Automated Security Review of PHP Web Applications with Static Code Analysis

An evaluation of current tools and their applicability*

Nico L. de Poel

Supervisors
Frank B. Brokken
Gerard R. Renardel de Lavalette

May 28, 2010

xkcd.com

*With thanks to Arjen R. van der Veen for building a prototype implementation.

Abstract

Static code analysis is a class of techniques for inspecting the source code of a computer program without executing it. One specific use of static analysis is to automatically scan source code for potential security problems, reducing the need for manual code reviews. Many web applications written in PHP suffer from injection vulnerabilities, and static analysis makes it possible to track down these vulnerabilities before they are exposed on the web.

In this thesis, we evaluate the current state of static analysis tools targeted at the security of PHP web applications. We define an objective benchmark consisting of both synthetic and real-world tests, that we use to examine the capabilities and performance of these tools. With this information, we determine if any of these tools are suited for use in a system that automatically checks the security of web applications, and rejects unsecure applications before they are deployed onto a web server.

Contents

1 Introduction **6**

2 Motivation **7**

3 Software security **8**
- 3.1 Causes of security flaws 8
- 3.2 Vulnerability categories 9
- 3.3 Security testing 10
- 3.4 Organizations 11

4 Web application security **13**
- 4.1 Web application vulnerabilities 14
- 4.2 Cross-site scripting 16
- 4.3 SQL injection 19
- 4.4 Other vulnerabilities 22

5 Static analysis **23**
- 5.1 Applications of static analysis 23
- 5.2 History 25
- 5.3 Steps 25
  - 5.3.1 Model construction 26
  - 5.3.2 Analysis 27
  - 5.3.3 Rules 28
  - 5.3.4 Results processing 30
- 5.4 Limitations 31

6 PHP **32**
- 6.1 Language complexity 32

7 Problem statement **36**

8 Testing methodology **39**
- 8.1 Benchmarking criteria 39
- 8.2 Static analysis benchmarking 40
- 8.3 Motivating comparison 41
- 8.4 Task sample 42
- 8.5 Performance measures 44

9 Tool evaluation **47**
- 9.1 Fortify 360 47
- 9.2 CodeSecure 55
- 9.3 Pixy 61

9.4 PHP-sat . . . 66
9.5 Remaining tools . . . 70
9.5.1 WebSSARI . . . 70
9.5.2 PHPrevent . . . 71
9.5.3 phc . . . 71
9.5.4 RATS . . . 72
9.5.5 PHP String Analyzer . . . 73

10 Discussion **74**
10.1 Commercial vs. open source . . . 74
10.2 Software as a Service . . . 75
10.3 Future work . . . 77
10.3.1 Static analysis for PHP . . . 77
10.3.2 Benchmark development . . . 78
10.3.3 Dynamic analysis . . . 79

11 Conclusion **80**

References **82**

Appendices

A Micro test results **88**

B Micro-benchmark code **100**

List of Figures

1 Typical web application architecture (adapted from [9]) 13
2 Example of an HTML form and a dynamically generated result page . . . 16
3 Example of web page layout changed through XSS 17
4 Example of Javascript code added through XSS 17
5 Example result page with sanitized input 18
6 Web application architecture with added static analysis step 36
7 Relation between code size and analysis time in Fortify SCA 52
8 Relation between code complexity and analysis time in Fortify SCA . . . 53
9 CodeSecure visual reports 59

List of Tables

1 Macro-benchmark test applications 43
3 Fortify micro-benchmark test results (abridged) 49
4 Fortify macro-benchmark test results 50
5 CodeSecure micro-benchmark test results (abridged) 57
6 CodeSecure macro-benchmark test results 58
7 Pixy micro-benchmark test results (abridged) 62
8 Pixy macro-benchmark test results[1] 63
9 PHP-sat micro-benchmark test results (abridged) 67
10 PHP-sat macro-benchmark test results 68

List of Listings

1 Example of an XSS vulnerability 16
2 Example of an SQL query constructed through string concatenation . . . 20
3 Example of unsafe usage of sanitized input 21
4 Using prepared SQL statements in PHP 22
5 Example PQL code 29

1 Introduction

Web applications are growing more and more popular as both the availability and speed of the internet increase. Many web servers nowadays are equipped with some sort of scripting environment for the deployment of dynamic web applications. However, most public web hosting services do not enforce any kind of quality assurance on the applications that they run, which can leave a web server open to attacks from the outside. Poorly written web applications are highly vulnerable to attacks because of their easy accessibility on the internet. One careless line of program code could potentially bring down an entire computer network.

The intention of this thesis is to find out if it is feasible to use static code analysis to automatically detect security problems in web applications before they are deployed. Specifically, we look at static analysis tools for the PHP programming language, one of the most popular languages for the development of web applications. We envision a tool running in the background of a web server, scanning PHP programs as they are uploaded and deploying only those programs that are found to be free of vulnerabilities.

First, we look at software security in general, what type of security problems can occur and why, and which methods exist to test the security of an application. Next, we take a more in-depth view at web application security and the most common vulnerabilities that plague web applications. We then explain what static analysis is, how it works, what it can be used for, and the specific challenges involved with static analysis of the PHP programming language.

We formulate a series of benchmark tests that can be used to evaluate the performance of PHP static analysis tools, both in general and for our specific use case. With the requirements for our use case and this benchmark, we evaluate the current offering of PHP static analysis tools, both in quantitative and qualitative terms. Finally, we discuss the results of our evaluations and the observations we made, as well as the possible future directions of this project.

2 Motivation

This research project is a direct result of a hack on a real web host. A customer of the web host had placed a simple PHP script on their web space that allowed one to remotely execute system commands on the web server by entering them in a web browser. This script was left in a publicly accessible location, so it was only a matter of time before someone with bad intentions would find it. When that happened, the attacker had the power to execute commands with the access rights of the web server software, effectively gaining access to all other web applications on the same server.

Static analysis is an increasingly popular way to check the source code of an application for problems, both security-related and otherwise. It is primarily used by developers in the form of tools that help to find mistakes that are not easily found by hand.

We are interested in seeing whether we can use static analysis tools to automatically check PHP web applications for security-related problems, before they are deployed onto a web server and exposed to the world. This way, we could prevent future attacks like the one mentioned above, by removing the root of the problem – the vulnerable application itself.

Simply installing an automated security review will not immediately solve the problem, however. Consider the following example situation. A casual customer has recently registered an account with a web hosting service, which includes some web space, unrestricted scripting support and access to a database. The customer wants to upload their favorite prefab web application to this web host, for instance a phpBB message board. Past versions of phpBB have been known to suffer from serious security leaks [51], but the customer is oblivious to this fact and unwittingly uploads an old version of phpBB. The web host automatically performs a security review of the uploaded code, finds several high-risk security flaws, rejects the application and returns a detailed report to the customer. The customer is not familiar with the source code of phpBB and does not know how to fix any of these problems, nor do they care. They just want to get their message board up and running as quickly as possible. The likely outcome of this situation is that the customer files a complaint or cancels their account altogether. Expecting an average customer to deal with any problems in their application is not a sufficient solution.

We want to keep the applications running on the web host safe and secure, yet we also want customers to retain their freedom, and we certainly do not want to scare customers away in the process.

3 Software security

Software and information security are based on the principles of confidentiality, integrity and availability, or CIA for short. Constraining information flow is fundamental in this [25]: we do not want secret information to reach untrusted parties (*confidentiality*), and we do not want untrusted parties to corrupt trusted information (*integrity*). The easiest way to reach this goal would be to forbid interaction between parties of different trust levels, but this would make information systems such as the Internet unusable. We do want select groups of people to be able to access confidential information under certain conditions (*availability*).

If a program tester or a tool can find a vulnerability inside the source code of a software system, then so can a hacker who has gotten access to the source. This is especially true for open source software, which is free for everyone to inspect, including hackers. If a hacker finds a vulnerability in a software system, they might use it as a basis for a virus, worm or trojan. Thus, it is vital for developers and security managers to find vulnerabilities before a hacker can exploit them.

3.1 Causes of security flaws

There are several reasons why poor quality and unsecure code gets written. First, and most obviously, is limited skill of the programmer. This is especially true for web application programmers. Easily accessible languages such as PHP make it possible for anyone with basic programming skills to quickly create impressive web applications, even if the quality of these applications is far from professional [44].

Unsecure code can also be attributed to a lack of awareness on the part of the programmer. Many programmers are not aware or simply do not care that the security of their application could be compromised through entry of malicious data. This phenomenon is worsened by poor education. Most programming textbooks do not stress the importance of security when writing code and are in some cases actually guilty of teaching habits that can lead directly to unsecure code [9]. Additionally, most programming jobs do not require any certificates from their applicants to demonstrate the programmer's skill and expertise in their area.

Another important reason not attributable to programmers are financial and time constraints imposed by management. Most software is produced under strict deadlines and with a tight budget, meaning that corners need to be cut and careful design and exten-

sive testing are restricted. Creating a functional and deployable application quickly is often top priority, while security is an afterthought at best.

Traditional quality assurance mostly focuses on how well an implementation conforms to its requirements, and less on any possible negative side-effects of that implementation [1, Sec.1.3]. Security problems are often not violations of the requirements.

For a program to be secure, all portions of the program must be secure, not just the parts that explicitly address security [1, Sec.1.2]. Vulnerabilities often originate from code that is unrelated to security and of which the programmer is not aware it can cause a vulnerability.

Techniques for avoiding security vulnerabilities are not a self-evident part of the development process [5]. Even programmers who are aware of the risks can overlook security issues, especially when working with undocumented procedures and data types. For example, what data should be considered as program input and thus as untrusted is not always clear. Data coming from a database or an interrupt is also input, yet most programmers do not regard them as such [1, pg.121].

3.2 Vulnerability categories

Tsipenyuk et al. group software vulnerabilities into seven different categories, which they call the seven Pernicious Kingdoms [2]. These are as follows:

Input validation and representation. Information coming from an untrusted source, e.g. a user or an external file or database, is given too much trust and is allowed to influence the behavior and/or output of a program without proper validation. This category includes injection vulnerabilities such as cross-site scripting and SQL injection [1, Ch.5], buffer overflows [1, Ch.6][5] and integer overflows [1, Ch.7]. Injection vulnerabilities are the most common types of vulnerabilities in web applications, and consequently are the primary focus of this article.

API abuse. Application programming interfaces (APIs) typically specify certain rules and conditions under which they should be used by a program. If these rules are broken, an API may exhibit undefined behavior, or may not be guaranteed to work identically on other platforms or after upgrades. Programmers often violate the contracts in an API or rely on undocumented features and/or buggy behavior.

Security features. This includes insecure storage and broken authentication or access control. An example of broken authentication is the inclusion of a hard-coded password inside the program code.

Time and state. Programs that use multiple threads or that have multiple instances linked together over a network can have problems with their synchronization. Interactions within a multiprocessing system can be difficult to control, and network latency and race conditions may bring such a system into an unexpected state.

Error handling. Improper handling of errors or neglecting to catch exceptions can cause a program to crash or exhibit undefined behavior [1, Ch.8]. Attackers can willfully cause a program to produce errors, leaving the program in a potentially vulnerable state.

Code quality. Poor code quality leads to unpredictable behavior. Poorly written programs may exhibit resource leaks, null pointer dereferencing, or infinite loops. Attackers may use these flaws to stress the system in unexpected ways.

Encapsulation. Software systems need to draw strong boundaries between different trust levels. Users should not be allowed to access private data without proper authorization, source code from an untrusted party should not be executed, web sites should not execute script code originating from other web sites, etc.

3.3 Security testing

Methods for testing the security of a software system can be roughly divided into three groups: human code reviews, run-time testing or dynamic analysis, and static analysis.

Human code reviews are time-consuming and expensive but can find conceptual problems that are impossible to find automatically [5]. However, the quality of human code reviews depends strongly on the expertise of the reviewer and there is a risk that more mundane problems are overlooked. Human code reviews can be assisted by tools such as a debugger or a profiler.

Dynamic analysis involves observing a program during execution and monitoring its run-time behavior. Standard run-time security tests can be categorized as *black-box tests*, meaning the tester has no prior knowledge of the software system's internals. *Penetration tests* have testers attack the program from the outside, trying to find weaknesses in the system. *Fuzzing* is a similar technique, which involves feeding the program random input [1, Sec.1.3]. The problem with these methods is that they do not make use of prior knowledge of the system, and they can target only very specific areas of a program,

which means that their coverage is limited [22]. Dynamic analysis techniques will also add a performance overhead to the program's execution [17, 20]. Examples of dynamic analysis tools are OWASP WebScarab [69], SWAP [20], OpenWAVES [61] (now Armorize HackAlert), WASP [21], Fortity RTA and ScanDo.

Static analysis is a form of *white-box testing*, i.e. the entire software system is an open book and can be fully inspected. Static analysis can be seen as an automated version of human code reviews. The advantages over human code reviews are that the quality of the results is consistent and it demands less human resources. Static analysis is also capable of finding problems in sections of code that dynamic analysis can never reach. The main weakness of static analysis is that it suffers from the conceptual limitation of undecidability [22], meaning its results are never completely accurate.

Combining dynamic and static analysis allows a security testing tool to enhance the strengths of both techniques, while mitigating their weaknesses. Most examples of this combination are dynamic analysis tools that inspect incoming and outgoing data, using knowledge of the application obtained through static analysis to increase the accuracy of the dynamic analysis.
Balzarotti et al. developed a tool called Saner [23], which uses a static analyzer based on Pixy to detect sanitization routines in a program, and uses dynamic analysis to check whether the sanitization is correct and complete.
Vogt et al. combine dynamic and static data tainting in a web browser to stop cross-site scripting attacks on the client side [22].
Halfond and Orso propose a technique to counter SQL injection attacks [14]. It uses static analysis to build a conservative model of legitimate SQL queries that could be generated by the application, and uses dynamic analysis to inspect at run-time whether the dynamically generated queries comply with this model.

3.4 Organizations

There are many organizations specializing in software security research, consulting and education. A number of them are refered to in this thesis and are introduced here.

OWASP (Open Web Application Security Project, `http://www.owasp.org`) is a non-profit organization focusing on improving the security of application software. Their mission is to make application security visible, so that people and organizations can make informed decisions about true application security risks. OWASP's work includes the OWASP Top Ten of web application security risks, the dynamic analysis tool Web-

Scarab for Java web application security, the deliberately insecure J2EE application WebGoat designed to teach web application security lessons, and the CLASP guidelines for integration of security into the software development lifecycle.

WASC (Web Application Security Consortium, `http://www.webappsec.org`) is a non-profit organization made up of an international group of experts who produce best-practice security standards for the World Wide Web. WASC's work includes the Web Application Security Scanner Evaluation Criteria, which is a set of guidelines to evaluate web application scanners on their ability to effectively test web applications and identify vulnerabilities.

NIST (National Institute of Standards and Technology, `http://www.nist.gov`) is the organizer of the Static Analysis Tool Exposition (SATE), intended to enable empirical research on static analysis tools and to encourage tool improvement. NIST's other software security-related work includes the National Vulnerability Database, and the Software Assurance Metrics And Tool Evaluation (SAMATE) project aimed at the identification, enhancement and development of software assurance tools.

WhiteHat Security (`http://www.whitehatsec.com`) is a provider of website risk management solutions. Their Website Security Statistics Report offers insight on the state of website security and the issues that organizations must address to avert attack.

Fortify (`http://www.fortify.com`) is a supplier of software security assurance products and services to protect companies from the threats posed by security flaws in software applications. Their flagship product Fortify 360 includes the static analysis tool Source Code Analyzer, and the dynamic analysis tool Real Time Analyzer.

Armorize (`http://www.armorize.com`) delivers security solutions to safeguard enterprises from hackers seeking to exploit vulnerable web applications. Armorize develops both static and dynamic analysis tools (CodeSecure and HackAlert respectively), which are offered as hosted software services.

4 Web application security

A web application, as the name implies, is a computer application that is accessed through a web-based user interface. This is typically implemented through a client-server setup, with the server running an HTTP server software package (such as Apache or Microsoft IIS) capable of generating dynamic web pages, while the client communicates with the server through a web browser (such as Microsoft Internet Explorer or Mozilla Firefox). The working of such a web application is roughly sketched in Figure 1.

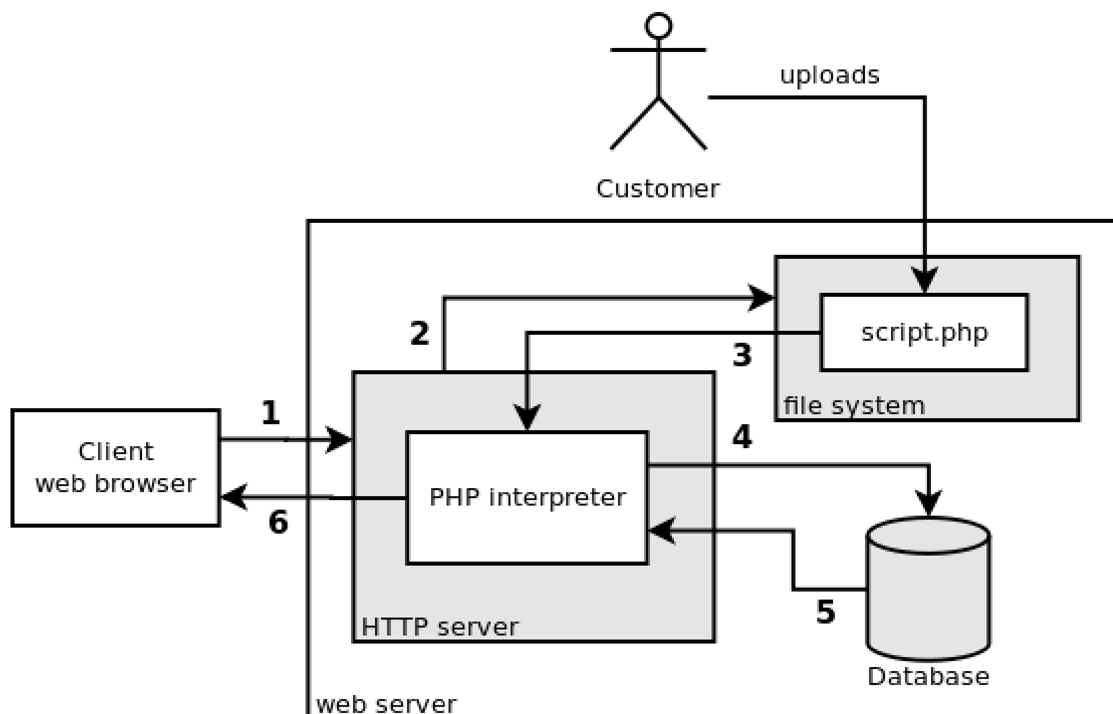


Figure 1: Typical web application architecture (adapted from [9])

Whenever the client interacts with the server, communication takes place in the form of HTTP requests. The client sends a request (typically a GET or POST request) to the server, along with a series of parameters (step 1 in Fig. 1). The HTTP server recognizes that this is a request for a dynamic web page, in this case a page that is to be generated by a PHP script. It fetches the corresponding PHP script from the web server's file system (step 2) and sends it off to be processed by the integrated PHP interpreter (step 3). The PHP interpreter then executes the PHP script, making use of external resources whenever necessary. External resources can for example be a database (steps 4 and 5), but also the file system or an API exposed by the operating system. The executed PHP

script typically produces output in the form of an HTML page, which is sent back to the client and displayed in the web browser (step 6).

A crucial detail in this process is that the script fetched and executed in steps 2 and 3 is uploaded directly to the web server's file system by a system administrator or a customer of the web host. There is no compilation or quality control step in between; the code stored and executed on the web server is identical to what the customer uploaded. This means that unsafe code will be executed unconditionally.

This security risk is made worse by the fact that vulnerabilities in the source code of a dynamic web page on a public site are open for anyone on the internet to exploit. Web applications are by default executed with the access rights of the HTTP server, meaning that if even one web application is compromised, it could potentially bring down the entire web server and take with it all the other web pages hosted on the same server.

Many public web hosts circumvent this problem by restricting the right to use dynamically executed code. In the early days of the internet, common practice for public web hosts was to provide their users with a preselected handful of CGI scripts that they could use for dynamic behavior of their websites, but no more than that. Today it is more common for web hosts to provide a prefab application framework (e.g. a blogging application) with strongly restricted customization options. While such strategies do indeed limit the risk of attacks, they also limit the number of applications that a web host can be used for.

4.1 Web application vulnerabilities

The most common and most dangerous vulnerabilities appearing in web applications belong to a general class of vulnerabilities called *taint-style vulnerabilities* [7, 9, 12, 26]. The common characteristic of taint-style vulnerabilities is that data enters a program from an untrusted source and is passed onto a vulnerable part of the program without having been cleaned up properly. Data originating from an untrusted source is called *tainted*. Users of the system are the most common type of untrusted source, though tainted data may also originate from other sources, such as a database or a file. Tainted data should pass through a *sanitization routine* to cleanse it from potentially harmful content, before it is passed onto a *sensitive sink*, or vulnerable part of the program. Failure to do so results in a taint-style vulnerability, a weak spot in the program for malicious users to exploit. Which data sources are tainted, which sinks are sensitive and what kind of sanitization should be used depends on the context. Each type of taint-style

vulnerability typically has its own set of sources, sinks and sanitization routines.

