnectit/my-eyes-are-up-here
interconnectit/Search-Replace-DB
InterNations/http-mock
Intervention/image
Intervention/imagecache
In-Touch/laravel-newrelic
ionize/ionize
ircmaxell/filterus
ircmaxell/monad-php
ircmaxell/password_compat
ircmaxell/PHP-CryptLib
ircmaxell/PHP-PasswordLib
ircmaxell/PHPPHP
ircmaxell/RandomLib
isaacsu/twich
isotope/core
itsgoingd/clockwork
ivanakimov/hashids.php
ivanweiler/Inchoo_Facebook
ivkos/Pushbullet-for-PHP
iwind/rockmongo
iwyg/jitimage
j4mie/idiorm
j4mie/paris
J7mbo/twitter-api-php
jackalope/jackalope
jacwright/RestServer
jadell/neo4jphp
jakajancar/DropboxUploader
jakubledl/dissect
JakubOnderka/PHP-Parallel-Lint
Jalle19/xbmc-video-server
jamesgpearce/modernizr-server
JamesHeinrich/getID3
jamesiarmes/php-ews
jamierumbelow/codeigniter-base-controller
jamierumbelow/codeigniter-base-model
jamierumbelow/pigeon
janmarek/WebLoader
janodvarko/harviewer
jarednova/timber
Jasig/phpCAS
jasongrimes/silex-simpleuser
jasonhinkle/phreeze
jasonlewis/basset
jasonlewis/enhanced-router
jasonlewis/expressive-date
jasonlewis/resource-watcher
javiereguiluz/Cupon
jaxl/JAXL
jayli/combo

jaysalvat/image2css
jaytaph/HTRouter
jaz303/phake
jbroadway/analog
jbroadway/elefant
jbroadway/phpactiveresource
jbroadway/urlify
jchristopher/attachments
JDare/ClankBundle
jdp/redisent
jdp/twitterlibphp
jeckman/YouTube-Downloader
JeffreyWay/Easy-WordPress-Custom-Post-Types
JeffreyWay/Laravel-4-Generators
JeffreyWay/Laravel-Model-Validation
JeffreyWay/Laravel-Test-Helpers
jenssegers/codeigniter-advanced-images
jenssegers/codeigniter-hmvc-modules
jenssegers/laravel-date
jenssegers/laravel-mongodb
jenssegers/php-proxy
jeremeamia/super_closure
jeremyclark13/automatic-theme-plugin-update
jeremyFreeAgent/Bitter
jeremykendall/php-domain-parser
jeromevdl/android-holo-colors
jfoucher/flickholdr
jgrossi/corcel
jigoshop/jigoshop
jimdoescode/CodeIgniter-Dropbox-API-Library
jimdoescode/CodeIgniter-YouTube-API-Library
jim/fitzgerald
jimmykane/The-Three-Little-Pigs-Siri-Proxy
jimrubenstein/php-profiler
jjgrainger/wp-custom-post-type-class
jmathai/epiphany
jmathai/foursquare-async
jmathai/php-multi-curl
jmathai/twitter-async
jmespath/jmespath.php
jmstriegel/php.googleplusapi
joelcox/codeigniter-redis
johmue/mysql-workbench-schema-exporter
johnbillion/extended-cpts
johnbillion/query-monitor
joindin/joind.in
jokkedk/ZFDebug
jolicode/JoliTypo
jonaswouters/XhprofBundle
jonathangeiger/kohana-jelly
joomla/joomla-cms
joomla/joomla-framework
joomla/joomla-platform
josegonzalez/cakephp-upload
josegonzalez/php-git
JosephLenton/PHP-Error
joshcam/PHP-MySQLi-Database-Class
joshdick/miniProxy
jpfuentes2/php-activerecord
jquery/jquery-wp-content
jquery/testswarm
jrbasso/MeioUpload
jrconlin/oauthsimple
jreinke/magento-elasticsearch
jsebrech/php-o
jstayton/Miner
jtopjian/gluephp
jublonet/codebird-php
justinrainbow/json-schema
justintadlock/hybrid-base
justintadlock/hybrid-core
justinwalsh/daux.io
jwage/easy-csv
jwage/php-mongodb-admin
jwage/purl
JWHennessey/phpInsight
kakserpom/phpdaemon
kallaspriit/Cassandra-PHP-Client-Library
kamisama/Cake-Resque
kamisama/Fresque
kamisama/php-resque-ex
kasparsd/minit
katzgrau/KLogger
kbjr/Git.php
kbsali/php-redmine-api
kellan/pinterest.api.php
kerns/dummy
Kerrick/readability-js
kevinlebrun/colors.php
Khan/khan-api
khoaofgod/phpfastcache
kimai/kimai
klaussilveira/gitter
klevo/wildflower
klokantech/tileserver-php
KnpLabs/DoctrineBehaviors
KnpLabs/Gaufrette
KnpLabs/KnpBundles
KnpLabs/KnpGaufretteBundle
KnpLabs/KnpIpsum
KnpLabs/KnpMarkdownBundle
KnpLabs/KnpMenu
KnpLabs/KnpMenuBundle
KnpLabs/KnpPaginatorBundle
KnpLabs/KnpSnappyBundle
KnpLabs/KnpTimeBundle
KnpLabs/marketplace
KnpLabs/php-github-api
KnpLabs/snappy
koconder/android-market-api-php
KodiCMS-Kohana/cms
kohana/auth
kohana/core
kohana/database
kohana/kohana
kohana/minion
kohana/orm
kohana/unittest
kohana/userguide
kohkimakimoto/altax
kolber/stacey
komarserjio/notejam
komola/Bootstrap-Zend-Framework
koraktor/steam-condenser-php
koto/phar-util
kriansa/openboleto
kriswallsmith/assetic
kriswallsmith/Buzz
kriswallsmith/spork
Kroc/NoNonsenseForum
kronusme/dota2-api
K-S-V/Scripts
ktamas77/firebase-php
Kunena/Kunena-Forum
kvz/cakephp-rest-plugin
kvz/system_daemon
kylereicks/picturefill.js.wp
lanthaler/JsonLD
laravelbook/ardent
laravelbook/laravel4-phpstorm-helper
laravelbook/laravel4-sublimetext-helper
laravel/framework
laravel/laravel
lastguest/mu
laurencedawson/embr
ldleman/Leed
leafo/lessphp
leafo/scssphp
LeaseWeb/LswMemcacheBundle
LeaVerou/rgba.php
Leeflets/leeflets
leemason/NHP-Theme-Options-Framework
lencioni/SLIR
LeonardoCardoso/Facebook-Link-Preview
leroy-merlin-br/mongolid-laravel
lexik/LexikFormFilterBundle
lexik/LexikMaintenanceBundle
lexik/LexikTranslationBundle
LExpress/symfony1
lichtner/fluentpdo
liebig/cron
ligboy/Wechat-php
liip/LiipCacheControlBundle
liip/LiipFunctionalTestBundle
liip/LiipHelloBundle
liip/LiipImagineBundle
liip/LiipMonitorBundle
liip/LiipThemeBundle

liip/php-osx
liip/RMT
LimeSurvey/LimeSurvey
lisphp/lisphp
liu21st/extend
liu21st/thinkphp
liuggio/ExcelBundle
liuggio/StatsDClientBundle
LiveHelperChat/livehelperchat
livestreet/livestreet
lkwdwrd/git-deploy
loadsys/twitter-bootstrap-helper
loic-sharma/profiler
lonnieezell/codeigniter-forensics
lphuberdeau/Neo4j-PHP-OGM
lsolesen/pel
lstrojny/functional-php
ludovicchabant/PieCrust
Lullabot/drupal-boilerplate
lunaru/MongoRecord
lunetics/LocaleBundle
Lusitanian/PHPoAuthLib
lyrixx/Silex-Kitchen-Edition
m4tthumphrey/php-gitlab-api
mac-cain13/notificato
machuga/authority
machuga/authority-l4
macuenca/Instagram-PHP-API
madalinoprea/magneto-debug
madalinoprea/magneto-varnish
mage-eag/mage-enhanced-admin-grids
magento-ecg/coding-standard
magento/magento2
magento/taf
mageplus/mageplus
magic-fields-team/Magic-Fields-2
mako-framework/framework
malyshev/yii-debug-toolbar
mandango/mandango
manifestinteractive/easyapns
mantisbt/mantisbt
mapkyca/ifttt-webhook
marcelog/PAMI
marcj/php-rest-service
marcoarment/secondcrack
marco-fiset/Testify.php
markjaquith/WordPress
markjaquith/WordPress-Skeleton
markjaquith/WP-Stack
markjaquith/WP-TLC-Transients
markomarkovic/simple-php-git-deploy
markstory/acl_extras
markstory/asset_compress
markuspoerschke/iCal
martinbean/api-framework
Mashape/unirest-php
masterexploder/PHPThumb
Masterminds/html5-php
mathiasbynens/php-url-shortener
mathiasverraes/money
mattab/trello-backup
mattbanks/Genesis-Starter-Child-Theme
mattbanks/WordPress-Starter-Theme
matteosister/GitElephant
mattg888/GCM-PHP-Server-Push-Message
MatthewRuddy/Wordpress-Timthumb-alternative
matthiasmullie/minify
mattpass/ICEcoder
mattstauffer/Simple-RESS
mauricesvay/php-facedetection
maximebf/php-debugbar
maxmind/GeoIP2-php
maxmind/geoip-api-php
mboynes/super-cpt
Medalink/laravel-blade
medialab/iwanthue
MediovskiTechnology/php-crontab-manager
meeech/shopify.tmbundle
meenie/munee
mewebstudio/captcha
mewebstudio/Purifier
mexitek/phpColors
mgibbs189/custom-field-suite
mheap/Silex-Extensions
mhoofman/wordpress-heroku
mibe/FeedWriter
michaeldewildt/wordpress-backup-to-dropbox
michael-romer/zf-boilerplate
michelf/php-markdown
michelsalib/BCCCronManagerBundle
michelsalib/BCCExtraToolsBundle
microweber/microweber
mikecao/flight
mikecao/sparrow
mikegogulski/bitcoin-php
mikehaertl/phpwkhtmltopdf
mikeland86/graphp
mikelbring/tinyissue
mikemand/logviewer
mikey179/vfsStream
MikoMagni/Alfred-for-Trello
miled/wordpress-social-login
milesj/decoda
milesj/forum
milesj/uploader
MiniCodeMonkey/Vagrant-LAMP-Stack
miniflux/miniflux
minimaldesign/mHTML.tmbundle
minkphp/Mink
misd-service-development/phone-number-bundle
mishamx/yii-user
MISP/MISP
mixu/useradmin
mjaschen/phpgeo
mledoze/countries
mlively/Phake
mmoreram/GearmanBundle
modolabs/Kurogo-Mobile-Web
modxcms/evolution
modxcms/revolution
mojeda/ServerStatus
moltin/laravel-cart
mongodb/mongo-php-driver
MongoDB-Rox/phpMoAdmin-MongoDB-Admin-Tool-for-PHP
moodle/moodle
morgan/kohana-restify
morrisonlevi/Ardent
mpalmer/jekyll-static-comments
Mparaiso/Silex-Blog-App
MPOS/php-mpos
mptre/php-soundcloud
MrJuliuss/syntara
MrRio/shellwrap
msgpack/msgpack-php
msurguy/laravel-ajax-example
mtdowling/cron-expression
mtibben/html2text
murtaugh/HTML5-Reset-WordPress-Theme
mustangostang/spyc
mwillbanks/Zend_Mobile
mwunsch/thimble
mybb/mybb
myclabs/php-enum
mzsanford/twitter-text-php
namshi/jose
namshi/notificator
nategood/commando
nategood/httpful
nathanstaines/starkers-html5
navruzm/lmongo
nb/wordpress-tests
Needlworks/Textcube
neitanod/forceutf8
nekudo/php-websocket
nelmio/alice
nelmio/NelmioApiDocBundle
nelmio/NelmioCorsBundle
nelmio/NelmioJsLoggerBundle
nelmio/NelmioSecurityBundle
nelmio/NelmioSolariumBundle
NeonHorizon/berryio
nervetattoo/elasticsearch
netputer/netputweets
netputer/wechat-php-sdk
nette/latte
nette/nette
nette/tester
nette/tracy

netz98/n98-magerun
newsapps/wordpress-mtv
nexcess/magento-turpentine
nfephp-org/nfephp
nganhtuan63/GXC-CMS
niallkennedy/open-graph-protocol-tools
nicmart/Tree
nicokaiser/php-websocket
niemanlab/openfuego
nikic/iter
nikic/PHP-Parser
nikic/scalar_objects
nikkobautista/laravel-tutorial
niklasvh/php.js
nilsteampassnet/TeamPass
nixsolutions/yandex-php-library
njh/easyrdf
Nodge/yii-eauth
nrk/predis
nrk/predis-async
nuovo/spreadsheet-reader
nZEDb/nZEDb
obenland/the-bootstrap
Ocramius/ProxyManager
olamedia/nokogiri
ollierattue/FormIgniter
omarabid/Self-Hosted-WordPress-Plugin-repository
omeka/Omeka
onelogin/php-saml
onlinecity/php-smpp
oott123/bpcs_uploader
opauth/opauth
opencart-ce/opencart-ce
opencart/opencart
opencfp/opencfp
OpendataCH/Transport
openemr/openemr
openfootball/world-cup
opengovfoundation/madison
opengovplatform/opengovplatform-DMS
openid/php-openid
OpenMage/magento-mirror
Openovate/eden
openpne/OpenPNE3
openscholar/openscholar
opensky/OpenSkyRuntimeConfigBundle
opensolutions/ViMbAdmin
opentape/opentape
OrayDev/tudu-web
orchestral/platform
orchestral/testbench
orderly/codeigniter-paypal-ipn
organicinternet/magento-configurable-simple
ornicar/lichess-old
ornicar/php-github-api
ornicar/php-git-repo
ornicar/php-user-agent
orno/di
orocrm/crm
orocrm/crm-application
orocrm/platform
orocrm/platform-application
osalabs/phpminiadmin
OSAS/strapping-mediawiki
oscarotero/Embed
oscarotero/imagecow
osclass/Osclass
osCommerce/oscommerce
osCommerce/oscommerce2
osTicket/osTicket-1.7
outlandishideas/wpackagist
Overv/Open.GL
OWASP/phpsec
owncloud/calendar
owncloud/core
owncloud/music
owncloud/news
owncloud/notes
oyejorge/gpEasy-CMS
padams/Open-Web-Analytics
padraic/mockery
padraic/mutagenesis
panique/huge
PANmedia/raptor-editor
parisholley/wordpress-fantastic-elasticsearch
partkeepr/PartKeepr
patricktalmadge/bootstrapper
pat/riddle
patrikf/glip
pattern-lab/patternlab-php
paulrobertlloyd/barebones
paulund/wordpress-theme-customizer-custom-controls
Pawka/phrozn
paypal/ipn-code-samples
paypal/merchant-sdk-php
paypal/PayPal-PHP-SDK
Payum/Payum
Payum/PayumBundle
pda/flexihash
pda/pheanstalk
pdepend/pdepend
PeeHaa/OpCacheGUI
peej/phpdoctor
peej/tonic
pengkong/A3M-for-CodeIgniter-2.0
peteboere/css-crush
petewarden/ParallelCurl
pfsense/pfsense-packages
Ph3nol/NotificationPusher
phacility/arcanist
phacility/libphutil
phacility/phabricator
phacility/xhprof
phalcon/cphalcon
PhalconEye/cms
phalcon/forum
phalcon/incubator
phalcon/invo
phalcon/mvc
phalcon/phalcon-devtools
phalcon/vokuro
phastlight/phastlight
phayes/geoPHP
PhenX/php-font-lib
phergie/phergie
philipbjorge/Infinite-Social-Wall
philipithomas/cv-philipithomas
philippe/FrogCMS
philippK-de/Collabtive
philsturgeon/codeigniter-cli
philsturgeon/codeigniter-curl
philsturgeon/codeigniter-oauth2
philsturgeon/codeigniter-restclient
philsturgeon/codeigniter-template
philsturgeon/fuel-ninjauth
phingofficial/phing
phly/PhlyRestfully
phpbb/phpbb
phpbrew/phpbrew
phpcr/phpcr
PHP-DI/PHP-DI
phpDocumentor/phpDocumentor2
PHP-FFMpeg/PHP-FFMpeg
php-fig/log
phpfreak/Project-Pier
phpfunk/alfred-spotify-controls
PHPGangsta/GoogleAuthenticator
PHPIDS/PHPIDS
p-h-p/instagraph
PHPixie/Project
PHPMailer/PHPMailer
phpmd/phpmd
phpmo/php.mo
phpmyadmin/phpmyadmin
phpnode/YiiRedis
PHPOffice/PHPExcel
PHPOffice/PHPPowerPoint
PHPOffice/PHPWord
phppgadmin/phppgadmin
phpseclib/phpseclib
phpsec/phpSec
phpspec/phpspec
phpspec/phpspec2-proof-of-concept
phpspec/prophecy
phpsysinfo/phpsysinfo
php-vcr/php-vcr
php/web-php
phpwind/windframework