Certain scripting languages (most notably Perl) have a special *taint mode* that considers every user-supplied value to be tainted and only accepts it as input to a sensitive sink function if it is explicitly untainted by the programmer. In Perl, this has to be done through the use of regular expressions that accept only input matching a specific pattern [53][32, Sec.1.3]. Variables filtered through a regular expression match will be considered safe by the Perl interpreter. This strategy has several major drawbacks. First of all, as also mentioned by Jovanovic et al. [12], custom sanitization using regular expressions is a dangerous practice, as regular expressions are very complex, and it is easy for programmers to make mistakes. Secondly, a lazy programmer can easily match tainted data to a pattern such as `/(.*)/`, which will match *any* input, effectively circumventing Perl's taint mode.

In deciding whether data is tainted or untainted, Perl's taint mode makes many assumptions. For instance, on the use of regular expressions to sanizite data, Perl's documentation mentions that "Perl presumes that if you reference a substring using $1, $2, etc., that you knew what you were doing when you wrote the pattern" [53]. This is of course a very large assumption and one that is certainly not true in all cases. Programmers often do not know what they are doing and even if they do, it is very easy to make mistakes when writing regular expressions. The risk of the assumptions and generalizations that Perl makes here is that its taint mode gives programmers a false sense of security.

The most prevalent and most exploited vulnerabilities in web applications are *cross-site scripting* (XSS) and *SQL injection* (SQLI). According to a top ten composed by the Open Web Application Security Project (OWASP), XSS and SQLI were the top two most serious web application security flaws for both 2007 and 2010 [66]. According to this list, the top five of security flaws has not changed over the past three years.

Vulnerabilities occurring in real-world applications are much more complicated and subtle than the examples appearing in this section. In many cases, user data is utilized in applications in ways that appear to be safe on the surface. However, due to complex interactions that are difficult to predict, unsafe data can still slip through in specific edge cases. Such vulnerabilities are hard to spot, even when using professional coding standards, careful code reviewing, and extensive testing.

4.2 Cross-site scripting

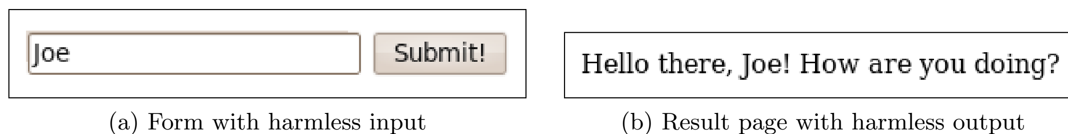
Cross-site scripting (XSS) is a type of vulnerability that allows attackers to inject unauthorized code into a web page, which is interpreted and executed by the user's web browser. XSS has been the number one web application vulnerability for many years, and according to WhiteHat Security, has been responsible for 66% of all website attacks in 2009 [50].

Web pages can include dynamic code written in Javascript to allow the web page's content to be altered within the web browser as the user interacts with it. Normally, a web browser will only execute Javascript code that originates from the same domain as the web page itself, and that code is only executed within a self-contained sandbox environment. This is the so-called Same Origin Policy [42]. This policy prevents attackers from making web browsers execute untrusted code from an arbitrary location.

```
<html>
<body>
<?php
// Retrieve the user's name from a form
$name = $_POST['name'];
// Print the user's name back to them
echo "Hello there, $name! How are you doing?";
?>
</body>
</html>
```

Listing 1: Example of an XSS vulnerability

XSS vulnerabilities allow attackers to inject Javascript code directly into a web page, making it appear to originate from the same source as the web page. Take for example the PHP code snippet in Listing 1. Normally, this page is accessed through a form, where the user enters their name (Fig. 2a) and after clicking the submit button, they are redirected to a page with a simple response message containing their name (Fig. 2b).



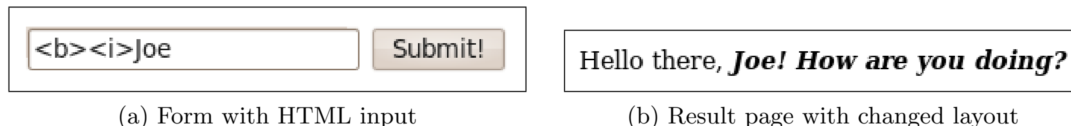
(a) Form with harmless input

(b) Result page with harmless output

Figure 2: Example of an HTML form and a dynamically generated result page

Because the name entered in the form is copied verbatim into the resulting HTML code, it is possible to add additional HTML code to the result page by entering it as a name.

For example, the name shown in Figure 3a could be entered into the form. This name includes the HTML tags for bold-face (<b>) and italic (<i>) text. These tags are copied verbatim into the result page and consequently are interpreted by the browser as HTML, resulting in the name being printed in bold and italic (Fig. 3b). Even worse, because the bold and italic tags are not closed off, the remaining text on the page is printed bold and italic as well. An XSS vulnerability in one part of a web page can affect other parts of the same web page as well.



(a) Form with HTML input

(b) Result page with changed layout

Figure 3: Example of web page layout changed through XSS

Changing the layout of a web page is still an innocent action. It gets more dangerous once we use this vulnerability to add Javascript code to the web page. For example, we can enter the following line in the form:

```
<img src=nonexistent onerror=alert(String.fromCharCode(88,83,83));>
```

This will add an HTML image tag to the web page that points to a file that does not exist. This triggers an error in the user's browser, which is caught by the `onerror` event handler in the image tag. This in turn executes the embedded Javascript code, which pops up an alert dialog containing the ASCII characters 88, 83 and 83 ('X', 'S' and 'S' respectively), as seen in Figure 4.

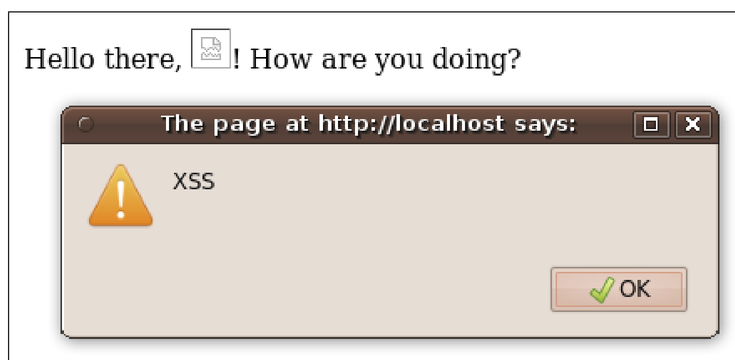


Figure 4: Example of Javascript code added through XSS

To the web browser, this added image tag and the embedded Javascript code appear to be normal parts of the web page, and it will process them without objection. Again, popping up an alert dialog is a relatively innocent example, but Javascript allows near

full control over the user's web browser. Coupled with the fact that the Same Origin Policy is circumvented, this means that the possibilities for an attacker are near limitless.

Cross-site scripting is often used to steal login information from a user, which is stored by the user's web browser in what is known as a *cookie*. The following input is a simple example of such an attack that would work with our vulnerable form:

```
<script>document.location='http://www.xss.com/cookie.php?'+document.
    cookie</script>
```

The HTML code of the response page then becomes:

```
<html>
<body>
Hello there, <script>document.location='http://www.xss.com/cookie.php
    ?'+document.cookie</script>! How are you doing?
</body>
</html>
```

This added Javascript code takes the user's session information (`document.cookie`) and sends it to a script on the attacker's site (`www.xss.com`). The attacker can then use this session information to steal the user's login credentials and perform actions on behalf of the user.

To prevent this XSS vulnerability, the programmer of the code in Listing 1 should have sanitized the user input from line 2 before it is used in the output on line 4. In PHP, the built-in function `htmlspecialchars` does just that. Characters that have a special meaning in HTML are encoded (e.g. ‘`<`’ is encoded as ‘`<`’), so that they are not interpreted by the web browser as HTML code anymore. When the input of Figure 3a is sanitized, instead of getting a bold and italic name, we get the result page as shown in Figure 5, which is the result we originally expected.

Hello there, <b><i>Joe! How are you doing?

Figure 5: Example result page with sanitized input

Disabling Javascript in the user's web browser is a possible client-side preventive measure for most cross-site scripting attacks. However, because many modern web sites rely on Javascript to function properly and break when it is disabled, many users are reluctant to do so. Besides that, disabling Javascript on the client side does not actually solve the cause of the problem, which is in the server-side code.

How an XSS vulnerability can be exploited and delivered to a victim depends on the nature of the vulnerability. A *reflected* XSS vulnerability such as the one described above seems harmless at first glance; a user can only compromise their own security by entering injection code into a form or URL by themselves. However, an attacker can easily lure an unsuspecting user into clicking on a malicious link by concealing it on a website or in an e-mail that the user trusts.

A *stored* XSS vulnerability occurs when the web server saves tainted data into a file or database and subsequently displays this on web pages without proper sanitization. This means an attacker can inject malicious data into a web page once, after which it will permanently linger on the server and is returned automatically to other users who view the web page normally. It is not necessary for an attacker to individually target their victims or to trick them into clicking on a link. An example of a potentially vulnerable application is an online message board system that allows users to post messages with HTML formatting, which are stored and rendered for other users to see.

4.3 SQL injection

SQL injection is a taint-style vulnerability, whereby an unsafe call to a database is abused to perform operations on the database that were not intended by the programmer. WhiteHat Security's report for 2009 lists SQL injection as responsible for 18% of all web attacks, but mentions that they are under-represented in this list because SQL injection flaws can be difficult to detect in scans [50].

SQL, or Structured Query Language, is a computer language specially designed to store and retrieve data in a database. Most database systems (e.g. MySQL, Oracle, Microsoft SQL Server, SQLite) use a dialect of SQL as a method to interact with the contents of the database. Scripting languages such as PHP offer an interface for programmers to dynamically construct and execute SQL queries on a database from within their program.

It is common practice for programmers to construct dynamic database queries by means of string concatenation. Listing 2 shows an example of an SQL query that is constructed and executed from a PHP script using this method. The variable `$username` is copied directly from the input supplied by the user and is pasted into the SQL query without modifications.

Although this query is constructed and executed on the server and users can not directly see the vulnerable code, it is possible through a few assumptions and educated guesses to find out that there is a database table called `users` and that the entered user name

```
1 // Obtain a user ID from the HTTP request
2 $username = $_GET['username'];
3 // Create a query that requests the user ID and password for this
     user name
4 $query = "SELECT id, password FROM users WHERE name='".$username."'";
5 // Execute the query on an open database connection
6 mysql_query($query);
```

Listing 2: Example of an SQL query constructed through string concatenation

is used to query this table. Further experimentation will reveal whether that query is indeed vulnerable to SQL injections.

If all users were to honestly enter their user name as is expected from them, then nothing would be wrong with this query. However, a malicious user might enter the following as their user name:

```
'; DROP TABLE users; --
```

If this text is copied directly into the SQL query as done in Listing 2, then the following query is executed on the database:

```
SELECT id, password FROM users WHERE name = ''; DROP TABLE users; -- '
```

The quotation mark and semicolon in the user name close off the `SELECT` statement, after which a new `DROP` statement is added. The entered user name essentially 'breaks out' of its quotation marks and adds a new statement to the query. The closing quotation mark in the original query is neutralized by the double hyphen mark, which turns the rest of the query text into a comment that is ignored by the SQL parser. The result of this is that instead of requesting information from the `users` table as intended by the programmer, the entire `users` table is deleted from the database.

An SQL injection vulnerability can be fixed by sanitizing the user input with the appropriate sanitization routine before it is used in a query. In the case of Listing 2, this means replacing line 2 with the following code:

```
$username = mysql_real_escape_string($_GET['username']);
```

By using this sanitization routine, all the special characters in the user name are escaped by adding an extra backslash in front of them. This cancels out their special function and consequently they are treated as regular characters. The query that is executed by the database then becomes as follows:

```
SELECT id, password FROM users WHERE name = '\'; DROP TABLE users; --'
```

Although the difference is subtle, the addition of a backslash before the quotation mark in the user name ensures that the SQL parser treats this quotation mark as a regular character, preventing the user input from breaking off the `SELECT` statement. The `DROP` statement, the semicolons and the double hyphens are now all treated as parts of the user name, not as part of the query's syntax.

Dangers with sanitized data

Even user input that has been sanitized can still be dangerous if the SQL query that uses it is poorly constructed. Take for example the code snippet in Listing 3. At first sight, this code appears to execute a valid SQL query with properly sanitized input.

```
$id = mysql_real_escape_string($_GET['id']);
mysql_query("SELECT name FROM users WHERE id=$id");
```

Listing 3: Example of unsafe usage of sanitized input

However, closer inspection reveals that in the construction of the query string, variable `$id` is not embedded in single quotes, which means that malicious input does not have to 'break out' in order to change the query's logic. Such a flaw can occur when the programmer expects the query parameter `id` to be a numerical value, which does not require quotation marks, but fails to recognize that PHP's weak typing allows variable `$id` to be a string as well. Setting the HTTP request variable `'id'` to the value "`1 OR 1=1`" would result in the following SQL query being executed:

```
SELECT name FROM users WHERE id=1 OR 1=1
```

The addition to the `WHERE` clause of the `OR` keyword with an operand that always evaluates to `TRUE` means that every row from table `users` will be returned, instead of a single one.

Prepared statements

SQL injection can be prevented by using prepared statements. While the use of prepared statements is considered to be good practice, it is not common practice yet [9]. For example, in PHP5 the query in Listing 3 should be replaced by the the code snippet in Listing 4 (assuming variable `$db` exists and refers to an open database connection).

```
$id = $_GET['id'];
$statement = $db->prepare("SELECT name FROM users WHERE id = ?");
$statement->bind_param("i", $id);  // The first argument sets the
    data type
$statement->execute();
```

Listing 4: Using prepared SQL statements in PHP

This method of constructing and executing SQL queries automatically checks each input parameter for its correct data type, sanitizes it if necessary and inserts it into the statement. At the same time, the statement string does not require any specific punctuation marks to surround the input parameters, preventing subtle errors such as the one in Listing 3. However, consistent usage of prepared statements to construct queries instead of string concatenation requires drastic changes in the habits of many programmers, and there is already a large amount of legacy code currently in use that would have to be patched [9].

Thomas et al. developed a static analysis tool that scans applications for SQL queries constructed through string concatenation, and automatically replaces them with equivalent prepared statements [13]. This solution does not actually detect SQL injection vulnerabilities in the program code, but it does remove these vulnerabilities from the code, as prepared statements are immune to SQL injection.

4.4 Other vulnerabilities

There are many other types of code injection vulnerabilities and attacks in existence, such as XPath injection, Shell injection, LDAP injection and Server-Side Include injection [10, Sec.4][67]. All of these belong to the general class of taint-style vulnerabilities and rely on the same basic principles as XSS and SQL injection: data coming from an untrusted source is left unchecked and used to construct a piece of code, allowing malicious users to insert new code into a program.

5 Static analysis

Static analysis is an umbrella term for many different methods of analyzing a computer program's source code without actually executing it. Executing a program will make it run through a single path, depending on the input during that execution, and consequently tools that analyze a program while it is running (dynamic analysis) can only examine those parts of the program that are reached during that particular run. Static analysis allows one to examine all the different execution paths through a program at once and make assessments that apply to every possible permutation of inputs.

5.1 Applications of static analysis

Static analysis is widely used for a variety of goals [1, Sec.2.2]. In general, any tool that examines the source code of a program without executing it, can be categorized as a static analysis tool.

Syntax highlighting. Many advanced text editors and integrated development environments (IDE) highlight keywords and syntactical constructs in program code for a given programming language. Syntax highlighting gives programmers a better overview of their program and makes it easier to spot typing errors.

Type checking. Compilers for typed programming languages such as C or Java check at compile-time whether variables are assigned values of the right data types, preventing run-time errors. Many IDEs can also perform basic type checking on source code while it is being written.

Style checking. This is used to enforce programming rules, such as naming conventions, use of whitespace and indentations, commenting and overall program structure. The goal is to improve the quality and consistency of the code. Examples of style checking tools are PMD (`http://pmd.sourceforge.net`) and Parasoft (`http://www.parasoft.com`).

Optimization. Program optimizers can use static analysis to find areas in a program that can execute faster or more efficiently when it is reorganized. This can include unrolling of loops or reordering CPU instructions to ensure the processor pipeline remains filled. Biggar and Gregg use static analysis to optimize PHP code in their PHP compiler tool phc [29].

Program understanding. Static analysis can be used to extract information from source code that helps a programmer understand the design and structure of a program, or to find sections that require maintenance. This includes calculating code metrics such as McCabe's cyclomatic complexity [38] and NPATH complexity [41]. Program refactoring and architecture recovery tools also fall in this category. The Eclipse IDE (`http://www.eclipse.org`) is capable of refactoring Java code. Softwarenaut (`http://www.inf.usi.ch/phd/lungu/softwarenaut`) is an example of a tool that employs static analysis for software architecture recovery.

Documentation generation. These tools analyze source code and annotations within the source code to generate documents that describe a program's structure and programming interface. Examples of such tools are Javadoc (`http://java.sun.com/j2se/javadoc`), Doxygen (`http://www.doxygen.org`) and Doc++ (`http://docpp.sourceforge.net`).

Bug finding. This can help point out possible errors to the programmer, for instance an assignment operator (`=`) used instead of an equality operator (`==`), or a memory allocation without a corresponding memory release. Examples of bug finding tools are FindBugs (`http://www.findbugs.org`) and CppCheck (`http://cppcheck.wiki.sourceforge.net`).

Security review. This type of static analysis is related to bug finding, but more specifically focuses on identifying security problems within a program. This includes checking whether input is properly validated, API contracts are not violated, buffers are not susceptible to overflows, passwords are not hard-coded, etc. Using static analysis to find security problems is the main interest of this thesis.

All of these applications of static analysis have one thing in common: they help programmers to understand their code and to make it easier to find and solve problems.

Tools that use static analysis vary greatly in speed, depending on the complexity of their task [1, pg.38]. Syntax highlighters and style checkers perform only simple lexical analysis of the program code and can do their work in real-time. This makes it possible for them to be integrated in an IDE, much like a spell checker is integrated into a word processor. Conversely, inspecting a program's security requires a tool to understand not only the structure of the program, but also what the program does and how data flows through it. This is a highly complicated task that may take up to several hours to complete. In general, the more precise the static analysis technique, the more computationally expensive it is.

5.2 History

Static analysis tools have been used for a long time to search for bugs and security problems in programs, but only recently have they become sophisticated enough that they are both easy to use and able to find real bugs [60].

The earliest security tools, such as RATS, ITS4 and Flawfinder, were very basic in their functionality. These tools were only able to parse source files and look for calls to dangerous functions. They could not check whether these functions are actually called in such a way that they are vulnerable; these tools could only point out to a programmer that they should carefully inspect these function calls in a manual code review [1, pg.33]. These tools were effectively no more than a glorified version of `grep`.

Static analysis security tools became more useful when they started looking at the context within a program. Adding context allowed tools to search for problems that require interactions between functions. For example, every memory allocation requires a corresponding memory release, and every opened network connection needs to be closed somewhere.

The next evolution in static analysis came from extracting semantical information from a program. Program semantics allow a tool not only to see the basic structure of a program, but also to understand what it does. This makes it possible for a tool to understand the conditions under which a problem may occur and to report problems that require specific knowledge about a program's functionality and the assumptions that the program makes.

Current development in static analysis for security review focuses mostly on improving both accuracy and efficiency of the tools. Static analysis security tools for compiled and statically-typed languages such as C and Java have already reached a high level of maturity.

5.3 Steps

Though there are many different techniques for static analysis of source code, analysis processes that target code security can all be divided roughly into three steps [1, Ch.4]: model construction, analysis and results processing. Security knowledge is supplied during the analysis step in the form of rule sets.

5.3.1 Model construction

A static analysis tool has to transform source code into a program model. This is an abstract internal representation of the source code. This step shares many characteristics with the work that compilers typically perform [1, Sec.4.1].

The quality of a tool's analysis is largely dependent on the quality of the tool's program model [1, pg.37]. If for example a tool is incapable of producing precise information on the use of pointers in a program, then the analysis step will incorrectly mark many objects as tainted and hence report many false positives [30]. It is very important that an analysis tool has a good understanding of the language's semantics in order to build an accurate program model.

The construction of a program model can be broken up into a number of steps.

Lexical analysis. The tool strips the source code of unimportant features, such as whitespace and comments, and transforms it into a series of tokens. The earliest and simplest static analysis tools, such as RATS, ITS4 and Flawfinder, only perform lexical analysis on the source code and look for specific tokens that indicate an unsafe language feature is used.

Parsing. The series of tokens is transformed into a tree structure using a *context-free grammar*. The resulting *parse tree* is a hierarchical representation of the source code.

Abstract syntax tree. The parse tree created by the previous step is stripped of tokens that exist only to make the language syntax easier to write and parse. This leaves a tree structure representing only the significant parts of the source code, called an *abstract syntax tree*, which is simpler to analyze than the parse tree.

Semantic analysis. The analysis tool attributes meaning to the tokens found in the program, so it can for example determine which variables have which types and which functions are called when.

Control flow analysis. The possible paths that can be traversed through each program function are translated into a series of *control flow graphs*. Control flow between functions are summarized in *call graphs*.