phundament/app
picocms/Pico
pimcore/pimcore
pippinsplugins/WP-Logging
piwik/piwik
pixelandtonic/ContactForm
pkhamre/wp-varnish
pkp/ojs
plancake/official-library-php-email-parser
Pligg/pligg-cms
pluspeople/pesaPi
PocketMine/PocketMine-MP
podio/podio-php
podlove/podlove-publisher
pods-framework/pods
polyfractal/athletic
polyfractal/sherlock
pornel/PHPTAL
potsky/PimpMyLog
powder96/numbers.php
poweradmin/poweradmin
ppi/framework
PredictionIO/PredictionIO-PHP-SDK
preinheimer/xhprof
pressbooks/pressbooks
pressflow/6
pressflow/7
PressWork/PressWork
prestaconcept/PrestaSitemapBundle
PrestaShop/PrestaShop
PrestaShop/PrestaShop-modules
Problematic/ProblematicAclManagerBundle
processing/processing-web-archive
Program-O/Program-O
projectfork/Projectfork
project-open-data/csv-to-api
project-open-data/db-to-api
propelorm/Propel
propelorm/Propel2
propelorm/PropelBundle
propelorm/sfPropelORMPlugin
psistorm/alfredapp
psliwa/PdfBundle
psliwa/PHPPdf
psugand/CodeIgniter-S3
psynaptic/php-drupal.tmbundle
PUGX/badge-poser
punbb/punbb
purekid/mongodm
pusher/pusher-http-php
pydio/pydio-core
pyrocms/pyrocms
q2a/question2answer
QafooLabs/php-refactoring-browser
Qafoo/review
qiniu/php-sdk
quickapps/cms
Quixotix/PHP-PayPal-IPN
quizlet/oauth2-php-closed-source
rachelbaker/bootstrapwp-Twitter-Bootstrap-for-WordPress
rachelbaker/Font-Awesome-WordPress-Plugin
rackspace/php-opencloud
RackTables/racktables
radiosilence/Ham
radishconcepts/WordPress-GitHub-Plugin-Updater
rainphp/raintpl
rainphp/raintpl3
ralphschindler/NOLASnowball
ramsey/array_column
ramsey/uuid
randyjensen/handcrafted-wp-theme
rasmusbergpalm/jslate
ratchetphp/Ratchet
raulfraile/ladybug
raulfraile/LadybugBundle
raveren/kint
rchouinard/phpass
rcrowe/TwigBridge
rdlowrey/auryn
reactphp/gifsocket
reactphp/promise
reactphp/react
reactphp/zmq
recess/recess
recoilphp/recoil
redaxmedia/redaxscript
redbaron76/PongoCMS-Laravel-cms-bundle
regru/php-whois
researchgate/broker
Respect/Relational
Respect/Rest
Respect/Validation
Retina-Images/Retina-Images
rgrove/jsmin-php
richardshepherd/TwentyTenFive
richbradshaw/CSS-Transitions-Transforms-and-Animation
richsage/RMSPushNotificationsBundle
richthegeek/phpsass
rilwis/meta-box
RJ/playdar-core
rlerdorf/opcache-status
rlerdorf/WePloy
rmccue/Requests
robclancy/presenter
robmorgan/phinx
rocketeers/rocketeer
rodneyrehm/CFPropertyList
ronanguilloux/IsoCodes
ronanguilloux/php-gpio
roots/sage
roots/soil
roots/wp-h5bp-htaccess
rossriley/phrocco
RosYama/RosYama.2
RoumenDamianoff/laravel-feed
RoumenDamianoff/laravel-sitemap
roundcube/roundcubemail
rsms/gitblog
rtablada/package-installer
rubensayshi/gw2spidy
ruckus/ruckusing-migrations
ruflin/Elastica
runekaagaard/snowscript
rwarasaurus/nano
RWOverdijk/AssetManager
ryancramerdesign/ProcessWire
rydurham/L4withSentry
rynop/CakePlate
sabberworm/PHP-CSS-Parser
salsify/jsonstreamingparser
samacs/simple_html_dom
Sammaye/MongoYii
sams/Thematic-html5boilerplate
sanchothefat/wp-less
sapienza/CodeIgniter-admin-panel
satooshi/php-coveralls
savetheinternet/Tinyboard
sayakb/sticky-notes
sbisbee/sag
Scalr/scalr
Schepp/CSS-JS-Booster
schickling/git-s3
schickling/laravel-backup
schmittjoh/JMSAopBundle
schmittjoh/JMSDebuggingBundle
schmittjoh/JMSDiExtraBundle
schmittjoh/JMSI18nRoutingBundle
schmittjoh/JMSJobQueueBundle
schmittjoh/JMSPaymentCoreBundle
schmittjoh/JMSPaymentPaypalBundle
schmittjoh/JMSSecurityExtraBundle
schmittjoh/JMSSerializerBundle
schmittjoh/JMSTranslationBundle
schmittjoh/metadata
schmittjoh/php-collection
schmittjoh/php-option
schmittjoh/serializer
schmittjoh/twig.js
scottgonzalez/grunt-wordpress
scottmac/opengraph
ScottSmith95/Decode
scribu/wp-posts-to-posts
scribu/wp-scb-framework
Scriptor/pharen
scrutinizer-ci/php-analyzer
scrutinizer-ci/scrutinizer
sculpin/sculpin
seatgeek/djjob

sebastianbergmann/dbunit
sebastianbergmann/diff
sebastianbergmann/hhvm-wrapper
sebastianbergmann/money
sebastianbergmann/php-code-coverage
sebastianbergmann/phpcpd
sebastianbergmann/phpdcd
sebastianbergmann/phploc
sebastianbergmann/phpunit
sebastianbergmann/phpunit-mock-objects
sebgiroux/Cassandra-Cluster-Admin
seblucas/cops
sebsauvage/Shaarli
seedifferently/the-great-web-framework-shootout
Seldaek/jsonlint
Seldaek/monolog
Seldaek/php-console
Self-Evident/OneFileCMS
semsol/arc2
sendgrid/sendgrid-php
sensiolabs/security-checker
sensiolabs/SensioFrameworkExtraBundle
sensiolabs/SensioGeneratorBundle
sequelpro/Bundles
serbanghita/Mobile-Detect
sergejey/majordomo
sergeychernyshev/showslow
servergrove/ServerGroveLiveChat
servergrove/TranslationEditorBundle
sesser/Instaphp
Shadez/wowarmory
shadowhand/email
shadowhand/pagination
shameerc/TextPress
shaneharter/PHP-Daemon
ShawnMcCool/laravel-form-base-model
shenzhe/zphp
shoestrap/shoestrap-3
shopware/shopware
shrikeh/teapot
shuber/curl
Shumkov/Rediska
sidneywidmer/Latchet
silexphp/Pimple
silexphp/Silex
silexphp/Silex-Skeleton
silexphp/Silex-WebProfiler
silverstripe/silverstripe-cms
silverstripe/silverstripe-framework
silverstripe/silverstripe-installer
silverstripe/silverstripe-userforms
simfatic/RegistrationForm
simplebits/Pears
simpleinvoices/simpleinvoices
SimpleMachines/SMF2.1
simplepie/simplepie
simplethemes/skeleton_wp
simplethings/EntityAudit
simplethings/SimpleThingsFormExtraBundle
simshaun/recurr
sitecake/sitecake
sittercity/sprig
sjlu/CodeIgniter-Bootstrap
slimphp/Slim
slimphp/Slim-Skeleton
slimphp/Slim-Views
slywalker/cakephp-plugin-boost_cake
slywalker/TwitterBootstrap
SmItH197/SteamAuthentication
snc/SncRedisBundle
snytkine/LampCMS
SocalNick/ScnSocialAuth
sofadesign/limonade
solariumphp/solarium
somerandomdude/Frank
somewhereYu/OSAdmin
sonata-project/sandbox
sonata-project/SonataAdminBundle
sonata-project/SonataDoctrineORMAdminBundle
sonata-project/SonataMediaBundle
sonata-project/SonataNewsBundle
sonata-project/SonataPageBundle
sonata-project/SonataUserBundle
sonnyt/Tweetie
soonick/poMMo
sourcefabric/airtime
sourcefabric/Newscoop
spadefoot/kohana-orm-leap
Spea/SpBowerBundle
speedmax/h2o-php
splitbrain/dokuwiki
splorp/tersus
sqlmapproject/testenv
squizlabs/PHP_CodeSniffer
SSilence/selfoss
stackphp/builder
StanScates/Tweet.js-Mod
startbbs/startbbs
statedecoded/statedecoded
stecman/symfony-console-completion
stephpy/timeline-bundle
stfalcon/TinymceBundle
stil/curl-easy
stof/StofDoctrineExtensionsBundle
stojg/crop
stormuk/Gravity-Forms-ACF-Field
storytlr/storytlr
strangerstudios/paid-memberships-pro
straup/parallel-flickr
stripe/stripe-php
stripe/wilde-things
subtlepatterns/SubtlePatterns
sugarcrm/sugarcrm_dev
suin/php-rss-writer
suncat2000/MobileDetectBundle
super3/IRC-Bot
swiftmailer/swiftmailer
swoole/framework
syamilmj/Aqua-Page-Builder
syamilmj/Aqua-Resizer
Sybio/GifCreator
Sybio/ImageWorkshop
sydlawrence/alfred-dev-doctor
Sylius/Sylius
symfony2admingenerator/AdmingeneratorGeneratorBundle
symfony/AsseticBundle
symfony/ClassLoader
symfony-cmf/cmf-sandbox
symfony/Console
symfony/DomCrawler
symfony/HttpFoundation
symfony/Process
symfony/symfony
symfony/symfony1
symfony/symfony-standard
symfony/Yaml
symphonycms/symphony-2
Synchro/PHPMailer
szajbus/uploadpack
szjani/predaddy
t0k4rt/phpqrcode
tamagokun/pomander
tammyhart/Reusable-Custom-WordPress-Meta-Boxes
TankAuth/Tank-Auth
tappleby/laravel-auth-token
tareq1988/wordpress-settings-api-class
tareq1988/wp-project-manager
tchwork/utf8
tcpdf-clone/tcpdf
tcz/PHPTracker
TechEmpower/FrameworkBenchmarks
technosophos/querypath
tedious/Fetch
tedious/JShrink
tedious/Stash
teepluss/laravel-theme
teqneers/PHP-Stream-Wrapper-for-Git
textile/php-textile
textmate/php.tmbundle
textpattern/textpattern
TGMPA/TGM-Plugin-Activation
thebuggenie/thebuggenie
TheFootballSocialClub/FSCHateoasBundle
thelia/thelia
themattharris/tmhOAuth
themeskult/wp-svbtle
thenbrent/paypal-digital-goods
thenextweb/TNW-Social-Count