Data flow analysis. The analysis tool examines how data moves throughout the program. The control flow graphs and call graphs constructed by the control flow analysis are used for this step [26]. Compilers use data flow analysis to allocate registers, remove unused code and optimize the program's use of processor and memory. Security analyz-

ers use this step to determine where tainted data enters a program and whether it can reach a sensitive sink. For this step to produce reliable results, the analysis tool needs to have a good understanding of the language's pointer, reference and aliasing rules.

5.3.2 Analysis

After the construction of the language model, an analysis step determines the circumstances and conditions under which a certain piece of code will run. An advanced analysis algorithm consists of two parts: an *intraproduceral analysis* component for analyzing an individual function, and an *interprocedural analysis* component that analyzes the interaction between functions [1, Sec.4.2]. We will discuss those two components separately.

Intraprocedural analysis (or local analysis) involves tracking certain properties of data within a function, such as its taintedness and type state, and asserting conditions for which a function may be called safely. The simplest approach is to naively track every property of every variable in every step of the program and asserting these whenever necessary. However, when loops and branches are introduced, the number of paths throughout the code grows exponentially and this naive approach becomes highly impractical.

The key to succesful static analysis therefore lies in techniques that trade off some of their precision to increase their dependability. There are several such approaches to intraprocedural analysis:

- *Abstract interpretation.* Properties of a program that are not of interest are abstracted away, and an interpretation is performed using this abstraction [1, pg.89]. Abstract interpretation can include *flow-insensitive analysis*, where the order in which statements are executed is not taken into account, effectively eliminating the problems introduced by loops and branches. This reduces the complexity of the analysis, but also reduces its accuracy, because impossible execution orders may also be analyzed. WebSSARI uses abstract interpretation as part of its analysis model [8].
- *Predicate transformers.* This approach uses formal methods to derive a minimum precondition required for a function to succeed [1, pg.89]. It starts at the final state of the function (the postcondition) and then works backward through the program to transform this postcondition into the minimum required precondition.

A variant of this approach is *extended static checking*, which is used by the tools Eau Claire [4] and ESC/Java2 [40], amongst others.

- *Model checking.* Both the program and the properties required for the program to be safe are transformed into finite-state automatons, or models [1, pg.90]. These models are checked against each other by a model checker, and if a path can be found in which the safety property's model reaches its error state, then an issue is found. The tool Saturn uses *boolean satisfiability*, a form of model checking [3]. The second version of WebSSARI uses *bounded model checking* as part of its analysis model [8].

Interprocedural analysis (or global analysis) is about understanding the context in which a function is executed. A function might behave differently depending on the global state of the program, or a function might change certain properties of an external variable when called. Interprocedural analysis is needed to understand the effects on data flow that crosses function boundaries. Some tools (e.g. WebSSARI [7, 8]) will ignore interprocedural analysis altogether, assuming that all problems will be found if a program is analyzed one function at a time.

- *Whole-program analysis.* Every function is analyzed with a complete understanding of the context of its calling functions. This can be achieved for example through function inlining, meaning that the bodies of all functions are combined to form one large function that encompasses the entire program. Whole-program analysis is an extreme method that requires a lot of time and memory.

- *Function summaries.* This involves transforming a function into a pre- and a postcondition, using knowledge obtained through intraprocedural analysis. When analyzing a function call, instead of analyzing the entire called function, it is only necessary to look at the function summary to learn the effects that that function will have on its environment.

5.3.3 Rules

The analysis algorithm needs to know what to look for, so it is necessary to define a set of rules that specify the types of flaws and vulnerabilities that the analysis tool should report [1, Sec.4.3]. A rule set can also include information about API functions and the effects they have on their environment. Most static analysis tools come with a predefined

set of rules, specifying the most common flaws and vulnerabilities, and allow the user to customize or extend this rule set. A rule set can be defined in a number of ways:

Specialized rule files. Most static analysis tools use their own custom-designed file format for storing information about vulnerabilities, optimized for their own specific analysis methods. For example, RATS and Fortify SCA both use a custom XML-based rule format, while Pixy makes use of plain text files to define sanitization functions and sinks. Other tools may use a binary file format.

Annotations. Some static analysis tools require the rules to appear directly in the program code, usually in the form of a specialized comment. Annotations can be used to add extra type qualifiers to variables, or to specify pre- and postconditions for a function. Examples of tools that use annotations are Splint [5], Cqual and ESC/Java2 [40]. Documentation generators such as Javadoc and Doxygen also make use of annotations.

Program Query Language. One of the more interesting approaches to defining rule sets is the use of a query language to match bug patterns in a syntax tree or program trace. Examples of program query languages are ASTLOG [47], JQuery [48], Partiqle [49], and PQL [15]. PQL is used in a number of static and dynamic analysis tools, most notably the Java security tools SecuriFly [15], LAPSE [30] and the Griffin Software Security Project [32].

```
query main ()
uses
  object java.lang.String source, tainted;
matches {
  source = sample.UserControlledType.get(...);
  tainted := derivedString(source);
  sample.SystemCriticalSink.use(tainted);
}
executes net.sf.pql.matcher.Util.printStackTrace(*);
```

Listing 5: Example PQL code

Listing 5 shows an example of a PQL code snippet that can be used to find taint-style vulnerabilities within a program. PQL is designed so that a program query looks like a code excerpt corresponding to the shortest amount of code that would violate a design rule. It allows information about vulnerabilities to be expressed with more semantic detail than traditional XML- or text-based rule sets. For example, complex string manipulations can be captured with PQL, and possibly regular expressions as well.

5.3.4 Results processing

The results from the analysis step will contain both false alarms and warnings of different levels of urgency. The next step therefore is to process the results and to present them in such a way that the user is quickly able to spot the most critical flaws and to fix them.

If a problem-free section of code is inappropriately marked as vulnerable, we talk about a *false positive* or false alarm. If a tool fails to identify a vulnerability when in fact there is one, we talk about a *false negative*. Conversely, a correctly identified vulnerability is known as a *true positive*, while an appropriate absence of warnings on a secure section of code is called a *true negative*. False positives are generally seen as intrusive and undesirable and may lead to the rejection of a tool [31]. False negatives are arguably even worse, because they can give the user a false sense of security.

The way in which a tool reports results has a major impact on the value the tool provides. Static analysis tools often generate large numbers of warnings, many of which will be false positives [31]. Part of a tool's job is to present results in such a way that the user can decide which warnings are serious and which ones have lower priority.

Fortify's Audit Workbench groups results into four categories: Low, Medium, High and Critical. The category of a result is chosen based on the severity of the flaw and the confidence of the tool that it was detected correctly [1, Fig.4.10].

In addition to severity, Armorize CodeSecure also ranks vulnerabilities according to their depth, i.e. the number of branches and function calls that tainted data has to traverse to reach the vulnerable section of code. The higher the depth, the less exposed the vulnerability is, and so the less critical the warning is.

Huang et al. attempted to automate patching of vulnerabilities with WebSSARI, by automatically adding sanitization functions to vulnerable sections of code [7]. They claimed the added overhead of this solution was negligible, especially considering that these sanitization functions should be added anyway. However, considering the fact that this approach has not been adopted by other researchers and the feature is not present in WebSSARI's spiritual successor (Armorize CodeSecure) either, one may assume that this solution did not work as well in practice as Huang et al. claimed.

Nguyen-Tuong et al. attempted to automate the process of hardening PHP web applications by modifying the PHP interpreter to include their analysis techniques [9]. In their own words, all they require is that a web server uses their modified interpreter (PHPrevent) to protect all web applications running on the server. While this is a novel

idea that makes it very easy to seamlessly integrate static code security analysis with the normal tasks of a web server, it does have one major drawback that Nguyen-Tuong et al. conveniently overlooked in their article: it requires the authors of PHPrevent to actively support their modified PHP interpreter by keeping it up-to-date with the reference implementation of the PHP interpreter. If not, it becomes outdated quickly and ceases to be a serious option for use on real web servers. Possibly the authors themselves have realized this, because PHPrevent appears to have been taken down from its official web site [65].

5.4 Limitations

A fundamental limitation of static analysis is that it is an inherently undecidable problem. Turing already proved in the 1930's that no algorithm can exist that is capable of deciding whether a program finishes running or will run forever, based only on its description [45]. Rice's theorem expands upon Turing's halting problem and implicates that there is no general and effective method to decide whether a program will produce run-time errors or violate certain specifications [46]. The result of this undecidability is that all static analysis tools will always produce at least some false positives or false negatives [1, pg.35].

When analyzing the data flow and taint propagation within a program, most static analysis tools will assume that a sanitization function always does its work properly, i.e. fully remove any taintedness from an object. However, the sanitization process itself could be incorrect or incomplete [17, 18, 23], leaving data tainted despite having been sanitized. Unless a static analysis tool also analyzes the validity of the sanitization process, this could result in additional false negatives.

Consequently, a clean run from a static analysis tool does not guarantee that the analyzed code is perfect. It merely indicates that it is free of certain kinds of common problems [1, pg.21].

6 PHP

PHP, a recursive acronym for **P**HP **H**ypertext **P**reprocessor (`http://www.php.net`), is a scripting language primarily designed for the development of dynamic web applications. As with most scripting languages, PHP code is typically not compiled to native machine code before it is executed, but rather runs within an interpreter.

Although PHP is available as a stand-alone command-line interpreter, it is mainly used in the form of a plug-in module for web server software packages (such as Apache and Microsoft IIS) to allow dynamic generation of web content. In that role, PHP essentially acts as a filter that takes an HTTP request as input and produces an HTML page as output. PHP is used on many web servers nowadays to implement web applications with complex dynamic behavior and interactions with other systems, as opposed to the simple static web pages that characterized the early days of the world wide web. Large web sites such as Wikipedia, Facebook and Yahoo are built upon PHP scripts. PHP is one of the most popular languages for web application development and as of October 2008, was installed on 35 million web sites run by 2 million web servers [58].

PHP is popular mainly because of its smooth learning curve and the pragmatic approach to programming it provides. Its dynamic and interpreted nature makes programs easier to write and quicker to deploy on a web server than traditional natively compiled plug-in modules. PHP offers an extensive programming library with a large variety of functionality out of the box, such as database bindings, image manipulation, XML processing, web service coupling and cryptography. Its programming interface is also well-documented and many examples and code snippets are available on the official web site.

6.1 Language complexity

Being a dynamic scripting language that is executed by an interpreter gives PHP some exceptional properties that are not shared by statically compiled languages such as C or Java. PHP allows for many exotic constructions that can result in unintended or unexpected behavior.

Unlike most other programming languages, PHP does not have a formal specification. Instead, the language is defined by the main implementation produced by The PHP Group, which serves as a de facto standard [39, 59]. Even though PHP's syntax is well documented and relatively easy to parse [71], the exact semantics of a PHP program

can be difficult to describe. The only complete and accurate documentation of PHP's semantics is the source code of the reference implementation. The lack of a formal specification and the ad-hoc nature of the language's design make it difficult for analysis tools to accurately model the behavior of programs written in PHP.

Biggar and Gregg have written an extensive explanation of all the difficulties that they encountered while attempting to model PHP's complicated semantics in phc [28, 29]. Here follows a short summary of PHP's features that are challenging for static analysis of the language.

Variable semantics. The exact semantics of the PHP language differ depending on the circumstances.

- PHP versions 4 and 5 have different semantics for a number of language constructs that are syntactically identical, and there is no simple way to distinguish between the two. One example of a significant change comes with PHP5's new object model: object-type function arguments are passed by reference in PHP5 as opposed to PHP4, where objects are passed by value. This means that similar-looking code will behave differently between PHP4 and PHP5, and source code written in either version may not be cross-compatible with the other version. While PHP4 is slowly being phased out, there is still a large amount of source code in operation that has been written for PHP4.
- Maintenance releases of PHP regularly bring along bugfixes that subtly alter semantics details of the language. For example, PHP version 5.3.2 changed the way large literal number values are parsed. Integer values above the predefined value `LONG_MAX` are normally converted to floating-point representation. However, in previous versions of PHP, if a large value was written in hexadecimal notation, it would instead be truncated to the value of `LONG_MAX`. Since version 5.3.2, large hexadecimal values are converted normally to floating-point.
- PHP's semantics can be changed externally through its configuration file `php.ini`. For example, the `include_path` flag influences which source files are included at run-time, the `magic_quotes_gpc` flag changes the way user input strings are handled, and it is even possible to let PHP5 behave as PHP4 by changing the `zend.ze1_compatibility_mode` flag.

Run-time inclusion of source files. In PHP, a source file may or may not be included depending on the state of run-time variables and branches in the code. This is often used for localization purposes, where a run-time string value is used to include a specific language file containing localized text.

Run-time code evaluation. PHP's `eval` statement allows a dynamically constructed string value to be interpreted at run-time as a piece of source code. This effectively allows PHP programs to program themselves. Along with run-time source inclusion, this means that the exact source code of a program is not known until the program is executed.

Dynamic, weak and latent typing. A variable's data type can change at run-time (dynamic typing), its type does need not to be declared in the program code (latent typing) and its value can be converted automatically behind the scenes (weak typing). For example, take the following PHP code snippet:

```
if ($v == 0) print $v;
```

At first glance, it seems the only thing this line of code can do is either to print the number 0 to the screen, or to do nothing at all. However, because variables in PHP are weakly typed, the value in variable `$v` is automatically converted to an integer before the comparison with 0. If `$v` contains a string value that can not be converted to a number, this conversion will result in the value 0 and the test passes. Consequently, the above statement will print *any* non-numerical string value to the screen. To prevent this kind of behavior, PHP defines the special operator `===`, which not only compares the operands' values but also their types.

Duck-typing. Fields in an object may be added to and deleted from an object at any time. This means that an object's memory layout is not rigidly specified by its class type and cannot be deduced from its initial declaration only.

Implicit object and array creation. An assignment to an uninitialized array will cause the array to be created implicitly, while an assignment to an uninitialized object will create a new object of the class `stdClass`.

Aliasing. Much like other programming languages, PHP allows variables to reference or alias the value of another variable. However, unlike their counterparts in languages such as Java, C or C++, aliases in PHP are mutable and can be created and destroyed at run-time [26, 29]. Additionally, a function may be called with an aliased argument without the function itself knowing it.

Variable variables. The string value of one variable can be used to index another variable. This is made possible by PHP's use of a symbol table to store variables, instead of rigid memory locations. This symbol table can be indexed with any string value, even ones that are constructed at run-time.

String manipulation. This is an issue common for most programming languages, and one that makes accurate data flow analysis considerably more difficult. PHP offers an extensive library for manipulating character strings, which includes functions for concatenation, substring selection, and regular expression matching. This makes it possible for strings to be only partially tainted, and for tainted parts to be removed from a string [9, 16]. An accurate static analysis tool would need to know the exact semantics of every string library function and precisely track the taintedness of strings per individual character.

Regular expressions. String manipulation through regular expression matching complicates analysis even further, because it requires an analysis tool to also analyze the regular expression itself and decide whether or not it will replace tainted characters with untainted ones. In contrast, Perl's taint mode simply assumes every regular expression will always sanitize every string [53], even though it is easy to prove that this is a false assumption.

All these features added together mean that even a simple statement can have hidden semantics which are difficult to see at first glance.

PHP requires its own unique set of static analysis methods and tools have to be specifically adapted for this language. A C++ static analysis tool such as CppCheck would not be fit to check PHP code, at least not without a major overhaul. It is more efficient to build a completely new PHP source analyzer from the ground up than it is to convert for example a C++ source analyzer to support PHP. In general, static analysis methods and tools are not directly interchangeable between programming languages.

A taint mode similar to the one in Perl is being considered for inclusion in PHP [54], but has been rejected several times in the past, with the argument that it would require too much knowledge about the application context to be of any use [55, 56].

7 Problem statement

The goal is to find a static code analysis solution for PHP that has the potential to be deployed in a live web hosting environment as an automated quality assurance tool. This leads to an additional step in the web application architecture from Figure 1, in-between the uploading of a PHP script and its deployment. This additional step is illustrated in Figure 6. Customers should be able to upload PHP scripts to the web server through a special interface, after which these scripts are automatically analyzed. If a script is found to be vulnerable, then it is rejected or quarantined and a report is sent back. If the scripts are found to be free of serious vulnerabilities, then they are deployed onto the web server, ideally without the customer noticing the analysis step.

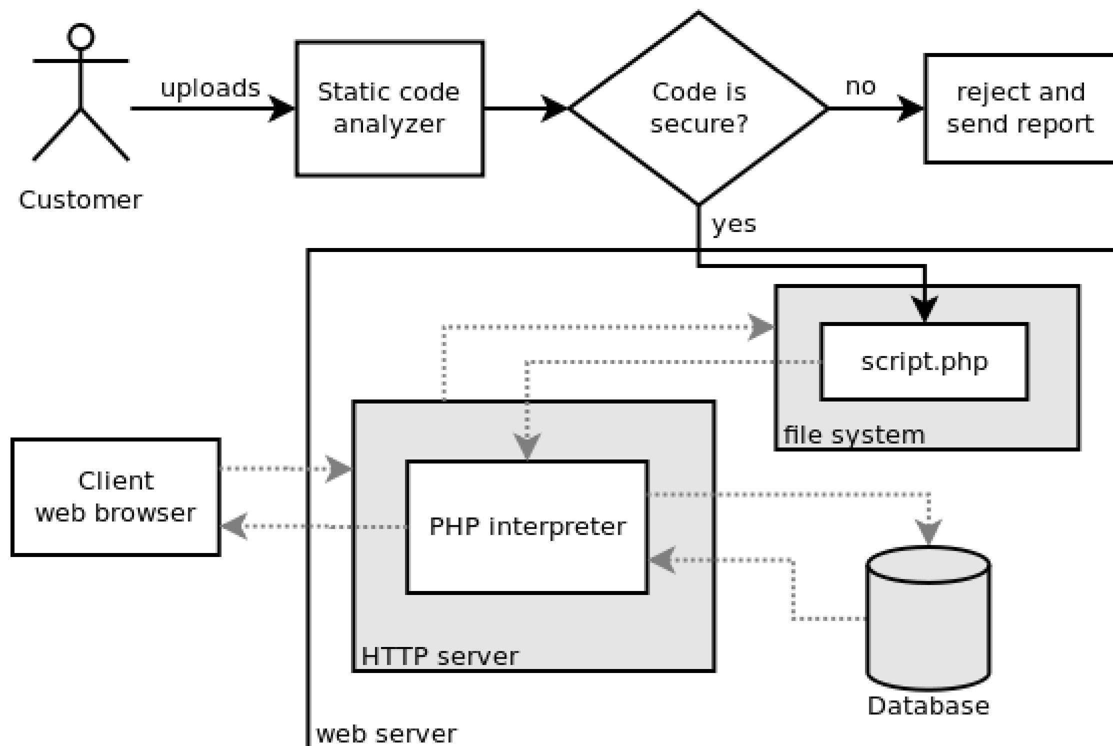


Figure 6: Web application architecture with added static analysis step

This concept imposes a number of requirements on the static code analysis tool that is used. We can view these requirements as a set of both quantitative and qualitative properties that each candidate tool will be judged on.

Accuracy. We require a tool that accurately models the syntax and semantics of PHP and that can precisely trace data flow and taint propagation within a program. The tool should be able to distinguish accurately between code that is vulnerable and code that is not vulnerable. The results it produces should contain little to no false positives and false negatives.

Robustness. The tool should not randomly crash or silently fail while performing its analysis. It should be capable of analyzing large applications without stumbling on irrecoverable problems. The tool must always finish its analysis and produce a result, even if it encounters a problem.

Responsiveness. As a tool for developers, static code analysis is used only incidentally, while in our situation it will be used more regularly and systematically. Security review using static code analysis is a complex process that consumes a considerable chunk of system resources. Even though the analysis of a program is a one-time process, in a real-time environment CPU time and memory consumption are always factors to take into consideration.
Ideally, we want the security review process to be transparent and close to real-time. To this end, we need to know how long it typically takes for a tool to fully analyze an application, how much system memory it uses in the process, how consistent these values are over multiple analysis runs, and how predictable the running time of analysis process is.

Usability. The analysis process should require little to no intervention from users or administrators. Users should not be assumed to have detailed knowledge about the structure or the inner workings of the source code. Users should only be informed about an issue in the source code if the analyzer is absolutely sure that the detected vulnerability poses a serious threat. Additionally, users should be assisted in fixing any vulnerabilities that are found, or vulnerabilities should be patched automatically.

Configurability. Configuration should be a simple one-time process that is independent of the programs that are analyzed. Preferably, a solution should be able to perform analysis out-of-the-box, without requiring manual entry of all the appropriate sources, sinks and sanitization functions. This immediately rules out any solutions that require annotations in the source code, since we can not expect customers to properly annotate every function in their code.

Openness. There needs to be openness about the implementation of a solution. This does not necessarily mean that the tools in question should be open source; rather, its creators should be open about the methods that they use and how they are implemented. If the creators of a solution refuse to disclose the details of how their solution works, then it is impossible to verify the correctness of the tool's methods. Open source tools have the added advantage that it is possible to make modifications that may be needed to integrate them in an existing tool chain. In the event that the creators of a tool decide to abandon it, an open source tool allows independent third parties to fork the source code and continue the development of the tool.

Development and support. The solution needs to be actively developed and supported. Both the PHP programming language and its applications are continuously evolving, as well as the techniques to analyze them. If a tool does not receive regular updates to keep up with these developments, it will become outdated quickly and lose its usefulness.

8 Testing methodology

In order to find out how well the available static analysis tools for PHP conform to the requirements laid out in Section 7, we need to formulate a series of objective tests that allows us to evaluate and compare these tools. This series of tests is called a benchmark, and needs to meet a number of criteria itself for it to be relevant.

In this section, we discuss the definition of a benchmark and the criteria it needs to meet, which benchmarks already exist for static analysis tools, the goals that we intend to reach with our own benchmark, and finally the formulation of a benchmark that we will use in our evaluation.

8.1 Benchmarking criteria

Sim et al. define a benchmark as a test or set of tests used to compare the performance of alternative tools or techniques [33, Sec.3.2]. A benchmark consists of three parts:

Motivating Comparison. This includes a clear definition of the comparison to be made, as well as a motivation of why this comparison is important.

Task Sample. A benchmark includes a selection of tasks that are representative for the entire domain that the benchmark covers.

Performance Measures. A number of quantitative or qualitative measures that are used to determine a tool or technique's fitness for its purpose.

A good benchmark can help to advance a research community by increasing its cohesiveness, providing a template against which new research results can be tested and compared [33, Sec.2].

A good benchmark should be objective. It is only human nature to select test cases that skew the results in favor of a hypothesis, demonstrating the good qualities of a tool or technique, while avoiding tests that do the opposite [34]. This is especially true for benchmarks designed for business or marketing purposes [33].

A benchmark can only make an impact and become an industry standard if it has been developed through collaboration and is reviewed by a large number of participants from various backgrounds. Benchmarks created by a single individual or company are not likely to be used widely and thus will not have a big impact on the research community [33].

Sim et al. present seven desiderata for a benchmark to be successful [33, Sec.4.3]:

Accessibility. The benchmark should be easy to obtain and easy to use.

Affordability. The costs of the benchmark should be proportional to the benefits. This includes human resources, as well as software and hardware costs.

Clarity. The benchmark specification should be easy to understand and unambiguous.

Relevance. The tasks should be representative for the entire domain that the benchmark applies to.

Solvability. The task set should not be too difficult for most tools to complete.

Portability. The benchmark should be abstract enough that it can easily be translated to other tools or techniques.

Scalability. The benchmark should be applicable to tools and techniques of various sizes and levels of maturity.

These are the criteria that we keep in mind while designing and building our own benchmark.

8.2 Static analysis benchmarking

By far the most common method to evaluate a static analysis tool is to run it on a series of real-world applications and manually inspect the results. Every single paper referenced in this article that describes the design of a static analysis tool uses this evaluation method. There appears to be no consensus on which real-world applications should be present in a test set, as each paper presents its own selection of test applications. Consequently, none of these papers contain any direct quantitative comparisons between different static analysis tools. One minor exception is the article by Huang et al. [7], which provides a comparison of feature sets between WebSSARI and other code analysis tools.

It is difficult to objectively benchmark static analysis tools because there is no widely agreed-upon yardstick for comparing results. Every static analysis tool makes a trade-off between false positives and false negatives, and there is no consensus on which trade-offs are better. Chess and West advise to evaluate static analysis tools using a real body of code that the user knows well, and to compare results based on one's own particular needs [1, pg.41].

There have been a number of independent initiatives to construct an objective benchmark for evaluating static analysis tools.

- The SAMATE (Software Assurance Metrics and Tool Evaluation) project at NIST (National Institute of Standards and Technology) [74] includes a Reference Dataset of nearly 1800 test cases for use in the evaluation of static analysis tools. Of those, currently only 18 test cases are written for PHP.
- Chess and Newsham developed a prototype benchmark for static analysis tools for C source code called ABM (Analyzer BenchMark), which includes a combination of micro- and macro-benchmarks [35]. The test cases selected for these benchmarks have been donated to the NIST SAMATE project.
- Livshits developed a pair of benchmarks for Java static and dynamic analyzers. SecuriBench [75] is a collection of open source web-based applications written in Java, which are known to suffer from a variety of vulnerabilities. SecuriBench Micro [76] is a series of small Java programs designed to stress different parts of a static security analyzer.
- OWASP designed WebGoat [68], which is a deliberately insecure Java web application designed to teach web application security lessons. It has also been used on a number of occasions to evaluate the performance of a static analysis tool.
- Walden et al. tested the effect of code complexity on the success rate of Fortify SCA's analysis of C applications [36]. They measured the complexity of an application using a combination of source lines of code and McCabe's cyclomatic complexity measure [38].

8.3 Motivating comparison

The motivating comparison of our benchmark is to find which static analysis tool for PHP web application security review offers the best balance between precision, resource consumption and user-friendliness, while also being completely robust. This benchmark is intended to be a general test of a tool's performance, but it can also be used specifically to find a tool which meets the requirements presented in Section 7.

8.4 Task sample

There is currently no objective, public and peer-reviewed evaluation method for PHP static analysis tools. We therefore have to construct our own benchmark suite from scratch. This benchmark consists of two parts: a macro-benchmark consisting of a representative selection of real-world PHP web applications, which gives us information about the general performance and robustness of a tool, and a micro-benchmark consisting of a set of small synthetic test cases, which highlights specific characteristics of a tool.

While this benchmark conforms to the seven benchmark desiderata defined by Sim et al. [33], it is not peer-reviewed nor constructed through consensus. This is an unavoidable consequence of the fact that there is no precedent in PHP static analysis tool evaluation, combined with the limited scope of this project. This benchmark can be considered objective, as we have no specific bias towards a single static analysis tool.

Macro-benchmark

The macro-benchmark consists of a selection of real-world web applications written in PHP. These applications are selected to be representative for a large variety of PHP web applications. They vary in their application type, minimum supported version of PHP, code size, code complexity, coding style, age, maturity and acceptance. The selection is made without any bias towards a single static analysis tool. The full list of applications selected for the macro tests, along with their basic properties, is shown in Table 1.

For each of these applications, we measured the code size and code complexity using PHP Depend by Manuel Pichler (`http://pdepend.org`). Code size is expressed in lines of pure PHP code, i.e. without comments, empty lines and HTML markup. Code complexity is measured by McCabe's cyclomatic complexity number [38], which is essentially the total number of branch points (`if`, `while`, `switch`, etc) within the code. Walden et al. previously used McCabe's cyclomatic complexity to measure the relation between code complexity and the rate of false positives and -negatives produced by a static analysis tool [36]. We will mainly use the code size and code complexity measures to see if they can be related to the time needed by a tool to analyze an application.

Two of the applications were selected because are known to contain specific vulnerabilities that have been exploited for actual attacks. Using the knowledge of these vulnerabilities, we can look at the analysis results of a tool and see if these vulnerabilities are

Application	Version	Application type	PHP version	Entry file	Lines of code	Cyclomatic complexity
DokuWiki	2009-02-14b	Wiki	4.3.3	doku.php	57871	9093
eXtplorer	2.0.1	File manager	4.3	index.php	30272	3501
Joomla	1.5.15	CMS	4.3.10	index.php	204967	21599
MediaWiki	1.15.2	Wiki	5.0	index.php	180445	24035
Phorum	5.2.14	Message board	5.2	index.php	43288	3814
phpAlbum	0.4.1.14	Image gallery	4.3.3	main.php	5755	1105
phpBB	2.0.10	Message board	3.0.9	index.php	20946	1569
PHP-Fusion	7.00.07	CMS	4	news.php	16693	1917
phpLDAPadmin	1.1.0.7	LDAP manager	5.0.0	htdocs/index.php	23778	3589
phpMyAdmin	3.3.0	DB manager	5.2.0	index.php	124830	13203
phpPgAdmin	4.2.2	DB manager	4.1	index.php	45111	5845
Php-Stats	0.1.9.2	Website stats	4.0.6	php-stats.php	14008	2502
Serendipity	1.5.2 LITE	Weblog	4.3.0	index.php	89483	9840
SquirrelMail	1.4.20	Webmail	4.1.0	src/index.php	46953	4261
WordPress	2.9.2	Weblog	4.3	index.php	132201	17327
Zenphoto	1.2.9	Image gallery	4.4.8	index.php	62532	9028

Table 1: Macro-benchmark test applications

actually reported. We can also evaluate the ease with which these vulnerabilities can be spotted within a tool's analysis report, and if they stand out in some way.

- phpBB version 2.0.10 is known to contain a vulnerability in the file `viewtopic.php` that allows arbitrary PHP code to be executed on the server, which has led to the creation of the Santy worm [51]. This vulnerability involves an improper combination of the `htmlspecialchars` and `urldecode` functions, along with dynamically evaluated PHP code embedded within a call to `preg_replace` (which performs a search-and-replace on a string using regular expression matching). It is a very difficult vulnerability to spot through manual code review and should be a good challenge for static analysis tools.
- phpLDAPadmin version 1.1.0.7 was recently reported to contain a vulnerability in the file `cmd.php` that allows a user to use a null-terminated URL to view the contents of files on the server [52]. This is a more straightforward taint-style vulnerability, involving an unsanitized request variable from the user that is used to dynamically include a source file.

Some tools might perform better on these tests with careful configuration of sanitization rules than in their default configuration. However, we want to see how a tool will generally perform on an arbitrary set of PHP code without any prior knowledge of

the code, so any application-specific configuration of the tools for the macro tests is prohibited.

Micro-benchmark

The micro-benchmark consists of 110 small hand-written PHP programs comprising 55 test cases, each one focusing on a single language feature, vulnerability type or sanitization routine. These programs are designed to stress the analysis tool, presenting it with a number of semantically complex (yet not uncommon) situations and challenging it to assess whether the code contains a vulnerability or not. This benchmark tells us which vulnerabilities a tool can find, how well it can find them, and where the tool's problem areas are. It also gives us an indication of how reliable the warnings are that are produced in the macro-benchmark tests.

The micro-benchmark test cases are structured according to the same pattern used by Zitser et al. [37] and Newsham and Chess [35]. Each test case consists of a `BAD` program that includes a vulnerability (the vulnerable line of code is annotated with the type of vulnerability), and an `OK` program which patches the vulnerability. Static analysis tools are to find the vulnerability in the `BAD` case, and to remain silent for the `OK` case. Test cases are grouped according to their subject, and separated into three overarching categories (*Language support*, *Injection vulnerability detection*, and *Sanitization support*).

The source code of all the micro-benchmark test cases is listed in Appendix B.

8.5 Performance measures

Macro-benchmark

We can use the macro-benchmark test for several purposes. Firstly, we can gather quantitative information about a tool, such as the time it takes to analyze an application, its peak memory usage, the number of errors encountered during analysis, and the number of vulnerabilities it finds. Using these numbers, we can attempt to find a relation between the basic properties of an application and the tool's resource usage, and subsequently draw conclusions about the predictability of the tool's performance. We can also look at the consistency of the tool's results over a number of test runs, both in terms of resource usage and vulnerabilities found.

Secondly, a macro-benchmark allows us to make a qualitative evaluation of a tool. This

includes the robustness of the tool when analyzing large projects, as well as the features offered by the tool for processing the results and the ease of use of its interface. By analyzing the result reports of a tool in combination with prior knowledge of the macro test applications, we can judge the effectiveness of the tool in helping a developer to find vulnerabilities.

Newsham and Chess chose their macro-benchmark test case specifically for its maturity and assume that relatively few vulnerabilities are left in its code [35, Sec.3.3]. For simplicity's sake, they subsequently assume that all vulnerabilities found in their macro-benchmark can be considered as false positives. While this is a highly controversial statement that is impossible to prove, there is a grain of truth in it. For instance, we included MediaWiki in our macro-benchmark. MediaWiki is used as the basis for Wikipedia and its code has been reviewed by thousands of expert programmers over the course of many years, which makes it arguably one of the most mature open source web applications in existence, and is therefore unlikely to contain any severe vulnerabilities. We do not make the same assumption as Newsham and Chess in our evaluation, but it is interesting to keep their statement in mind while reviewing the macro-benchmark test results.

Micro-benchmark

The results of the micro-benchmark tests are processed using the methodology described by Newsham and Chess [35].

When a tool finds a vulnerability in a `BAD` program, this is a true positive and the test is marked as a "pass". When a tool remains silent on an `OK` program, this is a true negative and the test is also marked as a "pass". Conversely, if a tool remains silent on a `BAD` case, this is a false negative and the test fails. If a tool finds a vulnerability on an `OK` program, this is a false positive and the test also fails.

Subject	**BAD tests**			**OK tests**			**Total**			**Discriminates**		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
Injection	7	9	78%	5	9	56%	12	18	67%	4	9	44%

Table 2: Example micro-benchmark test result

For each group of tests, the results are aggregated and a percentage of "pass" results is derived for both `BAD` and `OK` cases. The results of the `BAD` and `OK` tests are added up to a total, along with an average percentage. These results are all gathered in a single

table, such as the example in Table 2.

The "pass" results of the `BAD` and `OK` tests by themselves do not tell us whether a tool passed these tests legitimately. If a tool passes a `BAD` test, it may be because it found an unrelated problem or made a lucky guess without actually understanding the problem. If a tool passes an `OK` test, it may be because it ignored part of the program and thus did not find any problems.

These mistakes can be exposed by looking at the correspondence between the `BAD` and `OK` test results. An overly permissive tool will pass most `OK` tests but fail at the corresponding `BAD` tests, and vice versa for an overly paranoid tool. This is expressed in the results as *discrimination*. A tool passes the discrimination test for a single case if and only if it passes both the `BAD` and `OK` tests for that case. Passing a discrimination test ensures that the tool understands the problem and is able to distinguish between code that is vulnerable and similar code that is not vulnerable.

As with the `BAD` and `OK` test results, the discrimination results are aggregated and a percentage is derived. The overall discrimination percentage tells us how well a tool is capable of finding vulnerabilities and understanding the language.

9 Tool evaluation

For those static code security analysis tools of which a usable implementation is available, we look at how these tools perform on real world applications and how well they model the intricacies of the PHP programming language.

We look at each tool in light of the requirements defined in Section 7. In case a property of a tool can be quantitatively measured, we make use of the benchmarks that we defined in Section 8.

General test setup

All tests were performed on an Intel Core 2 Duo E8400 3.0 GHz processor with 4 GB RAM, running Ubuntu 9.10 64-bit edition. Java-based tools were given a maximum heap size of 3072 MB and a stack size of 16 MB per thread. Analysis time and peak memory usage were measured by running the UNIX tool `ps` every 5 seconds and recording the process's cumulative CPU time and resident set size values at each interval.

In the micro-benchmark tests for each tool, all vulnerability types other than the ones included in the test cases are filtered out of the analysis reports. This prevents unexpected bug reports from contaminating the benchmark results.

9.1 Fortify 360

Fortify 360 by Fortify Software Inc. (`http://www.fortify.com`) is a commercial suite of solutions for software security assurance. Fortify 360's static analysis tool, the Source Code Analyzer (SCA), can detect more than 400 categories of vulnerabilities in 17 different programming languages, amongst which are C++, Java and PHP.

Fortify SCA works by first compiling program code from its original language to a generalized intermediate language and building a program model from that. Analysis is performed on this generalized program model. This allows Fortify SCA to support many different programming languages using only a single language model.

Test setup

Both the micro- and macro-benchmark tests were performed on Fortify 360 Source Code Analyzer version 2.1.0, using the included `sourceanalyzer` command-line tool. For all these tests, the default rule packs for PHP were used that are included with Fortify SCA.

In the macro-benchmark test results (Table 4), analysis time consists of build time and analysis time added together, while peak memory usage is the maximum of these two processes. Only issues labeled 'High severity' and 'Critical' are counted, and added up to comprise the '# issues found' column.

In the micro-benchmark tests, the flow-insensitive nature of Fortify's analysis would cause vulnerabilities to be reported in certain files, while the tainted data source was located in an unrelated file. To prevent this behavior from contaminating the benchmark results, any vulnerabilities where the sink file differs from the source file (with the exception of `include_unsafe.php` and `include_safe.php`) are filtered out of Fortify's report.

Results of the micro-benchmark tests are shown in Table 3 (full results in Appendix A.1). Results of the macro-benchmark tests are shown in Table 4.

Accuracy

Fortify SCA's performance in the language support micro-benchmark (Table 3a) is remarkably poor. Although Fortify finds a good number of vulnerabilities in the `BAD` tests, it also often finds the same vulnerabilities in the equivalent `OK` tests, which severely hurts the discrimination score. Compared to Pixy's results (Table 7a), Fortify's alias analysis performance is shocking. Fortify is unable to distinguish between the `BAD` and `OK` cases on all four Aliasing tests, whereas Pixy achieves a perfect 100% score. The only language support test where Fortify achieves a 100% discrimination score is the 'Strings' test, which can be considered as the easiest test to pass in our benchmark, since even PHP-sat scores perfectly on it (Table 9a).

Fortify SCA's performance on the injection and sanitization micro-benchmarks is respectable. It does not find every type of vulnerability included in the benchmark, but the most common vulnerabilities (XSS and SQLI) are well supported. Fortify also does not recognize sanitization through the use of regular expressions or direct string manipulation, but this was to be expected. Normal sanitization routines for SQL and HTML data are properly recognized.

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	20	30	67%	22	30	73%	42	60	70%	12	30	40%
Aliasing	3	4	75%	1	4	25%	4	8	50%	0	4	0%
Arrays	1	2	50%	2	2	100%	3	4	75%	1	2	50%
Constants	1	2	50%	2	2	100%	3	4	75%	1	2	50%
Functions	3	5	60%	3	5	60%	6	10	60%	1	5	20%
Dynamic inclusion	3	3	100%	2	3	67%	5	6	83%	2	3	67%
Object model	6	8	75%	6	8	75%	12	16	75%	4	8	50%
Strings	3	3	100%	3	3	100%	6	6	100%	3	3	100%
Variable variables	0	3	0%	3	3	100%	3	6	50%	0	3	0%

(a) Language support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	13	18	72%	16	18	89%	29	36	81%	11	18	61%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	2	2	100%	2	2	100%	4	4	100%	2	2	100%
Code injection	2	2	100%	1	2	50%	3	4	75%	1	2	50%
SQL injection	4	6	67%	6	6	100%	10	12	83%	4	6	67%
Server-side include	2	2	100%	1	2	50%	3	4	75%	1	2	50%
XPath injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Cross-site scripting	3	3	100%	3	3	100%	6	6	100%	3	3	100%

(b) Injection vulnerability detection

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	7	100%	3	7	43%	10	14	71%	3	7	43%
Regular expressions	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	1	1	100%	1	1	100%	2	2	100%	1	1	100%
Strings	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Cross-site scripting	2	2	100%	2	2	100%	4	4	100%	2	2	100%

(c) Sanitization support

Table 3: Fortify micro-benchmark test results (abridged)

Application	Lines of code	Cyclomatic complexity	Analysis time (s)	Peak memory usage (MB)	# scan errors	# issues found
DokuWiki	57871	9093	191	2606	0	82
eXtplorer	30272	3501	81	1341	3	255
Joomla	204967	21599	984	3275	17	0
MediaWiki	180445	24035	1300	3264	14	69
Phorum	43288	3814	251	2100	7	1179
phpAlbum	5755	1105	66	1306	0	197
phpBB	20946	1569	553	2561	2	932
PHP-Fusion	16693	1917	312	1567	0	3729
phpLDAPadmin	23778	3589	100	1370	0	113
phpMyAdmin	124830	13203	833	3262	17	1
phpPgAdmin	45111	5845	171	1510	0	664
Php-Stats	14008	2502	177	1436	0	284
Serendipity	89483	9840	573	1934	5	3521
SquirrelMail	46953	4261	126	1424	0	6
WordPress	132201	17327	568	2227	1	572
Zenphoto	62532	9028	633	2412	9	1296

Table 4: Fortify macro-benchmark test results

One thing we observed while inspecting the macro-benchmark reports from Fortify is that DokuWiki often performs custom sanitization of data through use of the `preg_replace` function, which does a search-and-replace using regular expression matching. Fortify SCA does not recognize this type of sanitization, as was already evident from the micro-benchmark results, and thus reports a false positive every time this sanitization technique is used.

Fortify does not report the code injection vulnerability in phpBB2's `viewtopic.php` file. It does find the file inclusion vulnerability in phpLDAPadmin's `cmd.php` file, with High severity, but surprisingly reports the same vulnerability three times.

Robustness

Fortify SCA succeeded in completing analysis on all the macro tests, and produced meaningful results for all applications. Fortify did occasionally fail at parsing individual script files, producing an error message, but without a clear indication of the cause of the error. These script files are skipped and apparently ignored during the rest of the analysis process. The '# scan errors' column in Table 4 includes only errors related to PHP code, but Fortify produced even more errors on the parsing of embedded Javascript code.

Table 4 does show a few suspicious results: Fortify encounters a relatively large number of errors while analyzing both Joomla and phpMyAdmin, and the number of issues found in both applications is very low. The same can be said of MediaWiki, though Fortify does find a believable number of issues in this application. It is possible that Fortify encountered an irrecoverable error while analyzing these applications, but this can not be deduced from the results report.