thephpleague/factory-muffin
thephpleague/html-to-markdown
thephpleague/monga
thephpleague/oauth2-client
thephpleague/oauth2-server
thephpleague/shunt
thephpleague/statsd
ThePixelDeveloper/kohana-sitemap
there4/markdown-resume
theseer/Autoload
theseer/phpdox
thethemefoundry/twentytwelve
thethemefoundry/wordpress-capistrano
thewirelessguy/cornerstone
thiagoalessio/tesseract-ocr-for-php
thibaud-rohmer/PhotoShow
ThinkUpLLC/ThinkUp
thobbs/phpcassa
thomasbachem/php-short-array-syntax-converter
thomashempel/AlfredGoogleTranslateWorkflow
thomseddon/cakephp-oauth-server
thorsten/phpMyFAQ
thujohn/analytics-l4
thujohn/pdf-l4
thujohn/twitter
thyseus/yii-user-management
tijsverkoyen/CssToInlineStyles
timwhitlock/php-varnish
tj/php-selector
tlack/snaphax
tlhunter/neoinvoice
tlhunter/spidermonkey
tnc/php-amqplib
TobiasBg/TablePress
Toddish/Verify-L4
toddmotto/html5blank
toin0u/DigitalOcean
toin0u/Geotools-laravel
tokudu/PhpMQTTClient
tollmanz/wordpress-pecl-memcached-object-cache
tombenner/wp-mvc
TomBZombie/Dice
tomcreighton/Glider
tommcfarlin/page-template-example
tommcfarlin/WordPress-Settings-Sandbox
tommcfarlin/WordPress-Widget-Boilerplate
tomschlick/memcached-library
tontof/kriss_feed
tonydewan/Carabiner
tonydspaniard/Yii-extensions
toopay/gas-orm
topdown/phpStorm-CC-Helpers
torrage/Torrage
tplaner/When
tpyo/amazon-s3-php-class
travis-ci-examples/php
Trismegiste/Mondrian
troydavisson/PHRETS
tschoffelen/PHP-Passkit
tschoffelen/PHP-PKPass
tumblr/tumblr.php
TurbineCSS/Turbine
twigphp/Twig
twigphp/Twig-extensions
twilio/OpenVBX
twilio/twilio-php
twip/twip
twittem/wp-bootstrap-navwalker
tylerhall/php-growl
tylerhall/Shine
tylerhall/simple-php-framework
tylerhall/sosumi
typecho/framework
TYPO3/TYPO3.CMS
UCF/Theme-Updater
UnionOfRAD/framework
UnionOfRAD/lithium
UpThemes/UpThemes-Framework
ushahidi/Swiftriver-2011
ushahidi/Ushahidi_Web
usmanhalalit/pixie
uzyn/cakephp-opauth
vafour/vafpress-framework
valendesigns/option-tree
vanilla/vanilla
varspool/Wrench
vdesabou/alfred-spotify-mini-player
veloper/WordPress-Domain-Changer
vendo/vendo
VentureCraft/revisionable
vespakoen/authority-laravel
vespakoen/menu
vesparny/codeigniter-html5boilerplate-twitter-bootstrap
vesparny/silex-simple-rest
vespolina/vespolina-sandbox
Vheissu/Ci-Smarty
vichan-devel/vichan
victorstanciu/Wikitten
videlalvaro/php-amqplib
videlalvaro/RabbitMqBundle
videlalvaro/Thumper
Videola/videola
vimeo/vimeo-api-examples
vimeo/vimeo.php
vimeo/vimeo-php-lib
vim-php/phpctags
Vinai/groupscatalog2
virtphp/virtphp
visualidiot/Spiffing
VisualPHPUnit/VisualPHPUnit
vito/chyrp
vladgh/VladGh.com-LEMP
vladkens/VK
vlucas/bulletphp
vlucas/phpDataMapper
vlucas/phpdotenv
vlucas/valitron
voceconnect/thermal-api
vova07/yii2-start
vqmod/vqmod
vrana/adminer
vrana/notorm
wallabag/wallabag
wanze/Google-Analytics-API-PHP
wearerequired/required-foundation
web2project/web2project
webasyst/webasyst-framework
WebDevStudios/Custom-Metaboxes-and-Fields-for-WordPress
WebDevStudios/custom-post-type-ui
WebDevStudios/StartBox
WebTales/rubedo
webtechnick/CakePHP-Facebook-Plugin
webtechnick/CakePHP-FileUpload-Plugin
welaika/wordless
welovewordpress/SublimePhpTidy
wes/phpimageresize
WhatCD/Gazelle
whatthejeff/breeze
whatthejeff/nyancat-phpunit-resultprinter
WhichBrowser/WhichBrowser
WhiteHouse/petitions
whiteoctober/Pagerfanta
whiteoctober/WhiteOctoberPagerfantaBundle
whizark/php-patterns
widmogrod/zf2-assetic-module
wikimedia/mediawiki
wikireader/wikireader
willdurand/BazingaFakerBundle
willdurand/EmailReplyParser
willdurand/Hateoas
willdurand/Negotiation
willdurand/Propilex
wimg/PHPCompatibility
Wisembly/elephant.io
Wixel/GUMP
wmark/CDN-Linker
wolfcms/wolfcms
WoltLab/WCF
woothemes/woocommerce
WordPress-Coding-Standards/WordPress-Coding-Standards
WordPress/WordPress
Wouterrr/MangoDB
WP-API/WP-API
wpbrasil/odin
wp-cli/php-cli-tools
wp-cli/wp-cli
wpninjas/ninja-forms
WPO-Foundation/webpagetest