Responsiveness

Table 4 shows that Fortify SCA has a high memory cost and that analysis for large applications is a time consuming process. Even a relatively small application such as phpAlbum, consisting of a few thousands of lines of code, requires nearly one-and-a-half gigabytes of memory to analyze. For large applications such as Joomla, MediaWiki and phpMyAdmin, the progression of memory usage shown by Fortify is typical for a tool written in Java: the memory usage steadily increases over time, and Java's garbage collector does not reclaim any memory until the heap size limit of 3 gigabytes is reached.

With even the smallest application (phpAlbum) taking more than one minute to analyze, and the largest applications up to 20 minutes, we can safely say that real-time code security review using Fortify SCA on a live web server will be experienced as intrusive. Although these analysis times can be decreased by increasing the processor power, Fortify's analysis process will always take a noticeable amount of time.

Fortify SCA has shown itself to be consistent in its running times and memory usage. Over several macro-benchmark test runs, the variation in those values has been minimal. Fortify also always reported the same number and types of issues within each application.

We are interested in finding out if there is a relation between the metrics of an application's source code and the time it takes Fortify SCA to analyze the code. This can be useful to make predictions about Fortify's running time for specific applications, and to determine how the running time scales as the size of an application grows. To this end, we have plotted the lines of code and analysis time numbers from Table 4 against each other, with the results shown in Figure 7.

The points in Figure 7 suggest that there is a linear relation between code size and analysis time, but with a number of outliers. We have performed a linear regression on this data set, the result of which is visualized by the gray dashed line. The coefficient of determination for this linear model is $R^2 = 0.75$, making the correlation coefficient $r = 0.87$, which means that approximately 75% of the variation in analysis time can be

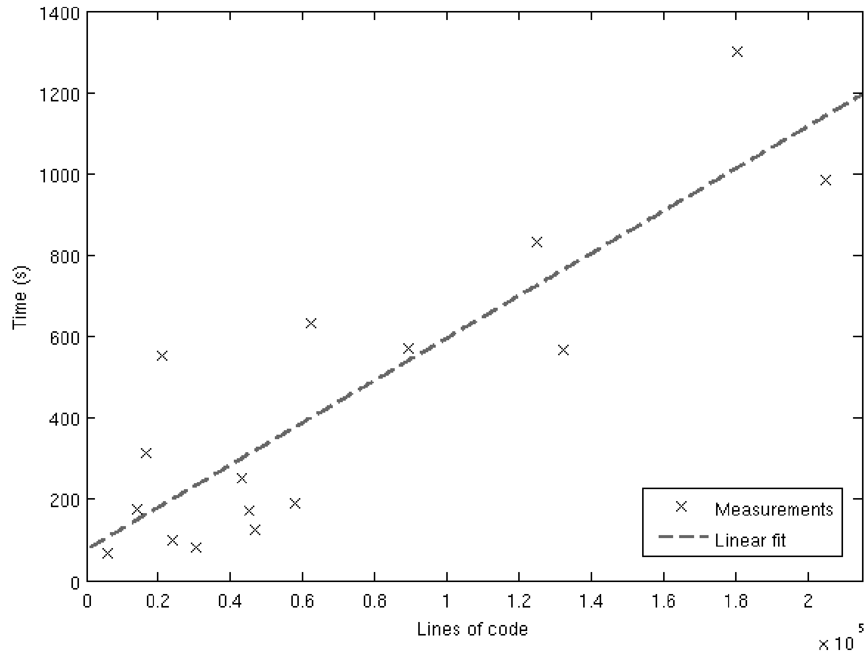


Figure 7: Relation between code size and analysis time in Fortify SCA

explained by the code size. This confirms that there is a strong correlation between code size and analysis time.

Plotting McCabe's cyclomatic complexity number against analysis time (Figure 8) yields a similar result to Figure 7. The linear regression in this case yields a coefficient of determination $R^2 = 0.74$ (correlation coefficient $r = 0.86$), meaning that analysis time is almost equally correlated with code complexity as with code size.

Indeed, code size and complexity are strongly correlated in our macro-benchmark test set, with linear regression on the two yielding a determination coefficient $R^2 = 0.96$. This need not be the case for all applications (e.g. so-called "spaghetti" code is typically highly complex code within few lines of code, while signal processing code is often long code with little branching), but our selection of macro-benchmark applications turns out to have a very consistent ratio between code size and complexity.

In short, code size and complexity do have a strong influence on the analysis time of Fortify SCA, but they do not explain all the variation. Other software metrics may play a role as well, and some of the variation may even be caused by inaccuracy in the metrics produced by PHPdepend.

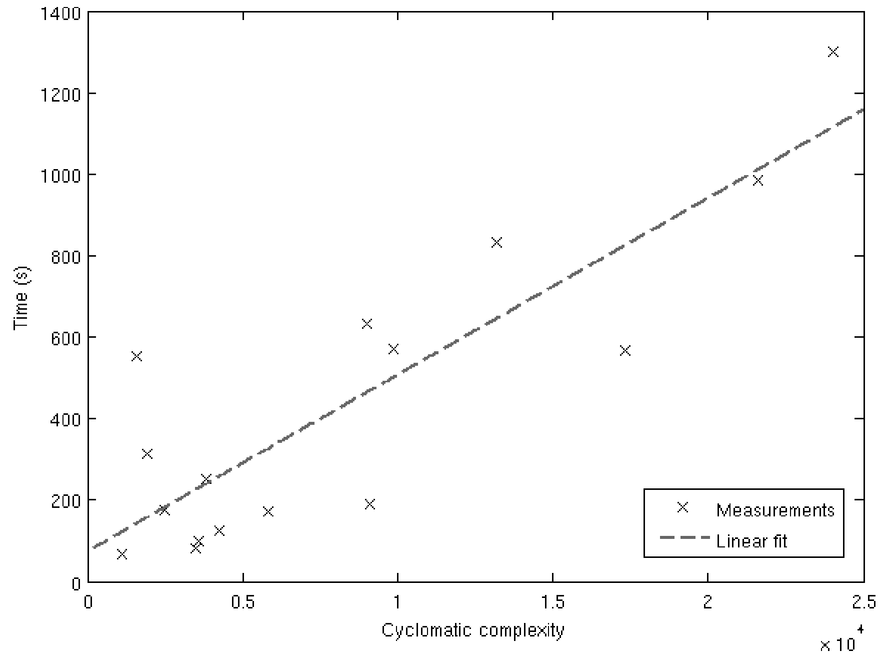


Figure 8: Relation between code complexity and analysis time in Fortify SCA

Usability

Fortify SCA groups the vulnerabilities it finds into one of four categories: 'Critical', 'High', 'Medium' or 'Low'. Judging by Chess' explanation, vulnerabilities are assigned to each category depending on the severity of the vulnerability in general, and the confidence of the analyzer that the vulnerability was correctly detected [1, pg.107]. With the included Audit Workbench, the vulnerabilities in an application can be browsed by their type, or by the source file in which they appear. Analysis results can also be transformed into a formatted report using the included Report Generator tool.

Although Fortify SCA is not capable of automatically fixing detected vulnerabilities, it does give suggestions for each vulnerability on how to fix it manually and how to prevent it from appearing in the future. However, with some of the macro-benchmark tests netting several hundreds of 'High severity' flaws, manually fixing these flaws is not a trivial task. This large number of reports also makes it difficult to spot the truly dangerous vulnerabilities.

Configurability

Fortify SCA requires source, sink and sanitization functions to be specified in the form of *rule packs*, a custom file format tailor-made for use by Fortify 360. Default rule packs for each supported programming language are included with Fortify, and updates can be downloaded from the Fortify customer portal. Rule packs can be modified and expanded by the user to add support for custom source, sink and sanitization functions.

Fortify has a number of quick-configuration settings that assume a certain level of security from the environment in which the source code is run. Code quality can be measured and is quick-configurable as well. Flaws and vulnerabilities are grouped into several severity categories. This can help in setting up security policies and classifying vulnerabilities as acceptable or unacceptable. Fortify is capable of monitoring the regular build process of a program to determine which files it needs to convert and scan, though this is only necessary for programs which are compiled (e.g. C and Java programs). Fortify can be configured and run entirely from the command line, which facilitates scripting and automation.

Openness

Fortify is commercial closed source software and its analysis algorithms are proprietary. Nothing is known about Fortify SCA's inner workings, other than the superficial design described in Chess' book [1]. Fortify is intended to be used as a black box, and only in-depth reviews of the results it produces may reveal something about its inner workings. Although Fortify's command-line tools allow for enough flexibility in the integration with other tools, it is not possible to make modifications to Fortify itself for this purpose.

Development and support

Fortify 360 is actively being developed and regularly receives updates for both software and rule packs. Fortify Software Inc. offers support contracts for their customers, which guarantees their support for an agreed upon period of time.

Initial conclusion

Of all the static analysis tools evaluated here, Fortify SCA provides the most complete and well-rounded package. It does a decent job at identifying vulnerabilities, and the

results it presents are clear and easy to navigate. The user interface is flexible, allowing the tool to be used both through a user-friendly graphical interface, and through a powerful command-line interface. The latter makes Fortify SCA well-suited for scripted integration with other tools.

Nevertheless, Fortify SCA's PHP analysis methods are far from infallible, producing large numbers of false positives and false negatives. A remarkable observation is that Fortify SCA regularly gets beaten in the micro-benchmark tests by a three-year old version of Pixy, and that Fortify's overall behavior is more unpredictable.

We implemented a prototype web-based file manager with a static analysis back-end using Fortify SCA, as illustrated by Figure 6. This prototype confirms the fundamental problems with using static analysis as an automated security check for PHP web applications. In general, the analysis process takes too long to be considered transparent and real-time, and the tool finds problems with and subsequently rejects all but the most trivial of web applications.

9.2 CodeSecure

CodeSecure is a commercial static code security analysis tool developer by Armorize (`http://www.armorize.com`). It is the spiritual successor of WebSSARI, as it was founded by the same authors and started out as a commercialized version of that tool. Unlike WebSSARI, CodeSecure can analyze source code in more languages than just PHP, namely Java, ASP and ASP.NET. Armorize claims that their improved language models have given CodeSecure a false positive rate below 1%.

CodeSecure's primary distribution format is as service-oriented software, also known as Software as a Service (SaaS) [43]. Armorize runs CodeSecure on their own machines, while their customers can let source code analysis be performed remotely. Developers and administrators can access the service through a web interface. Armorize also offers a workbench application and a number of plug-ins that allow CodeSecure to be integrated in the developer's local tool chain. The workbench and plug-ins are still dependent on the remote service from Armorize and need to be connected to it to perform any type of analysis. As an alternative to SaaS, CodeSecure is available in the form of a stand-alone enterprise appliance that allows it to be run locally.

Test setup

We benchmarked a trial version of CodeSecure through its standard web interface. Since the analysis is performed remotely, the specifications of the local computer are irrelevant. Memory usage of the tool is also not a concern, so this does not appear in any test results.

It is unknown which exact versions of the CodeSecure software were used throughout the benchmarks. Some of the test results suggest that the software may have been upgraded during the testing process, though no notification was received of this. The micro-benchmark tests and second macro-benchmark test run, of which the results appear in this section, were performed using CodeSecure version 3.5.1.

Results of the micro-benchmark tests are shown in Table 5 (full results in Appendix A.2). Results of the macro-benchmark tests are shown in Table 6.

Accuracy

CodeSecure achieves the best overall scores in the language support test of any tool evaluated here. Despite this, it achieves a perfect discrimination score in only one of the language support tests, and fails to find any vulnerabilities in both the 'Arrays' and 'Variable variables' tests. CodeSecure recognizes virtually the same types of injection vulnerabilities as Fortify SCA.

CodeSecure does not report the code injection vulnerability in phpBB2's `viewtopic.php` file, nor the file inclusion vulnerability in phpLDAPadmin's `cmd.php` file. The latter omission is remarkable because it is a straightforward file inclusion vulnerability, one that an analyzer of CodeSecure's caliber should be expected to find.

Robustness

CodeSecure is generally robust, in that scans will always run to completion and produce results, even if it does occasionally take a remarkable amount of time. One major exception is when CodeSecure was idle for over a week, apparently hung up on the analysis of another user's source code. This problem was eventually solved by Armorize, after resetting the analyzer and emptying the scan queue. Consequently, the test scans we had queued at that time were never performed.

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	21	30	70%	27	30	90%	48	60	80%	18	30	60%
Aliasing	4	4	100%	3	4	75%	7	8	88%	3	4	75%
Arrays	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Constants	2	2	100%	2	2	100%	4	4	100%	2	2	100%
Functions	3	5	60%	4	5	80%	7	10	70%	2	5	40%
Dynamic inclusion	3	3	100%	2	3	67%	5	6	83%	2	3	67%
Object model	7	8	88%	8	8	100%	15	16	94%	7	8	88%
Strings	2	3	67%	3	3	100%	5	6	83%	2	3	67%
Variable variables	0	3	0%	3	3	100%	3	6	50%	0	3	0%

(a) Language support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	15	18	83%	14	18	78%	29	36	81%	11	18	61%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	2	2	100%	1	2	50%	3	4	75%	1	2	50%
Code injection	2	2	100%	1	2	50%	3	4	75%	1	2	50%
SQL injection	5	6	83%	6	6	100%	11	12	92%	5	6	83%
Server-side include	2	2	100%	1	2	50%	3	4	75%	1	2	50%
XPath injection	1	2	50%	1	2	50%	2	4	50%	0	2	0%
Cross-site scripting	3	3	100%	3	3	100%	6	6	100%	3	3	100%

(b) Injection vulnerability detection

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	7	100%	3	7	43%	10	14	71%	3	7	43%
Regular expressions	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	1	1	100%	1	1	100%	2	2	100%	1	1	100%
Strings	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Cross-site scripting	2	2	100%	2	2	100%	4	4	100%	2	2	100%

(c) Sanitization support

Table 5: CodeSecure micro-benchmark test results (abridged)

Application	Lines of code	Cyclomatic complexity	Run #1 Analysis time (s)	Run #1 # issues found	Run #2 Analysis time (s)	Run #2 # issues found
DokuWiki	57871	9093	299	177	306	398
eXtplorer	30272	3501	52	289	219	309
Joomla	204967	21599	98	25	6027	1953
MediaWiki	180445	24035	624	29	8322	878
Phorum	43288	3814	48092	3173	202	1061
phpAlbum	5755	1105	16	178	20	178
phpBB	20946	1569	1665	39431	542	9847
PHP-Fusion	16693	1917	324	8026	634	7579
phpLDAPadmin	23778	3589	32	20	38	41
phpMyAdmin	124830	13203	6321	96273	6321	96273
phpPgAdmin	45111	5845	953	1874	4871	1984
Php-Stats	14008	2502	46	104	47	145
Serendipity	89483	9840	9419	1229	8460	2931
SquirrelMail	46953	4261	797	3480	520	3628
WordPress	132201	17327	1854	918	3009	952
Zenphoto	62532	9028	1453	6408	799	9401

Table 6: CodeSecure macro-benchmark test results

Responsiveness

The analysis time for each scan is heavily dependent on the amount of activity from other CodeSecure users. Scan operations are queued on the central CodeSecure server and processed one-by-one on a first-come-first-serve basis. This means that the wait time for a scan operation depends on the number of scans already queued, and the complexity of each queued scan. The actual running time of a single scan is comparable in order of magnitude to that of Fortify SCA (Table 4), but CodeSecure lacks consistency.

We have performed two separate test runs on the macro-benchmark, the results of which are shown in Table 6. Both the running time and the number of issues found differ significantly throughout the two runs, even though they were performed on the exact same source code using the exact same analyzer configuration. The only logical explanation for this disparity is that the software was upgraded behind the scenes, in between the two test runs or during the execution of the first test run. If that is the case, no notification was made of this change and no news of an upgrade can be found in retrospect.

Since the analysis time of CodeSecure is clearly dependent on more factors than only the properties of the analyzed code, it would make no sense to plot code size and/or complexity against analysis time here.

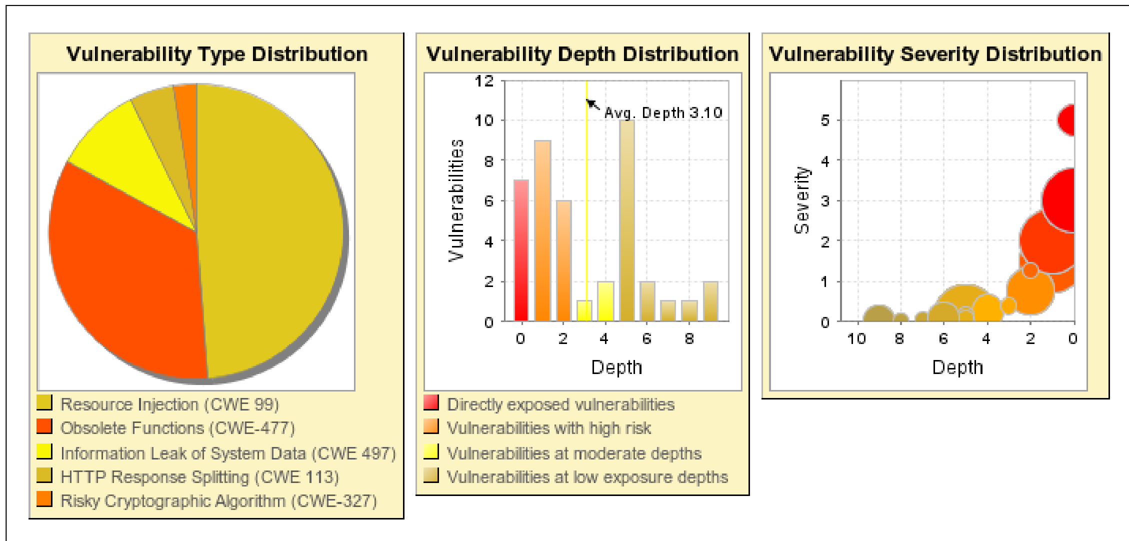


Figure 9: CodeSecure visual reports

Usability

CodeSecure generates detailed reports on the vulnerabilities it finds, including visualizations as in Figure 9. CodeSecure creates pie charts of the distribution of vulnerability types, prioritizes and groups vulnerabilities according to their severity, and visualizes the severity distribution. CodeSecure shows the 'depth' at which vulnerabilities occur, indicating how exposed the vulnerabilities are. CodeSecure can also show how the quality of a project changes over time, as vulnerabilities are fixed and new issues arise.

Although CodeSecure is the spiritual successor of WebSSARI, it does not share WebSSARI's capability to automatically patch vulnerabilities. The reason why this feature was not carried over is unknown. Presumably, automatically patching false positives will introduce new side-effects and bugs in the code, but we could get no confirmation of this theory from Armorize.

The responsiveness of CodeSecure's web-based interface leaves a lot to be desired. Result pages can take several tens of seconds to load, depending on the number of issues that are being reported. When downloading reports in XML or HTML format, it can take several minutes before the download starts, and in one case (an XML report for phpMyAdmin) the download would simply refuse to start, even after waiting for over 30 minutes. This makes inspection of CodeSecure's scan results a frustrating and painfully slow process.

Configurability

CodeSecure has an extensive knowledge base of PHP vulnerabilities and the PHP API functions out of the box. This knowledge base can be expanded with custom sanitization functions and security rules, but this can only be done for each project individually. CodeSecure includes the option to specify 'static includes', which are specific files that are always included with every source file during analysis, bypassing CodeSecure's internal file inclusion mechanism.

Openness

CodeSecure is commercial closed source software and its analysis algorithms are proprietary. According to its creators, CodeSecure's algorithms have advanced far beyond those in WebSSARI [61], though it is unknown how the new algorithms work. No research papers have been published on the technology behind CodeSecure, despite promises from its authors. Questions concerning technical details of CodeSecure (for instance why the ability to automatically patch vulnerabilities is absent) were left unanswered.

Development and support

CodeSecure is under active development and there is continuing technical support from the Armorize staff. Nevertheless, communication with Armorize was difficult and slow at times, and some of our questions have remained unanswered.

As mentioned earlier, CodeSecure's analysis results changed drastically at one point during the evaluation process, and no explanation was given of the cause. We presume the software was upgraded, but in any case a notification would have been appropriate, especially given the drastic changes in functionality.

Initial conclusion

The concept of Software as a Service does not live up to its promise in this case. Instead of being improved by SaaS, the CodeSecure experience is severely hampered by it. The advantage in analysis time over other tools is minimal at best, the behavior of the software is dependent on the activity of other users, and there is no control over when the software is updated. Processing analysis results is an arduous work, and the heavy reliance on Armorize's support combined with poor communication does not help either.

9.3 Pixy

One of the more popular real-world and open solutions is Pixy, a tool written in Java that scans PHP4 programs for XSS and SQL injection vulnerabilities [62]. The initial version of Pixy as described in the first paper by Jovanovic et al. only scanned for XSS vulnerabilities in programs of low complexity [12]. A later article by the same authors described the addition of a model for PHP aliasing [26]. However, according to Biggar, Pixy's alias analysis is both overcomplicated and incomplete [29].

Since the first articles on Pixy were published in 2006, the tool has received several upgrades not described in any article, most notably the inclusion of SQL injection detection. The details of these upgrades are not available on Pixy's website anymore, but given that Pixy is open source the details of these additions could be derived from the implementation. According to the test results of its own authors, Pixy has a false positive rate of around 50%, which they consider to be acceptable [12].

A major shortcoming of Pixy is that it only supports code written for PHP version 4. PHP5's new language features, including its redesigned object model, are not supported at all. Since Pixy's last update in 2007, PHP5 has become widely adopted.