wsdl2phpgenerator/wsdl2phpgenerator
X2Engine/CRM
XaminProject/handlebars.php
XCMer/larry-four-generator
xdebug/xdebug
Xeoncross/DByte
Xeoncross/forumfive
Xeoncross/micromvc
xianglei/easyhadoop
xiaosier/libweibo
xPaw/PHP-Minecraft-Query
xPaw/PHP-Source-Query
xpressengine/xe-core
xw2423/nForum
YahnisElsts/plugin-update-checker
YahnisElsts/wp-update-server
yandod/candycane
yandod/php5-nginx-vagrant-sample
yellowflag/cribbb
yi12345/TravianZ
yiiext/nested-set-behavior
yiiext/with-related-behavior
yiisoft/yii
yiisoft/yii2
Yoast/wordpress-seo
yohang/CalendR
yohang/Finite
yoozi/swf-docs-generator
YOURLS/YOURLS
yugene/Gearman-Monitor
yuguo/33pu
yupe/yupe
zamoose/themehookalliance
zencoder/html5-boilerplate-for-wordpress
zendframework/modules.zendframework.com
zendframework/ZendDeveloperTools
zendframework/ZendSkeletonModule
zendframework/zf1
zendframework/zf2
zendframework/ZFTool
zengchao/MOMO_SERVER
zenphoto/zenphoto
ZF-Commons/ZfcAdmin
ZF-Commons/ZfcBase
ZF-Commons/zfc-rbac
ZF-Commons/ZfcUser
zikula/core
zircote/swagger-php
Zizaco/confide
Zizaco/entrust
Znarkus/postmark-php
zombor/KOstache
ZoneMinder/ZoneMinder
zordius/lightncandy
zpanel/zpanelx
zscorpio/weChat

References

[Adida 2008] Ben Adida. “Helios: Web-based Open-Audit Voting”. In: *Proceedings of the 17th USENIX Security Symposium (USENIX Security ’08)*. USENIX Association, July 2008, pages 335–348 (Cited on pages 3, 49, 54, 64, 76, 78).

[Adida 2009] Ben Adida. *Helios Election System*. 2009. URL: https://github.com/benadida/helios-server (visited on 30th March 2017) (Cited on page 54).

[Adida 2013] Ben Adida. *Princeton Undergraduate Elections*. 2013. URL: https://princeton.heliosvoting.org/ (visited on 30th March 2017) (Cited on page 49).

[Adida *et al.* 2009] Ben Adida, Olivier de Marneffe, Olivier Pereira and Jean-Jacques Quisquater. “Electing a University President Using Open-Audit Voting: Analysis of Real-World Use of Helios”. In: *Proceedings of the 2009 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections (EVT/WOTE ’09)*. USENIX Association, August 2009 (Cited on pages 49, 54–56).

[Aho *et al.* 2006] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd edition. Addison-Wesley Longman Publishing Co., Inc., September 2006. ISBN: 0321486811 (Cited on pages 25, 27, 29, 30, 38, 41, 42).

[Allen 1970] Frances E. Allen. “Control Flow Analysis”. In: *Proceedings of a Symposium on Compiler Optimization*. ACM, July 1970, pages 1–19 (Cited on page 33).

[Allen & Cocke 1972] Frances E. Allen and John Cocke. *Graph-theoretic Constructs for Program Flow Analysis*. Technical report. IBM Thomas J. Watson Research Center, July 1972 (Cited on page 38).

[Alrabaee *et al.* 2015] Saed Alrabaee, Paria Shirani, Lingyu Wang and Mourad Debbabi. “SIGMA: A Semantic Integrated Graph Matching Approach for identifying reused functions in binary code”. In: *Digital Investigation* 12.Supplement-1 (March 2015), pages 61–71. ISSN: 1742-2876 (Cited on page 119).

[Andersen 1994] Lars Ole Andersen. "Program Analysis and Specialization for the C Programming Language". PhD thesis. University of Copenhagen, May 1994 (Cited on page 60).

[Ashkenas 2009] Jeremy Ashkenas. *Underscore.js*. 2009. URL: `http://underscorejs.org` (visited on 30th March 2017) (Cited on pages 53, 60).

[Askarov & Myers 2010] Aslan Askarov and Andrew C. Myers. "A Semantic Framework for Declassification and Endorsement". In: *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP '10)*. Springer, March 2010, pages 64–84 (Cited on page 23).

[Askarov & Sabelfeld 2009] Aslan Askarov and Andrei Sabelfeld. "Tight Enforcement of Information-Release Policies for Dynamic Languages". In: *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF '09)*. IEEE Computer Society, July 2009, pages 43–59 (Cited on page 82).

[Backes *et al.* 2016] Michael Backes, Christian Hammer, David Pfaff and Malte Skoruppa. "Implementation-level analysis of the JavaScript helios voting client". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing – SAC 2016*. ACM, April 2016, pages 2071–2078 (Cited on page 81).

[Backes *et al.* 2008] Michael Backes, Catalin Hritcu and Matteo Maffei. "Automated Verification of Remote Electronic Voting Protocols in the Applied Pi-Calculus". In: *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF '08)*. IEEE Computer Society, June 2008, pages 195–209 (Cited on page 79).

[Backes *et al.* 2009] Michael Backes, Boris Köpf and Andrey Rybalchenko. "Automatic Discovery and Quantification of Information Leaks". In: *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P '09)*. IEEE Computer Society, May 2009, pages 141–153 (Cited on page 23).

[Balzarotti *et al.* 2008] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel and Giovanni Vigna. "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications". In: *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P '08)*. IEEE Computer Society, May 2008, pages 387–401 (Cited on page 117).

[Benaloh 2006] Josh Benaloh. “Simple Verifiable Elections”. In: *Proceedings of the 2006 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT ’06)*. USENIX Association, August 2006 (Cited on page 57).

[Benaloh 2007] Josh Benaloh. “Ballot Casting Assurance via Voter-Initiated Poll Station Auditing”. In: *Proceedings of the 2007 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT’07)*. USENIX Association, August 2007 (Cited on page 58).

[Bernhard *et al.* 2011] David Bernhard, Véronique Cortier, Olivier Pereira, Ben Smyth and Bogdan Warinschi. “Adapting Helios for Provable Ballot Privacy”. In: *Proceedings of the 16th European Symposium on Research in Computer Security (ESORICS ’11)*. Springer, September 2011, pages 335–354 (Cited on page 80).

[Bernhard *et al.* 2012a] David Bernhard, Véronique Cortier, Olivier Pereira and Bogdan Warinschi. “Measuring Vote Privacy, Revisited”. In: *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS ’12)*. ACM, October 2012, pages 941–952 (Cited on page 80).

[Bernhard *et al.* 2012b] David Bernhard, Olivier Pereira and Bogdan Warinschi. “How Not to Prove Yourself: Pitfalls of the Fiat-Shamir Heuristic and Applications to Helios”. In: *Proceedings of the 18th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT ’12)*. Springer, December 2012, pages 626–643 (Cited on page 80).

[Biba 1977] Kenneth J. Biba. *Integrity Considerations for Secure Computer Systems*. Technical report. MITRE Corporation, April 1977 (Cited on page 23).

[Bulens *et al.* 2011] Philippe Bulens, Damien Giry and Olivier Pereira. “Running Mixnet-Based Elections with Helios”. In: *Proceedings of the 2011 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections (EVT/WOTE ’11)*. USENIX Association, August 2011 (Cited on pages 49, 81).

[Chugh *et al.* 2009] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala and Sorin Lerner. “Staged Information Flow for JavaScript”. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’09)*. ACM, June 2009, pages 50–62 (Cited on page 82).

[Cooper *et al.* 2006] Keith D. Cooper, Timothy J. Harvey and Ken Kennedy. *A Simple, Fast Dominance Algorithm.* Technical report. Rice University, Department of Computer Science, 2006 (Cited on page 38).

[Cortier *et al.* 2013] Véronique Cortier, David Galindo, Stéphane Glondu and Malika Izabachène. "Distributed ElGamal à la Pedersen: Application to Helios". In: *Proceedings of the 12th annual ACM Workshop on Privacy in the Electronic Society (WPES '13).* ACM, November 2013, pages 131–142 (Cited on page 81).

[Cortier *et al.* 2014] Véronique Cortier, David Galindo, Stéphane Glondu and Malika Izabachène. "Election Verifiability for Helios under Weaker Trust Assumptions". In: *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS '14).* Springer, September 2014, pages 327–344 (Cited on page 80).

[Cortier & Smyth 2011] Véronique Cortier and Ben Smyth. "Attacking and Fixing Helios: An Analysis of Ballot Secrecy". In: *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF '11).* IEEE Computer Society, June 2011, pages 297–311 (Cited on pages 55, 79).

[Cramer *et al.* 1997] Ronald Cramer, Rosario Gennaro and Berry Schoenmakers. "A Secure and Optimally Efficient Multi-Authority Election Scheme". In: *Proceedings of the 16th International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT '97).* Springer, May 1997, pages 103–118 (Cited on page 56).

[Curtsinger *et al.* 2011] Charlie Curtsinger, Benjamin Livshits, Benjamin G. Zorn and Christian Seifert. "ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection". In: *Proceedings of the 20th USENIX Security Symposium (USENIX Security '11).* USENIX Association, August 2011 (Cited on pages 48, 82).

[Cytron *et al.* 1989] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. "An Efficient Method of Computing Static Single Assignment Form". In: *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages (POPL '89).* ACM Press, January 1989, pages 25–35 (Cited on pages 40, 60).

[Dahse & Holz 2014a] Johannes Dahse and Thorsten Holz. "Simulation of Built-in PHP Features for Precise Static Code Analysis". In: *Proceedings of the 21st Annual Network and Distributed System Security Symposium*

(NDSS '14). The Internet Society, February 2014 (Cited on pages 87, 117, 118).

[Dahse & Holz 2014b] Johannes Dahse and Thorsten Holz. "Static Detection of Second-Order Vulnerabilities in Web Applications". In: *23rd USENIX Security Symposium (USENIX Security '14)*. USENIX Association, August 2014, pages 989–1003 (Cited on pages 87, 117).

[Delaune *et al.* 2009] Stéphanie Delaune, Steve Kremer and Mark Ryan. "Verifying privacy-type properties of electronic voting protocols". In: *Journal of Computer Security* 17.4 (December 2009), pages 435–487. ISSN: 0926-227X (Cited on page 79).

[Desmedt & Chaidos 2012] Yvo Desmedt and Pyrros Chaidos. "Applying Divertibility to Blind Ballot Copying in the Helios Internet Voting System". In: *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS '12)*. Springer, September 2012, pages 433–450 (Cited on page 80).

[Deveria 2017] Alexis Deveria. *Can I use Web Workers?* 2017. URL: `http://caniuse.com/#feat=webworkers` (visited on 30th March 2017) (Cited on page 77).

[Ecma 2007] Ecma Technical Committee. *Proposed ECMAScript 4th Edition – Language Overview*. 2007. URL: `http://www.ecmascript.org/es4/spec/overview.pdf` (visited on 24th October 2016) (Cited on page 48).

[Estehghari & Desmedt 2010] Saghar Estehghari and Yvo Desmedt. "Exploiting the Client Vulnerabilities in Internet E-voting Systems: Hacking Helios 2.0 as an Example". In: *Proceedings of the 2010 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections (EVT/WOTE '10)*. USENIX Association, August 2010 (Cited on page 81).

[FB 2017] Facebook. *Company Info*. 2017. URL: `http://newsroom.fb.com/company-info/` (visited on 30th March 2017) (Cited on page 1).

[Ferrante *et al.* 1987] Jeanne Ferrante, Karl J. Ottenstein and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), pages 319–349. ISSN: 0164-0925 (Cited on pages 36, 39, 42, 119).

[Gloc 2007] Tomasz Gloc. *The jTemplates Engine*. 2007. URL: `http://jtemplates.tpython.com` (visited on 30th March 2017) (Cited on page 62).

[Goguen & Meseguer 1984] Joseph A. Goguen and José Meseguer. "Unwinding and Inference Control". In: *Proceedings of the 5th IEEE Symposium on Security and Privacy (S&P '84)*. IEEE Computer Society, April 1984, pages 75–87 (Cited on page 81).

[Goldsmith *et al.* 2005] Simon F. Goldsmith, Robert O'Callahan and Alex Aiken. "Relational Queries Over Program Traces". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, October 2005, pages 385–402 (Cited on page 118).

[Graf 2010] Jürgen Graf. "Speeding Up Context-, Object- and Field-sensitive SDG generation". In: *Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM '10)*. IEEE Computer Society, September 2010, pages 105–114 (Cited on page 119).

[Guarnieri & Livshits 2009] Salvatore Guarnieri and V. Benjamin Livshits. "GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code". In: *Proceedings of the 18th Conference on USENIX Security Symposium (USENIX Security '09)*. USENIX Association, June 2009, pages 151–168 (Cited on page 82).

[Guarnieri *et al.* 2011] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet and Ryan Berg. "Saving the World Wide Web from Vulnerable JavaScript". In: *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, July 2011, pages 177–187 (Cited on page 82).

[Guha *et al.* 2009] Arjun Guha, Shriram Krishnamurthi and Trevor Jim. "Using Static Analysis for Ajax Intrusion Detection". In: *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*. ACM, April 2009, pages 561–570 (Cited on page 82).

[Halfond *et al.* 2006] William G.J. Halfond, Jeremy Viegas and Alessandro Orso. "A Classification of SQL-Injection Attacks and Countermeasures". In: *Proceedings of the International Symposium on Secure Software Engineering*. IEEE Computer Society, March 2006 (Cited on page 9).

[Hallem *et al.* 2002] Seth Hallem, Benjamin Chelf, Yichen Xie and Dawson Engler. "A System and Language for Building System-Specific, Static Analyses". In: *Proceedings of the 23rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '02)*. ACM, June 2002, pages 69–82 (Cited on page 118).

[Hammer 2009] Christian Hammer. "Information Flow Control for Java – A Comprehensive Approach based on Path Conditions in Dependence Graphs". PhD thesis. Universität Karlsruhe (TH), July 2009 (Cited on pages 119, 121).

[Hammer *et al.* 2006] Christian Hammer, Jens Krinke and Gregor Snelting. "Information Flow Control for Java Based on Path Conditions in Dependence Graphs". In: *Proceedings of the International Symposium on Secure Software Engineering (ISSSE '06)*. IEEE, March 2006, pages 87–96 (Cited on page 82).

[Hammer & Snelting 2009] Christian Hammer and Gregor Snelting. "Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs". In: *International Journal of Information Security* 8.6 (December 2009), pages 399–422. ISSN: 1615-5262 (Cited on page 59).

[Hansen 2015] Robert Hansen. *Magic Hashes*. 2015. URL: `https://www.whitehatsec.com/blog/magic-hashes` (visited on 30th March 2017) (Cited on page 32).

[Harris 2008] Brantley Harris. *jQuery JSON Plugin*. 2008. URL: `https://github.com/Krinkle/jquery-json` (visited on 30th March 2017) (Cited on page 62).

[Hedin *et al.* 2016] Daniel Hedin, Luciano Bello and Andrei Sabelfeld. "Information-flow security for JavaScript and its APIs". In: *Journal of Computer Security* 24.2 (May 2016), pages 181–234. ISSN: 0926-227X (Cited on pages 48, 82).

[Hedin & Sabelfeld 2012] Daniel Hedin and Andrei Sabelfeld. "Information-Flow Security for a Core of JavaScript". In: *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF '12)*. IEEE Computer Society, June 2012, pages 3–18 (Cited on pages 48, 82).

[Horwitz *et al.* 1990] Susan Horwitz, Thomas Reps and David Binkley. "Interprocedural Slicing Using Dependence Graphs". In: *ACM Transactions on Programming Languages and Systems* 12.1 (January 1990), pages 26–60. ISSN: 0164-0925 (Cited on page 68).

[HF 2017] Hosting Facts. *Internet Stats & Facts for 2016*. 2017. URL: https://hostingfacts.com/internet-facts-stats-2016/ (visited on 30th March 2017) (Cited on page 1).

[Huang *et al.* 2004a] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee and Sy-Yen Kuo. "Securing Web Application Code by Static Analysis and Runtime Protection". In: *Proceedings of the 13th international conference on World Wide Web (WWW '04)*. ACM, May 2004, pages 40–52 (Cited on page 116).

[Huang *et al.* 2004b] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee and Sy-Yen Kuo. "Verifying Web Applications Using Bounded Model Checking". In: *Proceedings of the International Conference on Dependable Systems and Networks (DSN '04)*. IEEE Computer Society, June 2004, pages 199–208 (Cited on page 116).

[IACR 2010] IACR. *International Association for Cryptologic Research: Should the IACR use e-voting for its elections?* 2010. URL: http://iacr.org/elections/eVoting/ (visited on 30th March 2017) (Cited on page 49).

[IBM 2006] IBM T.J. Watson Research Center. *T.J. Watson Libraries for Analysis (WALA)*. 2006. URL: http://wala.sf.net (visited on 30th March 2017) (Cited on page 59).

[ILS 2017] Internet Live Stats. *Internet Users*. 2017. URL: http://www.internetlivestats.com/internet-users/ (visited on 30th March 2017) (Cited on pages 1, 2).