Test setup

Both the micro- and macro-benchmark tests are performed with Pixy version 3.03, dated July 25th 2007. Pixy is executed using the same commands that appear in the included `run-all.pl` script. All tests are performed with the `-A` command-line flag that enables Pixy's alias analysis features.

Results of the micro-benchmark tests are shown in Table 7 (full results in Appendix A.3). Results of the macro-benchmark tests are shown in Table 8.

Accuracy

The most remarkable result from Table 7a is that Pixy passes flawlessly on all the aliasing tests, outperforming both Fortify and CodeSecure. This confirms that the work done by Jovanovic et al. on alias analysis has paid off. Pixy also passes the SQL injection test where sanitized input is used in a query without surrounding quotes (`sql2-bad`, example in Listing 3). Neither Fortify nor CodeSecure pass this specific test.

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	19	30	63%	21	30	70%	40	60	67%	11	30	37%
Aliasing	4	4	100%	4	4	100%	8	8	100%	4	4	100%
Arrays	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Constants	2	2	100%	1	2	50%	3	4	75%	1	2	50%
Functions	5	5	100%	3	5	60%	8	10	80%	3	5	60%
Dynamic inclusion	1	3	33%	1	3	33%	2	6	33%	0	3	0%
Object model	0	8	0%	8	8	100%	8	16	50%	0	8	0%
Strings	3	3	100%	3	3	100%	6	6	100%	3	3	100%
Variable variables	2	3	67%	1	3	33%	3	6	50%	0	3	0%

(a) Language support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	18	39%	17	18	94%	24	36	67%	6	18	33%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Code injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
SQL injection	2	6	33%	6	6	100%	8	12	67%	2	6	33%
Server-side include	2	2	100%	1	2	50%	3	4	75%	1	2	50%
XPath injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Cross-site scripting	3	3	100%	3	3	100%	6	6	100%	3	3	100%

(b) Injection vulnerability detection

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	6	7	86%	3	7	43%	9	14	64%	2	7	29%
Regular expressions	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Strings	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Cross-site scripting	2	2	100%	2	2	100%	4	4	100%	2	2	100%

(c) Sanitization support

Table 7: Pixy micro-benchmark test results (abridged)

Application	Lines of code	Cyclomatic complexity	Analysis time (s)	Peak memory usage (MB)	# scan errors	# issues found
DokuWiki	57871	9093	50	2319	RTE	-
eXtplorer	30272	3501	20	1549	RTE	-
Joomla	204967	21599	5	8	0	0
MediaWiki	180445	24035	5	9	1	-
Phorum	43288	3814	30	1638	RTE	-
phpAlbum	5755	1105	3601	3146	RTE	-
phpBB	20946	1569	216	3126	OOM	-
PHP-Fusion	16693	1917	30	1238	RTE	-
phpLDAPadmin	23778	3589	5	8	0	0
phpMyAdmin	124830	13203	5	9	1	-
phpPgAdmin	45111	5845	5	7	1	-
Php-Stats	14008	2502	76	1485	0	1
Serendipity	89483	9840	802	3075	RTE	-
SquirrelMail	46953	4261	5	8	0	0
WordPress	132201	17327	10	193	RTE	-
Zenphoto	62532	9028	10	178	0	15

Table 8: Pixy macro-benchmark test results[1]

For the remainder of the micro-benchmark tests, Pixy's performance is only average. Pixy recognizes only a small set of vulnerability types (Table 7b) and fails to understand some of the more complex language features of PHP. Its dynamic file inclusion mechanism is not entirely reliable, which has serious consequences when analyzing large applications. On the object model tests, Pixy does not report any vulnerabilities at all, which confirms the fact that Pixy does not support this language features and simply ignores its usage. On the variable variables test, Pixy reports each use as a potential vulnerability, on the condition that PHP's `register_globals` feature is enabled. While this is in fact true, it is an overly cautious assessment and it fails to point out the real vulnerabilities within the code.

Robustness

The first thing we notice from the macro-benchmark test results in Table 8 is that Pixy fails to complete its analysis on a large portion of the tested applications for a variety of reasons. In some cases Pixy ran out of memory, while in other cases it crashed with a generic run-time error or a nondescript error message.

[1] Explanation of abbreviations:
OOM = Out Of Memory
RTE = Run-Time Exception

In addition, in some cases where the analysis does appear to have been completed, the results are highly suspect. For example, Pixy did not find any vulnerabilities in Joomla, but given that the analysis took only a few seconds for over 200,000 lines of code, it is more likely that the analysis simply failed silently. This is not entirely surprising, given that Joomla is written in object-oriented form, which Pixy does not support.

Responsiveness

Given the results in Table 8, not much can be said about Pixy's responsiveness, because it simply failed to finish on a majority of the tests. The most remarkable result is the time it took to analyze phpAlbum; it is the smallest application in the entire macro-benchmark, yet it took by far the longest time to analyze, before Pixy ultimately crashed.

Usability

When executing Pixy, the user needs to specify the source file that contains the entry point of the application. Pixy uses its dynamic file inclusion mechanism to automatically analyze the files that are included by this entry file, either directly or indirectly. There are three problems with this approach:

- The user is required to have knowledge about the source code, or has to be able to find out which file is the entry point.
- Some applications have multiple entry points (e.g. PHP-Fusion can redirect to one of several opening pages, depending on its configuration), which cannot be expressed in a single analysis run.
- If Pixy's dynamic file inclusion mechanism fails to resolve a dynamic inclusion (which does happen in practice), then it will not analyze all the files in the source code and consequently produce false negatives. This appears to have happened in the macro-benchmark with Zenphoto, where the analysis of a large project was successfully completed in merely 10 seconds (Table 8).

Pixy provides a simple text-based report of the vulnerabilities it finds, plus where it found them. It can generate graphs that visualize the propagation of tainted data throughout a program. Pixy does not prioritize the vulnerabilities it finds, nor does it help the user with patching of the vulnerabilities.

Configurability

Pixy uses its own text-based configuration file for specifying sources, sinks and sanitization functions. By default, Pixy is equipped only to find the most common XSS and SQLI vulnerabilities, but its configuration can be expanded to support a larger set of functions and more taint-style vulnerabilities.

Openness

Pixy is open source software written in plain Java. Its source code is included with the default distribution and can therefore be viewed and modified by anyone. Parts of Pixy's underlying analysis algorithms have been documented and published in scientific papers [12, 26], which have been studied and reviewed by other scientists.

Development and support

Pixy has not received any updates since 2007 and is effectively abandoned. Third parties have taken no effort either to fork Pixy's source code and continue its development. Pixy's authors do not offer any official support for the tool, and the user community surrounding the tool is small.

Initial conclusion

Pixy showed promise as a static analysis tool while it was still in development, and was beginning to be taken seriously by the PHP development community. Unfortunately, it was left in an incomplete and immature state several years ago and has become outdated since then. In its current state, Pixy is not robust enough to perform scans on large applications reliably.

9.4 PHP-sat

PHP Static Analysis Tool (PHP-sat) (`http://www.php-sat.org`) was developed by Bouwers as a Google Summer of Code project in 2006 [27]. The tool has been developed with the intention of creating a pragmatic solution for static analysis of PHP programs, building upon the foundation laid out by other research works on the subject. PHP-sat is built upon the Stratego/XT toolset for program transformation and relies on the PHP-front library by the same author.

Test setup

Both the micro- and macro-benchmark tests are performed with PHP-sat version 0.1pre466, built with PHP-front version 0.1pre465 and Stratego/XT version 0.17. PHP-sat is executed using the following command-line flags:

- `--release 5` – Treat source code as PHP5.
- `--simple-inclusion` – Use simple file inclusion.
- `--complex-inclusion` – Use file inclusion with constant propagation.

Results of the micro-benchmark tests are shown in Table 9 (full results in Appendix A.4). Results of the macro-benchmark tests are shown in Table 10.

Accuracy

PHP-sat's understanding of PHP's semantics is simply poor. PHP-sat ignores problems in the majority of the language support tests. It finds vulnerabilities in the aliasing tests, but fails to notice the absence of these vulnerabilities in the corresponding `OK` tests. This suggests that PHP-sat does not perform any aliasing analysis and simply made lucky guesses on the `BAD` tests. Only in the 'Strings' and 'Variable variables' tests does PHP-sat manage to register a score in the discrimination column.

The only type of injection vulnerability that PHP-sat recognizes is the simplest form of cross-site scripting. PHP-sat's documentation admits that the default configuration is currently very small.

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	30	23%	27	30	90%	34	60	57%	4	30	13%
Aliasing	3	4	75%	1	4	25%	4	8	50%	0	4	0%
Arrays	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Constants	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Functions	0	5	0%	5	5	100%	5	10	50%	0	5	0%
Dynamic inclusion	0	3	0%	3	3	100%	3	6	50%	0	3	0%
Object model	0	8	0%	8	8	100%	8	16	50%	0	8	0%
Strings	3	3	100%	3	3	100%	6	6	100%	3	3	100%
Variable variables	1	3	33%	3	3	100%	4	6	67%	1	3	33%

(a) Language support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	3	18	17%	16	18	89%	19	36	53%	1	18	6%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Code injection	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	0	6	0%	6	6	100%	6	12	50%	0	6	0%
Server-side include	0	2	0%	2	2	100%	2	4	50%	0	2	0%
XPath injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Cross-site scripting	1	3	33%	3	3	100%	4	6	67%	1	3	33%

(b) Injection vulnerability detection

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	3	7	43%	6	7	86%	9	14	64%	2	7	29%
Regular expressions	1	2	50%	1	2	50%	2	4	50%	0	2	0%
SQL injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Strings	1	2	50%	2	2	100%	3	4	75%	1	2	50%
Cross-site scripting	1	2	50%	2	2	100%	3	4	75%	1	2	50%

(c) Sanitization support

Table 9: PHP-sat micro-benchmark test results (abridged)

Application	Lines of code	Cyclomatic complexity	Analysis time (s)	Peak memory usage (MB)	# scan errors	# issues found
DokuWiki	57871	9093	5	0	0	7
eXtplorer	30272	3501	5	10	0	2
Joomla	204967	21599	5	19	0	2
MediaWiki	180445	24035	5	0	0	3
Phorum	43288	3814	5	20	0	4
phpAlbum	5755	1105	10	239	0	6
phpBB	20946	1569	10	78	0	4
PHP-Fusion	16693	1917	2036	3789	1	-
phpLDAPadmin	23778	3589	5	0	0	3
phpMyAdmin	124830	13203	5	6	0	2
phpPgAdmin	45111	5845	5	6	0	1
Php-Stats	14008	2502	5	6	0	5
Serendipity	89483	9840	5	6	1	-
SquirrelMail	46953	4261	5	6	0	0
WordPress	132201	17327	5	6	0	1
Zenphoto	62532	9028	5	6	0	6

Table 10: PHP-sat macro-benchmark test results

Robustness

Table 10 shows that PHP-sat finished analysis within ten seconds on all but one of the macro-benchmark tests. It crashed on only one occasion (analysis of Serendipity crashed with a meaningless segmentation fault), and had to be killed prematurely during analysis of PHP-Fusion. The analysis results of the macro-benchmark tests suggest that PHP-sat only analyzed the entry file of each application and ignored all included files, despite being run with its file inclusion features explicitly enabled.

Responsiveness

Since PHP-sat failed to fully analyze each macro-benchmark test application, nothing can be said about PHP-sat's responsiveness.

Usability

PHP-sat does not report which type of vulnerability it finds. It has only one general identification code for malicious code vulnerabilities (MCV), along with the message that "one of the parameters does not meet his [*sic*] precondition". Hence, it also does not prioritize vulnerabilities or help the user with finding the cause of vulnerabilities,

nor does it aid in patching vulnerabilities.

Like Pixy, PHP-sat requires the user to specify the entry file of the application to be analyzed. The other files that PHP-sat analyzes are derived from this entry file by tracing the file inclusions. This has the same disadvantages as with Pixy; PHP-sat fails to find any included files on most (if not all) the macro-benchmark tests and analyzes only the entry files, resulting in the short analysis times seen in Table 10.

Configurability

PHP-sat uses its own text-based configuration file format that allows specification of tainted sources, sensitive sinks and sanitization functions. Each of these items includes the level of taintedness that is applicable. The default configuration is limited only to the most common PHP functions related to cross-site scripting, but can be extended for larger API support and to recognize more types of taint-style vulnerabilities.

Openness

PHP-sat and the libraries it depends on are all open source software. The methods used by PHP-sat have been documented by Bouwers in his paper [27].

Development and support

There has been recent activity in PHP-sat's development, but progress is generally slow and the tool has not received any stable releases yet. There is also no community involved with the development of PHP-sat – so far it has only been a one-man project.

Initial conclusion

PHP-sat works well enough as a research tool, but as a tool for serious practical usage it is not nearly mature enough. Its analysis methods are severely limited, it is not capable of analyzing large applications, and the results it produces a devoid of meaningful information.

9.5 Remaining tools

There are a number of PHP static analysis tools that we are unable to evaluate for a variety of reasons. These tools are discussed in this section.

9.5.1 WebSSARI

By far the most influential article discussed here is the one written by Huang et al. [7], which was the first article to introduce static analysis of PHP code. Its authors had previously worked on several dynamic security analysis methods for web applications, resulting in a tool called Web Application Vulnerability and Error Scanner (WAVES) [6]. Their research into static analysis of PHP led to the development of a tool called Web Security via Static Analysis and Runtime Inspection (WebSSARI), which set the standard for other PHP static analysis tools to follow. Their first approach used a lattice of security-levels to track the taintedness of variables through a program. WebSSARI's techniques were later improved upon by applying bounded model checking instead of the typestate-based algorithm used earlier [8].

In addition to pioneering static analysis for PHP, WebSSARI introduced the idea of automatically patching vulnerabilities found in a program. When a vulnerable section of code is found, WebSSARI inserts an appropriate sanitization routine into the code that will untaint data at run-time. Huang et al. argue that the extra execution time caused by the added routine cannot be considered as overhead, because it is a necessary security check. However, this statement is only true if the detected flaw is indeed a vulnerability and not a false positive.

Although the results published by Huang et al. were very promising, WebSSARI still had many technical shortcomings. WebSSARI was only an early prototype that could not be considered a perfect solution yet [29]. Nguyen-Tuong et al. mention that WebSSARI's algorithm is coarsely grained [9], by which they mean that WebSSARI does not take into account that a variable can have varying degrees of taintedness, depending on its semantic type and the operations performed on it. According to Xie et al. WebSSARI produces a large number of false positives and negatives [11], though it is unknown how large those numbers are. This deficiency is mostly blamed on the fact that WebSSARI uses an intraprocedural algorithm, meaning that it cannot find any problems with data flow that crosses function boundaries. Additionally, Bouwers remarks that WebSSARI's authors do not mention how aliases or arrays are handled [27], and that WebSSARI does not automatically include files and fails to parse a substantial number of files.

Since the publishing of their two articles, Huang et al. have gone on to establish Armorize Technologies in 2006 and have commercialized both WAVES and WebSSARI into HackAlert and CodeSecure, respectively. Even though in their own words they have far advanced their algorithms [61], they have not seen fit to publish any more research articles. Additionally, both WAVES and WebSSARI have been made unavailable and their authors have not shown any intent to make them available again for other researchers to study [11, 12]. This unfortunately makes it impossible at this point to perform any tests and to produce meaningful information on WebSSARI's actual performance.

9.5.2 PHPrevent

PHPrevent is a static code security tool described by a paper by Nguyen-Tuong et al. [9]. It tracks the taintedness of variables at a finer granularity than its then-rival WebSSARI, resulting in a lower number of false positives. For example, if a string value in a program is constructed partially from tainted user data and partially from hard-coded data, then only part of that string value should be considered tainted. PHPrevent keeps precise track of which part of that value remains tainted as it flows throughout the program, and so it can make an exact judgment about the danger involved when that value reaches a sensitive sink.

PHPrevent is implemented as a modification to the PHP interpreter that hardens application source code while it is being interpreted. The development of PHPrevent has failed to keep up with changes to the official PHP interpreter and the tool has effectively been abandoned. It is not available from the authors anymore.

9.5.3 phc

phc (`http://www.phpcompiler.org`) is an ahead-of-time PHP compiler and optimizer, allowing PHP programs to be compiled into native applications and to be optimized for speed [28]. phc works by parsing PHP scripts into an abstract syntax tree, generating C code from that and finally compiling that into a stand-alone application or shared library that dynamically links to PHP's C API. A stand-alone compiled PHP program then works as it would if it were run using PHP's command line interpreter, while a PHP program compiled as a shared library can be used as a plug-in module for a web server application such as Apache. In this way, phc can transform a regular web application into an optimized web server extension. Because phc allows compiled programs to dynamically link to PHP's C API, it can offer support for the entire PHP standard

library. In order to accomplish this, a special version of PHP that includes support for the *embed SAPI* needs to be installed.

Recently, Biggar and Gregg have released a draft paper describing the development of a static analysis model for PHP that has been implemented using phc's optimizing compiler as the starting point [29]. Their paper acknowledges the many problems that are involved with modeling PHP's semantics and deals with most of them in the most accurate and complete manner that has yet been described. Most notably, they are the first to describe a complete model of PHP aliasing mechanics. At the same time, they admit that several language features have not been modeled yet [29, Sec.4.9]. Amongst the omitted features are dynamic code evaluation using the `eval` statement, error handling and exceptions.

The new static analysis model has not yet officially been merged into phc yet, but it is already available as an experimental branch of the source code. All it does at the moment is provide a model for PHP's language semantics; it does not perform any security analysis yet. However, once the implementation of the language model is complete, it will provide a solid basis for the development of a static code security analysis tool.

9.5.4 RATS

Rough Auditing Tool for Security, or RATS, is an open source tool designed to scan C, C++, Perl, PHP and Python source code for security related programming errors [63]. There is no documentation available about its methods or its implementation. Additionally, RATS has been acquired by Fortify Software, Inc. and is no longer under active development.

As also mentioned by Bouwers [27] and Chess and West [1], RATS is a generic tool that only performs lexical analysis. It accepts syntactically incorrect PHP code without warnings or errors, and is unable to perform any semantic analysis. It does not automatically perform source file inclusions, but skips `include` statements altogether. RATS only looks at the presence of known sensitive sink functions within a single source file and warns about the possible danger of using that function, through a generic warning message. For example, usage of the PHP function `mail` will always generate the following warning:

```
Arguments 1, 2, 4 and 5 of this function may be passed to an external
program. (Usually sendmail). Under Windows, they will be passed to a
remote email server. If these values are derived from user input,
make sure they are properly formatted and contain no unexpected
characters or extra data.
```

RATS reads a list of sensitive sink functions from a simple XML database file, containing both the names and some basic information on these functions. Although RATS technically performs syntax analysis, the way it is used makes the tool not much more useful than using the UNIX command line tool `grep` to search for occurrences of the names of sensitive sink functions.

9.5.5 PHP String Analyzer

Minamide [16] has written a tool called PHP String Analyzer [70] that models the manipulation of string values within a PHP program, and then approximates the string output of that program. PHP String Analyzer was developed as an academic tool that works as a proof-of-concept on small test programs. Its algorithms could potentially allow for detection of cross-site scripting vulnerabilities, but currently the tool only produces a list of possible permutations of a program's output. It is not suited for security analysis of large web applications.

10 Discussion

10.1 Commercial vs. open source

We have evaluated four static analysis tools in total, two of which are commercial (Fortify and CodeSecure) and two of which are open source (Pixy and PHP-sat). Both software philosophies have their pros and cons, and both types of tool have left different impressions. We discuss them here.

We observed that it is not uncommon for non-commercial static analysis tools to be abandoned after a few years of development. Lack of maintenance has caused Pixy, PHPrevent and WebSSARI (amongst others) to all become stale and outdated, despite their promising starts. This trend is especially visible with tools born from academic research; researchers tend to present their ideas, create a proof-of-concept implementation, and then move on to a different subject, abandoning their earlier work. This is of course common practice for researchers in any field, but the problem with this approach is that it leads to a large number of fragmented applications, none of which is mature enough to be used in practice.

Both commercial solutions that we evaluated have a closed implementation and use proprietary techniques. The downside of this is that we cannot examine the analysis methods of these tools, or to verify their correctness. Although CodeSecure's authors have stated their intent to publish articles about new techniques that have been applied in CodeSecure [61], they have not yet seen fit to do so. It is impossible to determine why these tools produce false positives and false negatives, and what improvements could be made or expected in the future.

Another obvious downside of closed source tools is they do not allow us to make our own modifications. Any changes that we might need would have to be submitted to and approved by the software vendor. Using a closed source commercial tool also means that we would be reliant on a single supplier for updates and support. This is not necessarily bad, but it adds an undesirable dependency to our own business and restricts freedom of movement.

One advantage of commercial solutions is that they do generally offer good guarantees for long-term support. Free and open source software is more reliant upon its community for long-term development and support, and the PHP static analysis community is clearly not strong enough for this kind of commitment.

The tools we have tested represent two extremes in the software world: closed source and proprietary commercial tools on one side, and open source research tools on the other side. The ideal solution would lie somewhere in-between these two extremes. There are companies with a business model that combines free open source software with optional commercial licensing and technical support, for example MySQL and Linux distributions such as Red Hat and Ubuntu. For PHP static analysis however, this combination currently does not exist yet.