[Jensen *et al.* 2012] Simon Holm Jensen, Peter A. Jonsson and Anders Møller. "Remedying the Eval that Men Do". In: *Proceedings of the 21st International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, July 2012, pages 34–44 (Cited on page 82).

[Jensen *et al.* 2011] Simon Holm Jensen, Magnus Madsen and Anders Møller. "Modeling the HTML DOM and Browser API in Static Analysis of JavaScript Web Applications". In: *Proceedings of the 19th ACM SIGSOFT*

Symposium on the Foundations of Software Engineering (FSE '11). ACM, September 2011, pages 59–69 (Cited on page 82).

[Jensen *et al.* 2009] Simon Holm Jensen, Anders Møller and Peter Thiemann. "Type Analysis for JavaScript". In: *Proceedings of the 16th International Symposium on Static Analysis (SAS '09)*. Springer, August 2009, pages 238–255 (Cited on page 82).

[Johns *et al.* 2011] Martin Johns, Bastian Braun, Michael Schrank and Joachim Posegga. "Reliable Protection Against Session Fixation Attacks". In: *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*. ACM, March 2011, pages 1531–1537 (Cited on page 20).

[Johnson *et al.* 2015] Andrew Johnson, Lucas Waye, Scott Moore and Stephen Chong. "Exploring and Enforcing Security Guarantees via Program Dependence Graphs". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, June 2015, pages 291–302 (Cited on page 119).

[Jovanovic *et al.* 2006] Nenad Jovanovic, Christopher Kruegel and Engin Kirda. "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities". In: *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P '06)*. IEEE Computer Society, May 2006, pages 258–263 (Cited on pages 87, 117).

[Jovanovic *et al.* 2010] Nenad Jovanovic, Christopher Kruegel and Engin Kirda. "Static analysis for detecting taint-style vulnerabilities in web applications". In: *Journal of Computer Security* 18.5 (September 2010), pages 861–907. ISSN: 0926-227X (Cited on pages 87, 117, 118).

[jQuery 2005] jQuery Foundation. *jQuery Core*. 2005. URL: `http://jquery.com` (visited on 30th March 2017) (Cited on pages 48, 53, 60).

[Karayumak *et al.* 2011a] Fatih Karayumak, Michaela Kauer, Maina M. Olembo, Tobias Volk and Melanie Volkamer. "User Study of the Improved Helios Voting System Interfaces". In: *1st Workshop on Socio-Technical Aspects in Security and Trust (STAST '11)*. IEEE Computer Society, September 2011, pages 37–44 (Cited on page 81).

[Karayumak *et al.* 2011b] Fatih Karayumak, Maina M. Olembo, Michaela Kauer and Melanie Volkamer. "Usability Analysis of Helios - An Open Source Verifiable Remote Electronic Voting System". In: *Proceedings of the*

2011 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections (EVT/WOTE '11). USENIX Association, August 2011 (Cited on page 81).

[Kinloch & Munro 1994] David A. Kinloch and Malcolm Munro. "Understanding C Programs Using the Combined C Graph Representation". In: *Proceedings of the 2nd International Conference on Software Maintenance (ICSM '94)*. IEEE Computer Society, September 1994, pages 172–180 (Cited on page 119).

[Kirda *et al.* 2006] Engin Kirda, Christopher Kruegel, Giovanni Vigna and Nenad Jovanovic. "Noxes: A Client-Side Solution for Mitigating Cross-Site Scripting Attacks". In: *Proceedings of the 21st ACM Symposium on Applied Computing (SAC '06)*. ACM, April 2006, pages 330–337 (Cited on page 16).

[Kremer *et al.* 2010] Steve Kremer, Mark Ryan and Ben Smyth. "Election Verifiability in Electronic Voting Protocols". In: *Proceedings of the 15th European Symposium on Research in Computer Security (ESORICS '10)*. Springer, September 2010, pages 389–404 (Cited on page 80).

[Küsters *et al.* 2012] Ralf Küsters, Tomasz Truderung and Andreas Vogt. "Clash Attacks on the Verifiability of E-Voting Systems". In: *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*. IEEE Computer Society, May 2012, pages 395–409 (Cited on page 80).

[Lam *et al.* 2005] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin and Christopher Unkel. "Context-sensitive Program Analysis as Database Queries". In: *Proceedings of the 24th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '05)*. ACM, June 2005, pages 1–12 (Cited on page 118).

[Lengauer & Tarjan 1979] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph". In: *ACM Transactions on Programming Languages and Systems* 1.1 (January 1979), pages 121–141. ISSN: 0164-0925 (Cited on page 38).

[Lerdorf 2003] Rasmus Lerdorf. *IT Conversations. Behind the Mic: PHP*. 2003. URL: http://itc.conversationsnetwork.org/shows/detail58.html (visited on 30th March 2017) (Cited on page 86).

[Livshits & Lam 2005] V. Benjamin Livshits and Monica S. Lam. "Finding Security Vulnerabilities in Java Applications with Static Analysis". In: *Proceedings of the 14th USENIX Security Symposium (USENIX Security '05)*. USENIX Association, July 2005 (Cited on page 118).

[Lowry & Medlock 1969] Edward S. Lowry and C. W. Medlock. "Object Code Optimization". In: *Communications of the ACM* 12.1 (January 1969), pages 13–22. ISSN: 0001-0782 (Cited on page 36).

[Martin *et al.* 2005] Michael C. Martin, V. Benjamin Livshits and Monica S. Lam. "Finding Application Errors and Security Flaws Using PQL: A Program Query Language". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, October 2005, pages 365–383 (Cited on page 118).

[Meyerovich & Livshits 2010] Leo A. Meyerovich and V. Benjamin Livshits. "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser". In: *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P '10)*. IEEE Computer Society, May 2010, pages 481–496 (Cited on page 82).

[Mitchelmore 2009] Blair Mitchelmore. *jQuery Query Object Plugin*. 2009. URL: `https://github.com/alrusdi/jquery-plugin-query-object` (visited on 30th March 2017) (Cited on page 62).

[Mozilla 1998] Mozilla Foundation. *Rhino*. 1998. URL: `https://developer.mozilla.org/en/docs/Rhino` (visited on 30th March 2017) (Cited on pages 60, 67).

[Munroe 2012] Alex Munroe. *PHP: a fractal of bad design*. 2012. URL: `https://eev.ee/blog/2012/04/09/php-a-fractal-of-bad-design` (visited on 30th March 2017) (Cited on page 85).

[Myers 1999] Andrew C. Myers. "JFlow: Practical Mostly-Static Information Flow Control". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, January 1999, pages 228–241 (Cited on page 81).

[Netcraft 2017] Netcraft. *Web Server Survey*. 2017. URL: `https://news.netcraft.com/archives/category/web-server-survey/` (visited on 30th March 2017) (Cited on page 2).

[OAuth 2006] The OAuth Group. *The OAuth authorization framework*. 2006. URL: http://oauth.net (visited on 30th March 2017) (Cited on page 56).

[Olivo *et al.* 2015] Oswaldo Olivo, Isil Dillig and Calvin Lin. "Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, October 2015, pages 616–628 (Cited on page 117).

[OWASP Top Ten 2013] OWASP. *OWASP Top 10 - 2013 - The Ten Most Critical Web Application Security Risks*. 2013. URL: https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/owasptop10/OWASP%20Top%2010%20-%202013.pdf (visited on 30th March 2017) (Cited on page 6).

[Paul & Prakash 1994] Santanu Paul and Atul Prakash. "A Framework for Source Code Search Using Program Patterns". In: *IEEE Transactions on Software Engineering* 20.6 (June 1994), pages 463–475. ISSN: 0098-5589 (Cited on page 118).

[PHP Group 2014] PHP Group. *PHP RFC: Abstract syntax tree*. 2014. URL: https://wiki.php.net/rfc/abstract_syntax_tree (visited on 30th March 2017) (Cited on pages 30, 93).

[PHP Group 2017a] PHP Group. *PHP Grammar*. 2017. URL: https://github.com/php/php-src/blob/master/Zend/zend_language_parser.y (visited on 30th March 2017) (Cited on page 29).

[PHP Group 2017b] PHP Group. *Predefined Variables*. 2017. URL: http://php.net/manual/en/reserved.variables.php (visited on 30th March 2017) (Cited on page 98).

[PHP Group 2017c] PHP Group. *Using remote files*. 2017. URL: http://php.net/manual/en/features.remote-files.php (visited on 30th March 2017) (Cited on page 13).