While there are plenty of different ideas, approaches, solutions and strategies in existence, it takes a serious commitment to transform all that material into a single, unified and relevant security analysis tool and to keep it up-to-date with new developments. We ourselves have deliberately decided not to start the development of such a tool, because we are unable to make this commitment. An organization such as OWASP, which specializes in web application security and the tools necessary to enforce it, would probably be the best candidate for this role.

10.2 Software as a Service

At first glance, Software as a Service (SaaS) seems like an excellent way to outsource computationally expensive tasks. There is no need to purchase expensive heavy-duty computers, nor do they need to be maintained by highly trained staff. It could potentially save a lot of time and money. However, having experienced SaaS first-hand through Armorize's CodeSecure, we have found a number of problems and concerns with this approach.

The reliance on a single supplier is already an issue for commercial tools, but commercial SaaS tools make this reliance even greater. While working with SaaS tools, we got an overwhelming sense of lack of control. The supplier decides how much processing power is allocated to each task, the supplier decides when each task is executed, and if something goes wrong, then the supplier is the only one who can fix it. Scanning tasks are placed in a queue and executed in first-come-first-serve order. If many users of the service are trying to analyze their source code at the same time, this can lead to traffic jams. All of this should not be a problem if the supplier provides plenty of processing power and maintenance support, but our experiences with CodeSecure show that this is not self-evident.

At one point during our evaluation of CodeSecure, the analyzer abruptly stopped processing tasks. Presumably, it got stuck while analyzing source code from another user.

In the meantime, the task queue grew longer and our own scanning tasks were left to wait indefinitely. It was not until more than a week passed that Armorize took notice, reset the analyzer and emptied the task queue. In such a situation, there is nothing we can do ourselves, except asking the supplier for help, waiting patiently and hoping the problem will be fixed eventually.

Similarly, the responsiveness of CodeSecure's web interface leaves a lot to be desired. Some of our macro-benchmark tests produced a large number of warnings, which resulted in a proportionally large report pages. Consequently, some of these pages could take minutes to load. It got even worse when trying to download analysis reports in XML format. It appears these are generated on-the-fly when the user tries to download them, but for large reports this could again take minutes, during which the web browser can not show if there is any progress. In one particular case (a report for phpMyAdmin with 96273 warnings), the XML file would simply refuse to download, even after multiple attempts and waiting up to half an hour on each attempt.

SaaS solutions also offer the user no control over when software upgrades are deployed. Our evaluation of CodeSecure shows that an upgrade of the underlying software can have a major impact on the functioning of a tool, yet there is no way to study the effects of an upgrade beforehand, to properly prepare for the upgrade, or to postpone the upgrade if necessary. If the supplier decides to deploy an upgrade, then the user simply has to abide by that decision.

On top of this comes the fear that the supplier might go out of business or simply drop their support one day. A normal off-line tool would remain usable if this were to happen, but a SaaS tool would instantly become completely inaccessible and unusable.

The use of SaaS also raises questions concerning the confidentiality of the source code. Using a SaaS solution involves transmitting the source code that is to be analyzed to the supplier and letting them perform the analysis remotely. In our situation, customers of the web host will upload their source code to the web host, and the web host passes it through to the SaaS supplier. However, customers will normally expect that their code is kept confidential and is not shared with any third parties. Even though Armorize assures that CodeSecure automatically deletes source code after completing its analysis, if there is a confidentiality agreement between customer and web host, then using a SaaS tool to scan source code technically breaks that agreement.

SaaS is only useful in our situation if the software can be accessed through a machine-friendly communication protocol. CodeSecure is currently only accessible through a

web interface and a workbench tool, but those options are only useful when there is a person directly interacting with the software. An automated system would require a programmatic interface designed for interaction between computers, for example a web service interface using the WSDL and SOAP protocols [72, 73]. Armorize says that support for programmatic interfaces is currently on the roadmap for CodeSecure, but with only a rough estimate for a 'late 2010' release.

10.3 Future work

Future research on this topic could focus on one of three distinct areas: keeping up-to-date with developments in static analysis for PHP, further development of the benchmark for PHP static analysis tools, or research into the feasibility of dynamic analysis tools as a practical solution for securing PHP web applications.

10.3.1 Static analysis for PHP

Of all the open source static analysis tools for PHP we have reviewed, phc currently shows the most promising developments. Although its implementation is not complete at this time, phc promises to deliver one of the most accurate language models of PHP in existence and certainly the most accurate of all open solutions. Once its language model is completed, it should provide an excellent basis for a taint-style vulnerability detection tool. Additionally, phc is made to be used in real production environments, so both the quality of the application and its long-term support are looking solid. All this makes phc a tool to keep a close eye on in the future.

At the time of writing, Fortify Software Inc. has just released a new application security suite called Fortify on Demand [64], which is a Software-as-a-Service (SaaS) solution much like Armorize's CodeSecure. Unfortunately, this new development came too late for an in-depth evaluation in this thesis. Presumably, Fortify on Demand uses the same analysis algorithms as Fortify 360, but with a web-based interface similar to that of CodeSecure. Several advantages and disadvantages of SaaS have been discussed in this thesis, but these are mostly based on the experiences with CodeSecure. It may be interesting to see how Fortify's SaaS solution compares to CodeSecure.

10.3.2 Benchmark development

The benchmark we have developed for our evaluation of static analysis tools is still fairly concise and has room for improvement. The goal would be to gain more detailed information about a tool's performance in every kind of circumstance.

The micro-benchmark tests currently focus largely on 'difficult' language features of PHP and eccentric ways in which data can be manipulated. It would be good to extend the micro-benchmark with test cases that focus on all the basic language features as well. The current test set also represents only a small selection of the most common types of injection vulnerabilities. These tests can be extended with cases of other vulnerability types (both injection-based and otherwise), cases of poor code quality, cases of improper error handling, and cases of wrongly used security features. The micro-benchmark tests can be extended to cover a larger portion of the PHP standard API, for example cases of SQL injection vulnerabilities using the Oracle, Microsoft SQL and ODBC APIs.

The applications comprising the macro-benchmark can be examined more thoroughly, to get a better view of where a tool succeeds or fails, and possibly why. Some applications use custom sanitization routines, for example DokuWiki's extensive use of `preg_replace`. It might be useful to make an inventory of these, as these are likely causes of false positives in an analysis report. Some applications are known to have actual vulnerabilities (e.g. phpBB2 and phpLDAPadmin), while some may reveal vulnerabilities on closer inspection. This knowledge can be used to pinpoint specific true positives and/or false negatives in analysis reports, and these known vulnerability patterns can be used as a basis for test cases in the micro-benchmark.

Currently, processing of the benchmark results is partially manual labor. To make the benchmark more accessible and portable, it should be rewritten as a fully automated process, similar to ABM by Newsham and Chess [35]. Possibly, it could be made compatible with ABM, so that it conforms to a certain standard.

Furthermore, for the benchmark to reach wider acceptance, it should be decoupled from this research project and published as a stand-alone project. It will then have the chance to be reviewed and augmented by other researchers, gaining relevance in the process.

10.3.3 Dynamic analysis

Regarding the original problem of securing web applications on a public web server, the solution might have to be sought in dynamic analysis. Static analysis has turned out to be too complicated for real-time use, and its undecidable nature makes it too unreliable. Dynamic analysis is a more down-to-earth solution that will work in real-time, albeit with some performance overhead. The main difference is that instead of preventing vulnerable web applications from being deployed altogether, dynamic analysis only prevents harmful data from entering a potentially vulnerable application.

The concept of combining dynamic analysis with a lightweight static analysis technique (as mentioned in Section 3.3) is worth further investigation too. The addition of a static analysis step can improve the precision of dynamic analysis, while static analysis in this case does not have to be as thorough as the tools we have evaluated in this thesis. This combination could theoretically provide a secure and reliable web application platform, while both deployment and execution remain fast enough to be considered real-time.

11 Conclusion

Software needs to be secure in order to allow parties of different trust levels to interact with each other, without risking that untrusted parties exploit critical parts of the software. Web applications are mostly susceptible to input validation vulnerabilities, also known as taint-style vulnerabilities, the most common of which are cross-site scripting and SQL injection. Because web applications are made to be easily accessible through the internet, they are particularly exposed to attacks.

Static analysis allows programmers to look for security problems in their source code, without the need to execute it. Static analysis tools will always produce false positives and false negatives, because they need to make trade-offs between accuracy and speed, and because program analysis is inherently undecidable. The PHP programming language makes static analysis especially complicated, because of PHP's highly dynamic nature and its vaguely defined semantics.

We have created a benchmark for the evaluation of static analysis tools that target PHP web application security. This benchmark consists of both a micro-benchmark with synthetic tests and a macro-benchmark with real-world web applications. Using this benchmark, we have evaluated the general performance of the current offering of tools, as well as their fitness for automated security control on web servers.

When it comes open static analysis solutions for PHP, they are either abandoned (Pixy), immature (PHP-sat) or incomplete (phc). Many tools are made for academic purposes and were simply never meant to be used in a serious production environment. Each of these applications implements a small number of self-contained ideas, with none of them ever reaching a mature state or attaining any relevance in the practical world. Pixy is the only open tool that has ever come close to being a serious contender, but by now it is gravely outdated.

The commercial solutions (Fortify and CodeSecure) are currently the only ones mature enough that they can be considered for serious usage. It is safe to assume that both of them use many of the techniques described in the papers referenced by this thesis. However, their closed implementations make it impossible to verify this, nor is it possible to make a solid assessment of the correctness of their methods. Additionally, the reliance on a single supplier for development and support makes them a difficult choice.

One major problem with static analysis in a web hosting environment is that it is a computationally intensive operation. A substantial amount of system resources would

have to be dedicated to running security tests, which normally would have been used for the serving of web pages. Even with those resources, analyzing an average web application takes a considerable amount of time and would cause a noticeable delay in the deployment of an application. Running static analysis remotely through a Software-as-a-Service solution does not solve this problem either.

A more general problem with static analysis tools is that they are not completely reliable. All the tools we tested produced significant numbers of false positives and false negatives on both synthetic and real-world tests. This partially stems from the fundamental problem of undecidability that all static analysis suffers from.

Even if the perfect static analysis tool were to exist, efficiently patching detected vulnerabilities remains another unsolved problem. Not a single solution we evaluated here had a satisfactory solution for this problem. Most tools return an analysis report in one form or another, but it is up to the user to process these reports manually and to fix the issues that are found. WebSSARI is the only tool that promised to fix taint-style vulnerabilities automatically, but the effectiveness of this feature can no longer be tested.

In conclusion, static analysis tools are great for helping programmers understand the source code they are working on, and to find *potential* problems. Static analysis is not flawless though, and should not be solely relied upon to decide whether an application is secure or not. Using static analysis to automatically reject unsecure applications on a web server is therefore not feasible.

References

[1] Brian Chess, Jacob West (2007). "Secure Programming with Static Analysis". Addison-Wesley. ISBN 978-0321424778.

[2] Katrina Tsipenyuk, Brian Chess, Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors." In *Proceedings of the NIST Workshop on Software Security Assurance Tools, Techniques and Metrics (SSATTM)*, pages 36–43, Long Beach, California, 2005.

[3] Yichen Xie, Alex Aiken. "Saturn: A SAT-based Tool for Bug Detection". In *Proceedings of CAV 2005*, Edinburgh, Scotland, UK.

[4] Brian V. Chess. "Improving Computer Security using Extended Static Checking". *IEEE Symposium on Security and Privacy*, 2002.

[5] David Evans, David Larochelle. "Improving Security Using Extensible Lightweight Static Analysis". *IEEE Software*, Jan/Feb 2002.

[6] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, Chung-Hung Tsai. "Web Application Security Assessment by Fault Injection and Behavior Monitoring". In *Proceedings of the Twelfth International Conference on World Wide Web (WWW2003)*, pages 148-159, May 21-25, Budapest, Hungary, 2003.

[7] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D.T. Lee, Sy-Yen Kuo. "Securing Web Application Code by Static Analysis and Runtime Protection". In *Proceedings of the Thirteenth International World Wide Web Conference (WWW2004)*, pages 40-52, New York, May 17-22, 2004.

[8] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D.T. Lee, Sy-Yen Kuo. "Verifying Web Applications Using Bounded Model Checking". In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN2004)*, pages 199-208, Florence, Italy, Jun 28-Jul 1, 2004.

[9] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, David Evans. "Automatically Hardening Web Applications Using Precise Tainting". In *IFIP Security 2005*, Chiba, Japan, May 2005.

[10] Zhendong Su, Gary Wassermann. "The Essence of Command Injection Attacks in Web Applications". In *Proceedings of POPL'06*, Charleston, South Carolina, January 11-13, 2006.

[11] Yichen Xie, Alex Aiken. "Static Detection of Security Vulnerabilities in Scripting Languages". In *Proceedings of the 15th USENIX Security Symposium*, pages 179–192, July 2006.

[12] Nenad Jovanovic, Christopher Kruegel, Engin Kirda. "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)". In *2006 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.

[13] Stephen Thomas, Laurie Williams, Tao Xie. "On automated prepared statement generation to remove SQL injection vulnerabilities". In *Information and Software Technology*, Volume 51, Issue 3 (March 2009), pages 589-598.

[14] William G.J. Halfond, Alessandro Orso. "Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks". In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 22-28, St. Louis, MO, USA, May 2005.

[15] Michael Martin, Benjamin Livshits, 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*, Volume 40, Issue 10, pages 365-383, October 2005.

[16] Yasuhiko Minamide. "Static Approximation of Dynamically Generated Web Pages". In *Proceedings of the 14th International World Wide Web Conference*, pp. 432-441, 2005.

[17] Gary Wassermann, Zhendong Su. "Sound and Precise Analysis of Web Applications for Injection Vulnerabilities". In *Proceedings of PLDI 2007*, San Diego, CA, June 10-13, 2007.

[18] Gary Wassermann and Zhendong Su. "Static Detection of Cross-Site Scripting Vulnerabilities". In *Proceedings of ICSE 2008*, Leipzig, Germany, May 10-18, 2008.

[19] Karthick Jayaraman, Grzegorz Lewandowski, Steve J. Chapin. "Memento: A Framework for Hardening Web Applications". In *Center for Systems Assurance Technical Report* CSA-TR-2008-11-01.

[20] Peter Wurzinger, Christian Platzer, Christian Ludl, Engin Kirda, Christopher Kruegel. "SWAP: Mitigating XSS Attacks using a Reverse Proxy". *The 5th International Workshop on Software Engineering for Secure Systems (SESS'09)*, 31st International Conference on Software Engineering (ICSE), IEEE Computer Society, Vancouver, Canada, May 2009.

[21] William G.J. Halfond, Alessandro Orso. "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation". In *IEEE Transactions on Software Engineering (TSE)*, Volume 34, Issue 1, pages 65–81, 2008.

[22] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis". In *Computers and Security Journal*, Elsevier, Vol: 28, No: 7.

[23] 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 *IEEE Security and Privacy*, Oakland, May 2008.

[24] Adrienne Felt, Pieter Hooimeijer, David Evans, Westley Weimer. "Talking to Strangers Without Taking Their Candy: Isolating Proxied Content". *Workshop on Social Network Systems*, 2008.

[25] Benjamin Livshits, Aditya V. Nori, Sriram K. Rajamani, Anindya Banerjee. "Merlin: Specification Inference for Explicit Information Flow Problems". *Conference on Programming Language Design and Implementation (PLDI) 2009*, June 2009.

[26] Nenad Jovanovic, Christopher Kruegel, Engin Kirda. "Precise Alias Analysis for Static Detection of Web Application Vulnerabilities". In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, Ottawa, Canada, June 2006.

[27] Eric Bouwers. "Analyzing PHP, An Introduction to PHP-Sat". *Technical report*, 2006.

[28] Paul Biggar, Edsko de Vries, David Gregg. "A Practical Solution for Scripting Language Compilers". In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1916–1923, Honolulu Hawaii, 2009.

[29] Paul Biggar, David Gregg. "Static analysis of dynamic scripting languages". *Draft*: Monday 17th August, 2009 at 10:29.

[30] V. Benjamin Livshits, Monica S. Lam. “Finding Security Vulnerabilities in Java Applications with Static Analysis”. *Technical report*, 2005.

[31] Sarah Heckman, Laurie Williams. “On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques”. In *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Kaiserslautern, Germany, October 9-10, 2008, pp. 41-50.

[32] Benjamin Livshits. “Improving Software Security With Precise Static And Runtime Analysis”. *Doctoral dissertation*, Stanford University, Stanford, California, December, 2006.

[33] Susan Elliott Sim, Steve Easterbrook, Richard C. Holt. “Using Benchmarking to Advance Research: A Challenge to Software Engineering”. In *Proceedings of the Twenty-fifth International Conference on Software Engineering*, Portland, Oregon, pp. 74-83, 3-10 May, 2003.

[34] Justin E. Harlow III. “Overview of Popular Benchmark Sets”. In *IEEE Design & Test of Computers*, Volume 17, Issue 3, pages 15-17, July-September 2000.

[35] Tim Newsham, Brian Chess. “ABM: A Prototype for Benchmarking Source Code Analyzers”. In *Proceedings of the Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM 05)*, Long Beach, California, 2005.

[36] James Walden, Adam Messer, Alex Kuhl. “Measing the Effect of Code Complexity on Static Analysis Results”. *International Symposium on Engineering Secure Software and Systems (ESSoS)*, Leuven, Belgium, February 4-6, 2009.

[37] Misha Zitser, Richard Lippmann, Tim Leek. “Testing Static Analysis Tools Using Exploitable Buffer Overflows From Open Source Code”. In *SIGSOFT Software Engineering Notes*, Volume 29, Issue 6, pages 97–106, November 2004.

[38] Thomas J. McCabe. “A Complexity Measure”. In *IEEE Transactions on Software Engineering (TSE)*, Volume 2, Issue 4, pages 308-320, December 1976.

[39] Derek M. Jones. “Forms of language specification”. *ISO/IEC Working Group on Programming Language Vulnerabilities*, 2008.

[40] Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., and Stata, R. “Extended Static Checking for Java.” In *Proc. 2002 ACM SIGPLAN Conference Programming Language Design and Implementation (PLDI2002)*, pages 234-245, volume 37(5) of ACM SIGPLAN Notices, Berlin, Germany, Jun 2002.

[41] Nejmeh, B.A. “NPATH: a measure of execution path complexity and its applications.” In *Communications of the ACM*, Volume 31, Issue 2, Pages: 188-200, February 1988.

[42] David Scott, Richard Sharp. “Abstracting application-level web security”. In *Proceedings of the 11th international conference on World Wide Web (WWW2002)*, pages 396-407, Honolulu, Hawaii, May 17-22, 2002.

[43] Mark Turner, David Budgen, Pearl Brereton. “Turning Software into a Service”. In *IEEE Computer*, vol. 36, no. 10, pp. 38-44, October, 2003.

[44] San Murugesan, Y. Deshpande. “Meeting the challenges of Web application development: the web engineering approach”. In *Proceedings of the 24rd International Conference on Software Engineering*, ICSE 2002.

[45] Alan Turing. "On computable numbers, with an application to the Entscheidungsproblem". In *Proceedings of the London Mathematical Society*, Series 2, Volume 42, pages 230–265, 1936.

[46] Rice, H. G. "Classes of Recursively Enumerable Sets and Their Decision Problems". In *Transactions of the American Mathematical Society*, 74, pages 358–366, 1953.

[47] Crew, R.F., "ASTLOG: A Language for Examining Abstract Syntax Trees". In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[48] Kris De Volder. "JQuery: A Generic Code Browser with a Declarative Configuration Language". In *Proceedings of PADL*, 2006.

[49] S. Goldsmith, R. O'Callahan, and A. Aiken. "Relational Queries Over Program Traces". In *Proceedings of the 2005 Conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2005.