[Princeton USG 2013] Princeton USG. *Princeton Elections Handbook*. 2013. URL: http://princetonusg.com/?page_id=975 (visited on 30th March 2017) (Cited on page 49).

[Prosser 1959] Reese T. Prosser. "Applications of Boolean Matrices to the Analysis of Flow Diagrams". In: *Papers Presented at the Eastern Joint*

IRE-AIEE-ACM Computer Conference. ACM, December 1959, pages 133–138 (Cited on page 36).

[Purdom & Moore 1972] Jr. Paul W. Purdom and Edward F. Moore. “Immediate Predominators in a Directed Graph”. In: *Communications of the ACM* 15.8 (August 1972), pages 777–778. ISSN: 0001-0782 (Cited on page 38).

[Reps 1998] Thomas Reps. “Program Analysis via Graph Reachability”. In: *Information and Software Technology* 40.11/12 (December 1998), pages 701–726. ISSN: 0950-5849 (Cited on page 118).

[Reps *et al.* 1994] Thomas Reps, Susan Horwitz, Mooly Sagiv and Genevieve Rosay. “Speeding up Slicing”. In: *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE ’94)*. ACM, December 1994, pages 11–20 (Cited on page 69).

[Reps *et al.* 1995] Thomas Reps, Susan Horwitz and Shmuel Sagiv. “Precise Interprocedural Dataflow Analysis via Graph Reachability”. In: *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’95)*. ACM Press, January 1995, pages 49–61 (Cited on page 60).

[Resig 2008] John Resig. *Simple JavaScript inheritence*. 2008. URL: `http://ejohn.org/blog/simple-javascript-inheritance` (visited on 30th March 2017) (Cited on pages 54, 60).

[Richards *et al.* 2011] Gregor Richards, Christian Hammer, Brian Burg and Jan Vitek. “The Eval That Men Do: A Large-Scale Study of the Use of Eval in JavaScript Applications”. In: *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP ’11)*. Springer, July 2011, pages 52–78 (Cited on pages 12, 82).

[Ryder 1979] Barbara G. Ryder. “Constructing the Call Graph of a Program”. In: *IEEE Transactions on Software Engineering* 5.3 (May 1979), pages 216–226. ISSN: 0098-5589 (Cited on page 44).

[Sabelfeld & Myers 2003] Andrei Sabelfeld and Andrew C. Myers. “Language-Based Information-Flow Security”. In: *IEEE Journal on Selected Areas in Communications* 21.1 (January 2003), pages 5–19. ISSN: 0733-8716 (Cited on pages 23, 82).

[Schwartz & Bloom 2014] Adam Schwartz and Zack Bloom. *You might not need jQuery*. 2014. URL: http://youmightnotneedjquery.com (visited on 30th March 2017) (Cited on page 78).

[Shankar *et al.* 2001] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster and David Wagner. "Detecting Format String Vulnerabilities with Type Qualifiers". In: *Proceedings of the 10th USENIX Security Symposium (USENIX Security '01)*. USENIX Association, August 2001, pages 201–220 (Cited on page 81).

[Smyth 2012] Ben Smyth. *Replay attacks that violate ballot secrecy in Helios*. April 2012. Cryptology ePrint Archive: 2012/185 (Cited on page 80).

[Smyth & Pironti 2013] Ben Smyth and Alfredo Pironti. "Truncating TLS Connections to Violate Beliefs in Web Applications". In: *7th USENIX Workshop on Offensive Technologies (WOOT '13)*. USENIX Association, August 2013 (Cited on pages 56, 81).

[Snelting *et al.* 2014] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr and Daniel Wasserrab. "Checking Probabilistic Noninterference Using JOANA". In: *it – Information Technology* 56.6 (December 2014), pages 280–287. ISSN: 2196-7032 (Cited on page 119).

[Snelting *et al.* 2006] Gregor Snelting, Torsten Robschink and Jens Krinke. "Efficient Path Conditions in Dependence Graphs for Software Safety Analysis". In: *ACM Transactions on Software Engineering and Methodology* 15.4 (October 2006), pages 410–457. ISSN: 1049-331X (Cited on page 81).

[Stock & Johns 2014] Ben Stock and Martin Johns. "Protecting users against XSS-based password manager abuse". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS '14)*. ACM, June 2014, pages 183–194 (Cited on page 16).

[Sturton *et al.* 2009] Cynthia Sturton, Susmit Jha, Sanjit A. Seshia and David Wagner. "On Voting Machine Design for Verification and Testability". In: *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. ACM, November 2009, pages 463–476 (Cited on page 80).

[Tripp *et al.* 2013] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot and Salvatore Guarnieri. "Andromeda: Accurate and Scalable Security Analysis of Web Applications". In: *Proceedings of the 16th International*

Conference on Fundamental Approaches to Software Engineering (FASE '13). Springer, March 2013, pages 210–225 (Cited on pages 59, 82).

[Tripp *et al.* 2009] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan and Omri Weisman. "TAJ: Effective Taint Analysis of Web Applications". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, June 2009, pages 87–97 (Cited on page 82).

[Vogt *et al.* 2007] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel and Giovanni Vigna. "Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In: *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS '07)*. The Internet Society, March 2007 (Cited on page 82).

[Volpano *et al.* 1996] Dennis Volpano, Cynthia Irvine and Geoffrey Smith. "A Sound Type System for Secure Flow Analysis". In: *Journal of Computer Security* 4.2/3 (January 1996), pages 167–188. ISSN: 0926-227X (Cited on page 81).

[W3Techs 2017a] W3Techs. *Usage of client-side programming languages for websites*. 2017. URL: `https://w3techs.com/technologies/overview/client_side_language/all` (visited on 30th March 2017) (Cited on page 48).

[W3Techs 2017b] W3Techs. *Usage of server-side programming languages for websites*. 2017. URL: `http://w3techs.com/technologies/overview/programming_language/all` (visited on 30th March 2017) (Cited on page 85).

[Wassermann & Su 2007] Gary Wassermann and Zhendong Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, June 2007, pages 32–41 (Cited on page 117).

[Wassermann & Su 2008] Gary Wassermann and Zhendong Su. "Static Detection of Cross-Site Scripting Vulnerabilities". In: *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, May 2008, pages 171–180 (Cited on page 117).

[Wasserrab & Lohner 2010] Daniel Wasserrab and Denis Lohner. “Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing”. In: *6th International Verification Workshop (VERIFY ’10)*. EasyChair, July 2010, pages 141–155 (Cited on page 69).

[Weiser 1979] Mark David Weiser. “Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method”. PhD thesis. University of Michigan, January 1979 (Cited on page 60).

[Weiser 1981] Mark David Weiser. “Program Slicing”. In: *Proceedings of the 5th International Conference on Software Engineering (ICSE ’81)*. IEEE Computer Society, March 1981, pages 439–449 (Cited on page 39).

[WhiteHat 2015] WhiteHat Security. *Website Security Statistics Report*. 2015. URL: https://info.whitehatsec.com/rs/whitehatsecurity/images/2015-Stats-Report.pdf (visited on 30th March 2017) (Cited on page 114).

[Xie & Aiken 2006] Yichen Xie and Alex Aiken. “Static Detection of Security Vulnerabilities in Scripting Languages”. In: *Proceedings of the 15th USENIX Security Symposium (USENIX Security ’06)*. USENIX Association, July 2006 (Cited on page 117).

[Yamaguchi 2015] Fabian Yamaguchi. “Pattern-based Vulnerability Discovery”. PhD thesis. University of Göttingen, November 2015 (Cited on pages 34, 89, 121).

[Yamaguchi *et al.* 2014] Fabian Yamaguchi, Nico Golde, Daniel Arp and Konrad Rieck. “Modeling and Discovering Vulnerabilities with Code Property Graphs”. In: *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P ’14)*. IEEE Computer Society, May 2014, pages 590–604 (Cited on pages 4, 32, 87, 89, 92, 97, 105, 119).

[Yu *et al.* 2010] Fang Yu, Muath Alkhalaf and Tevfik Bultan. “STRANGER: An Automata-based String Analysis Tool for PHP”. In: *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’10)*. Springer, March 2010, pages 154–157 (Cited on page 117).

[Zalewski 2011] Michal Zalewski. *The Tangled Web: A Guide to Securing Modern Web Applications*. 1st edition. No Starch Press, November 2011. ISBN: 1593273886 (Cited on page 114).

[Zapponi 2017] Carlo Zapponi. *GitHut*. 2017. URL: http://githut.info/ (visited on 30th March 2017) (Cited on page 85).