[50] The WhiteHat Website Security Statistics Report - 8th Edition - Fall 2009. `http://www.whitehatsec.com/home/resource/stats.html` *(Retrieved on 2010-03-29)*

[51] Santy Worm Spreads Through phpBB Forums. `http://news.netcraft.com/archives/2004/12/21/santy_worm_spreads_through_phpbb_forums.html` *(Retrieved on 2010-03-29)*

[52] phpldapadmin Local File Inclusion. `http://www.exploit-db.com/exploits/10410` *(Retrieved on 2010-03-29)*

[53] Perl Programming Documentation - Taint Mode. `http://perldoc.perl.org/perlsec.html#Taint-mode` *(Retrieved on 2010-04-01)*

[54] Taint support for PHP, June 2008. `http://wiki.php.net/rfc/taint` *(Retrieved on 2010-04-16)*

[55] Minutes PHP Developers Meeting, Paris, November 11th and 12th, 2005. `http://www.php.net/~derick/meeting-notes.html#sand-boxing-or-taint-mode` *(Retrieved on 2010-04-16)*

[56] Zend Weekly Summaries Issue #368, November 2007. `http://devzone.zend.com/article/2798-Zend-Weekly-Summaries-Issue-368` *(Retrieved on 2010-04-16)*

[57] PHP: History of PHP. `http://www.php.net/manual/en/history.php.php` *(Retrieved on 2010-04-16)*

[58] Nexen.net: PHP statistics for October 2008. `http://www.nexen.net/chiffres_cles/phpversion/18824-php_statistics_for_october_2008.php` *(Retrieved on 2010-04-16)*

[59] PHP 10.0 Blog: What is PHP, anyway? `http://php100.wordpress.com/2006/10/24/what-is-php-anyway` *(Retrieved on 2010-04-16)*

[60] Security Report: Static Analysis Tools. `http://www.securityinnovation.com/security-report/november/staticAnalysis1.htm` *(Retrieved on 2010-03-29)*

[61] OpenWAVES.net: Home for WAVES and WebSSARI. `http://www.openwaves.net` *(Retrieved on 2010-04-16)*

[62] Pixy: XSS and SQLI Scanner for PHP Programs. `http://pixybox.seclab.tuwien.ac.at/pixy` *(Retrieved on 2010-03-31)*

[63] RATS: Rough Auditing Tool for Security.
http://www.fortify.com/security-resources/rats.jsp *(Retrieved on 2010-04-16)*

[64] Fortify on Demand. http://www.fortify.com/products/ondemand *(Retrieved on 2010-04-16)*

[65] PHPrevent: PHP Security Project. http://www.cs.virginia.edu/nguyen/phprevent *(Retrieved on 2010-04-16)*

[66] OWASP: Top Ten Project. http://www.owasp.org/index.php/OWASP_Top_Ten *(Retrieved on 2010-04-16)*

[67] OWASP: Injection Attacks. http://www.owasp.org/index.php/Category:Injection *(Retrieved on 2010-04-16)*

[68] OWASP WebGoat Project. http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project *(Retrieved on 2010-03-31)*

[69] OWASP WebScarab NG Project. http://www.owasp.org/index.php/OWASP_WebScarab_NG_Project *(Retrieved on 2010-04-07)*

[70] PHP String Analyzer. http://www.score.is.tsukuba.ac.jp/~minamide/phpsa *(Retrieved on 2010-04-16)*

[71] phc Documentation: The Abstract Grammar.
http://www.phpcompiler.org/doc/latest/grammar.html *(Retrieved on 2010-04-16)*

[72] Web Service Definition Language (WSDL). http://www.w3.org/TR/wsdl *(Retrieved on 2010-04-16)*

[73] Simple Object Access Protocol (SOAP). http://www.w3.org/TR/soap12 *(Retrieved on 2010-04-16)*

[74] SAMATE - Software Assurance Metrics And Tool Evaluation. http://samate.nist.gov *(Retrieved on 2010-03-31)*

[75] Stanford SecuriBench. http://suif.stanford.edu/~livshits/securibench *(Retrieved on 2010-03-31)*

[76] Stanford SecuriBench Micro. http://suif.stanford.edu/~livshits/work/securibench-micro *(Retrieved on 2010-03-31)*

Appendices

A Micro test results

A.1 Fortify 360

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	13	18	72%	16	18	89%	29	36	81%	11	18	61%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	2	2	100%	2	2	100%	4	4	100%	2	2	100%
Code injection	2	2	100%	1	2	50%	3	4	75%	1	2	50%
SQL injection	4	6	67%	6	6	100%	10	12	83%	4	6	67%
Server-side include	2	2	100%	1	2	50%	3	4	75%	1	2	50%
XPath injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Cross-site scripting	3	3	100%	3	3	100%	6	6	100%	3	3	100%
arg1	no			yes			1	2	50%	no		
cmd1	yes			yes			2	2	100%	yes		
cmd2	yes			yes			2	2	100%	yes		
code1	yes			no			1	2	50%	no		
code2	yes			yes			2	2	100%	yes		
sql1	yes			yes			2	2	100%	yes		
sql2	no			yes			1	2	50%	no		
sql3	yes			yes			2	2	100%	yes		
sql4	yes			yes			2	2	100%	yes		
sql5	yes			yes			2	2	100%	yes		
sql6	no			yes			1	2	50%	no		
ssi1	yes			no			1	2	50%	no		
ssi2	yes			yes			2	2	100%	yes		
xpath1	no			yes			1	2	50%	no		
xpath2	no			yes			1	2	50%	no		
xss1	yes			yes			2	2	100%	yes		
xss2	yes			yes			2	2	100%	yes		
xss3	yes			yes			2	2	100%	yes		

Table 11: Fortify injection vulnerability detection

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	7	100%	3	7	43%	10	14	71%	3	7	43%
Regular expressions	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	1	1	100%	1	1	100%	2	2	100%	1	1	100%
Strings	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Cross-site scripting	2	2	100%	2	2	100%	4	4	100%	2	2	100%
preg1	yes			no			1	2	50%	no		
preg2	yes			no			1	2	50%	no		
sql1	yes			yes			2	2	100%	yes		
string1	yes			no			1	2	50%	no		
string2	yes			no			1	2	50%	no		
xss1	yes			yes			2	2	100%	yes		
xss2	yes			yes			2	2	100%	yes		

Table 12: Fortify sanitization support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	20	30	67%	22	30	73%	42	60	70%	12	30	40%
Aliasing	3	4	75%	1	4	25%	4	8	50%	0	4	0%
Arrays	1	2	50%	2	2	100%	3	4	75%	1	2	50%
Constants	1	2	50%	2	2	100%	3	4	75%	1	2	50%
Functions	3	5	60%	3	5	60%	6	10	60%	1	5	20%
Dynamic inclusion	3	3	100%	2	3	67%	5	6	83%	2	3	67%
Object model	6	8	75%	6	8	75%	12	16	75%	4	8	50%
Strings	3	3	100%	3	3	100%	6	6	100%	3	3	100%
Variable variables	0	3	0%	3	3	100%	3	6	50%	0	3	0%
alias1	yes			no			1	2	50%	no		
alias2	yes			no			1	2	50%	no		
alias3	yes			no			1	2	50%	no		
alias4	no			yes			1	2	50%	no		
array1	yes			yes			2	2	100%	yes		
array2	no			yes			1	2	50%	no		
constant1	yes			yes			2	2	100%	yes		
constant2	no			yes			1	2	50%	no		
function1	yes			yes			2	2	100%	yes		
function2	yes			no			1	2	50%	no		
function3	no			yes			1	2	50%	no		
function4	no			yes			1	2	50%	no		
function5	yes			no			1	2	50%	no		
include1	yes			yes			2	2	100%	yes		
include2	yes			yes			2	2	100%	yes		
include3	yes			no			1	2	50%	no		
object1	yes			yes			2	2	100%	yes		
object2	yes			no			1	2	50%	no		
object3	yes			yes			2	2	100%	yes		
object4	no			yes			1	2	50%	no		
object5	no			yes			1	2	50%	no		
object6	yes			yes			2	2	100%	yes		
object7	yes			yes			2	2	100%	yes		
object8	yes			no			1	2	50%	no		
string1	yes			yes			2	2	100%	yes		
string2	yes			yes			2	2	100%	yes		
string3	yes			yes			2	2	100%	yes		
varvar1	no			yes			1	2	50%	no		
varvar2	no			yes			1	2	50%	no		
varvar3	no			yes			1	2	50%	no		

Table 13: Fortify language support

A.2 CodeSecure

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	15	18	83%	14	18	78%	29	36	81%	11	18	61%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	2	2	100%	1	2	50%	3	4	75%	1	2	50%
Code injection	2	2	100%	1	2	50%	3	4	75%	1	2	50%
SQL injection	5	6	83%	6	6	100%	11	12	92%	5	6	83%
Server-side include	2	2	100%	1	2	50%	3	4	75%	1	2	50%
XPath injection	1	2	50%	1	2	50%	2	4	50%	0	2	0%
Cross-site scripting	3	3	100%	3	3	100%	6	6	100%	3	3	100%
arg1	no			yes			1	2	50%	no		
cmd1	yes			yes			2	2	100%	yes		
cmd2	yes			no			1	2	50%	no		
code1	yes			no			1	2	50%	no		
code2	yes			yes			2	2	100%	yes		
sql1	yes			yes			2	2	100%	yes		
sql2	no			yes			1	2	50%	no		
sql3	yes			yes			2	2	100%	yes		
sql4	yes			yes			2	2	100%	yes		
sql5	yes			yes			2	2	100%	yes		
sql6	yes			yes			2	2	100%	yes		
ssi1	yes			no			1	2	50%	no		
ssi2	yes			yes			2	2	100%	yes		
xpath1	no			yes			1	2	50%	no		
xpath2	yes			no			1	2	50%	no		
xss1	yes			yes			2	2	100%	yes		
xss2	yes			yes			2	2	100%	yes		
xss3	yes			yes			2	2	100%	yes		

Table 14: CodeSecure injection vulnerability detection

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	7	100%	3	7	43%	10	14	71%	3	7	43%
Regular expressions	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	1	1	100%	1	1	100%	2	2	100%	1	1	100%
Strings	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Cross-site scripting	2	2	100%	2	2	100%	4	4	100%	2	2	100%
preg1	yes			no			1	2	50%	no		
preg2	yes			no			1	2	50%	no		
sql1	yes			yes			2	2	100%	yes		
string1	yes			no			1	2	50%	no		
string2	yes			no			1	2	50%	no		
xss1	yes			yes			2	2	100%	yes		
xss2	yes			yes			2	2	100%	yes		

Table 15: CodeSecure sanitization support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	21	30	70%	27	30	90%	48	60	80%	18	30	60%
Aliasing	4	4	100%	3	4	75%	7	8	88%	3	4	75%
Arrays	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Constants	2	2	100%	2	2	100%	4	4	100%	2	2	100%
Functions	3	5	60%	4	5	80%	7	10	70%	2	5	40%
Dynamic inclusion	3	3	100%	2	3	67%	5	6	83%	2	3	67%
Object model	7	8	88%	8	8	100%	15	16	94%	7	8	88%
Strings	2	3	67%	3	3	100%	5	6	83%	2	3	67%
Variable variables	0	3	0%	3	3	100%	3	6	50%	0	3	0%
alias1	yes			yes			2	2	100%	yes		
alias2	yes			no			1	2	50%	no		
alias3	yes			yes			2	2	100%	yes		
alias4	yes			yes			2	2	100%	yes		
array1	no			yes			1	2	50%	no		
array2	no			yes			1	2	50%	no		
constant1	yes			yes			2	2	100%	yes		
constant2	yes			yes			2	2	100%	yes		
function1	yes			yes			2	2	100%	yes		
function2	yes			yes			2	2	100%	yes		
function3	no			yes			1	2	50%	no		
function4	no			yes			1	2	50%	no		
function5	yes			no			1	2	50%	no		
include1	yes			yes			2	2	100%	yes		
include2	yes			yes			2	2	100%	yes		
include3	yes			no			1	2	50%	no		
object1	yes			yes			2	2	100%	yes		
object2	yes			yes			2	2	100%	yes		
object3	yes			yes			2	2	100%	yes		
object4	yes			yes			2	2	100%	yes		
object5	yes			yes			2	2	100%	yes		
object6	yes			yes			2	2	100%	yes		
object7	yes			yes			2	2	100%	yes		
object8	no			yes			1	2	50%	no		
string1	yes			yes			2	2	100%	yes		
string2	no			yes			1	2	50%	no		
string3	yes			yes			2	2	100%	yes		
varvar1	no			yes			1	2	50%	no		
varvar2	no			yes			1	2	50%	no		
varvar3	no			yes			1	2	50%	no		

Table 16: CodeSecure language support

A.3 Pixy

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	18	39%	17	18	94%	24	36	67%	6	18	33%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Code injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
SQL injection	2	6	33%	6	6	100%	8	12	67%	2	6	33%
Server-side include	2	2	100%	1	2	50%	3	4	75%	1	2	50%
XPath injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Cross-site scripting	3	3	100%	3	3	100%	6	6	100%	3	3	100%
arg1	no			yes			1	2	50%	no		
cmd1	no			yes			1	2	50%	no		
cmd2	no			yes			1	2	50%	no		
code1	no			yes			1	2	50%	no		
code2	no			yes			1	2	50%	no		
sql1	yes			yes			2	2	100%	yes		
sql2	yes			yes			2	2	100%	yes		
sql3	no			yes			1	2	50%	no		
sql4	no			yes			1	2	50%	no		
sql5	no			yes			1	2	50%	no		
sql6	no			yes			1	2	50%	no		
ssi1	yes			no			1	2	50%	no		
ssi2	yes			yes			2	2	100%	yes		
xpath1	no			yes			1	2	50%	no		
xpath2	no			yes			1	2	50%	no		
xss1	yes			yes			2	2	100%	yes		
xss2	yes			yes			2	2	100%	yes		
xss3	yes			yes			2	2	100%	yes		

Table 17: Pixy injection vulnerability detection

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	6	7	86%	3	7	43%	9	14	64%	2	7	29%
Regular expressions	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Strings	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Cross-site scripting	2	2	100%	2	2	100%	4	4	100%	2	2	100%
preg1	yes			no			1	2	50%	no		
preg2	yes			no			1	2	50%	no		
sql1	no			yes			1	2	50%	no		
string1	yes			no			1	2	50%	no		
string2	yes			no			1	2	50%	no		
xss1	yes			yes			2	2	100%	yes		
xss2	yes			yes			2	2	100%	yes		

Table 18: Pixy sanitization support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	19	30	63%	21	30	70%	40	60	67%	11	30	37%
Aliasing	4	4	100%	4	4	100%	8	8	100%	4	4	100%
Arrays	2	2	100%	0	2	0%	2	4	50%	0	2	0%
Constants	2	2	100%	1	2	50%	3	4	75%	1	2	50%
Functions	5	5	100%	3	5	60%	8	10	80%	3	5	60%
Dynamic inclusion	1	3	33%	1	3	33%	2	6	33%	0	3	0%
Object model	0	8	0%	8	8	100%	8	16	50%	0	8	0%
Strings	3	3	100%	3	3	100%	6	6	100%	3	3	100%
Variable variables	2	3	67%	1	3	33%	3	6	50%	0	3	0%
alias1	yes			yes			2	2	100%	yes		
alias2	yes			yes			2	2	100%	yes		
alias3	yes			yes			2	2	100%	yes		
alias4	yes			yes			2	2	100%	yes		
array1	yes			no			1	2	50%	no		
array2	yes			no			1	2	50%	no		
constant1	yes			yes			2	2	100%	yes		
constant2	yes			no			1	2	50%	no		
function1	yes			yes			2	2	100%	yes		
function2	yes			yes			2	2	100%	yes		
function3	yes			no			1	2	50%	no		
function4	yes			yes			2	2	100%	yes		
function5	yes			no			1	2	50%	no		
include1	yes			no			1	2	50%	no		
include2	no			yes			1	2	50%	no		
include3	no			no			0	2	0%	no		
object1	no			yes			1	2	50%	no		
object2	no			yes			1	2	50%	no		
object3	no			yes			1	2	50%	no		
object4	no			yes			1	2	50%	no		
object5	no			yes			1	2	50%	no		
object6	no			yes			1	2	50%	no		
object7	no			yes			1	2	50%	no		
object8	no			yes			1	2	50%	no		
string1	yes			yes			2	2	100%	yes		
string2	yes			yes			2	2	100%	yes		
string3	yes			yes			2	2	100%	yes		
varvar1	yes			no			1	2	50%	no		
varvar2	yes			no			1	2	50%	no		
varvar3	no			yes			1	2	50%	no		

Table 19: Pixy language support

A.4 PHP-sat

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	total	perc	pass	total	perc	pass	total	perc	pass	total	perc
All	3	18	17%	16	18	89%	19	36	53%	1	18	6%
Argument injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Command injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Code injection	2	2	100%	0	2	0%	2	4	50%	0	2	0%
SQL injection	0	6	0%	6	6	100%	6	12	50%	0	6	0%
Server-side include	0	2	0%	2	2	100%	2	4	50%	0	2	0%
XPath injection	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Cross-site scripting	1	3	33%	3	3	100%	4	6	67%	1	3	33%
arg1	no			yes			1	2	50%	no		
cmd1	no			yes			1	2	50%	no		
cmd2	no			yes			1	2	50%	no		
code1	yes			no			1	2	50%	no		
code2	yes			no			1	2	50%	no		
sql1	no			yes			1	2	50%	no		
sql2	no			yes			1	2	50%	no		
sql3	no			yes			1	2	50%	no		
sql4	no			yes			1	2	50%	no		
sql5	no			yes			1	2	50%	no		
sql6	no			yes			1	2	50%	no		
ssi1	no			yes			1	2	50%	no		
ssi2	no			yes			1	2	50%	no		
xpath1	no			yes			1	2	50%	no		
xpath2	no			yes			1	2	50%	no		
xss1	yes			yes			2	2	100%	yes		
xss2	no			yes			1	2	50%	no		
xss3	no			yes			1	2	50%	no		

Table 20: PHP-sat injection vulnerability detection

Subject	**BAD tests**			**OK tests**			**Total**			**Discriminates**		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	3	7	43%	6	7	86%	9	14	64%	2	7	29%
Regular expressions	1	2	50%	1	2	50%	2	4	50%	0	2	0%
SQL injection	0	1	0%	1	1	100%	1	2	50%	0	1	0%
Strings	1	2	50%	2	2	100%	3	4	75%	1	2	50%
Cross-site scripting	1	2	50%	2	2	100%	3	4	75%	1	2	50%
preg1	yes			no			1	2	50%	no		
preg2	no			yes			1	2	50%	no		
sql1	no			yes			1	2	50%	no		
string1	no			yes			1	2	50%	no		
string2	yes			yes			2	2	100%	yes		
xss1	no			yes			1	2	50%	no		
xss2	yes			yes			2	2	100%	yes		

Table 21: PHP-sat sanitization support

Subject	BAD tests			OK tests			Total			Discriminates		
	pass	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**	**pass**	**total**	**perc**
All	7	30	23%	27	30	90%	34	60	57%	4	30	13%
Aliasing	3	4	75%	1	4	25%	4	8	50%	0	4	0%
Arrays	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Constants	0	2	0%	2	2	100%	2	4	50%	0	2	0%
Functions	0	5	0%	5	5	100%	5	10	50%	0	5	0%
Dynamic inclusion	0	3	0%	3	3	100%	3	6	50%	0	3	0%
Object model	0	8	0%	8	8	100%	8	16	50%	0	8	0%
Strings	3	3	100%	3	3	100%	6	6	100%	3	3	100%
Variable variables	1	3	33%	3	3	100%	4	6	67%	1	3	33%
alias1	yes			no			1	2	50%	no		
alias2	yes			no			1	2	50%	no		
alias3	yes			no			1	2	50%	no		
alias4	no			yes			1	2	50%	no		
array1	no			yes			1	2	50%	no		
array2	no			yes			1	2	50%	no		
constant1	no			yes			1	2	50%	no		
constant2	no			yes			1	2	50%	no		
function1	no			yes			1	2	50%	no		
function2	no			yes			1	2	50%	no		
function3	no			yes			1	2	50%	no		
function4	no			yes			1	2	50%	no		
function5	no			yes			1	2	50%	no		
include1	no			yes			1	2	50%	no		
include2	no			yes			1	2	50%	no		
include3	no			yes			1	2	50%	no		
object1	no			yes			1	2	50%	no		
object2	no			yes			1	2	50%	no		
object3	no			yes			1	2	50%	no		
object4	no			yes			1	2	50%	no		
object5	no			yes			1	2	50%	no		
object6	no			yes			1	2	50%	no		
object7	no			yes			1	2	50%	no		
object8	no			yes			1	2	50%	no		
string1	yes			yes			2	2	100%	yes		
string2	yes			yes			2	2	100%	yes		
string3	yes			yes			2	2	100%	yes		
varvar1	yes			yes			2	2	100%	yes		
varvar2	no			yes			1	2	50%	no		
varvar3	no			yes			1	2	50%	no		

Table 22: PHP-sat language support

B Micro-benchmark code

B.1 General

include_unsafe.php

```
1  <?php
2  return $_GET['text'];
3  ?>
```

include_safe.php

```
1  <?php
2  return htmlspecialchars($_GET['text']);
3  ?>
```

B.2 Injection

B.2.1 Argument injection

arg1-bad.php

```
1  <?php
2  if ( isset( $_SESSION['username'] ) )
3  {
4          $authorized = true;
5  }
6
7  if ( $authorized )        /* BAD */ /* Argument */
8  {
9          phpinfo();
10 }
11 ?>
```

arg1-ok.php

```
1  <?php
2  $authorized = false;
3  if ( isset( $_SESSION['username'] ) )
4  {
5          $authorized = true;
6  }
7
8  if ( $authorized )        /* OK */
9  {
10         phpinfo();
11 }
```

```
12 ?>
```

B.2.2 Command injection

cmd1-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 system( 'ls ' . $str ); /* BAD */ /* Command */
4 ?>
```

cmd1-ok.php

```
1 <?php
2 $str = escapeshellcmd( $_GET['text'] );
3 system( 'ls ' . $str ); /* OK */
4 ?>
```

cmd2-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 $output = array();
4 exec( 'ls ' . $str, $output ); /* BAD */ /* Command */
5 print_r( $output );
6 ?>
```

cmd2-ok.php

```
1 <?php
2 $str = escapeshellarg( $_GET['text'] );
3 $output = array();
4 exec( 'ls ' . $str, $output ); /* OK */
5 print_r( $output );
6 ?>
```

B.2.3 Code injection

code1-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 eval( 'echo "' . $str . '";' );  /* BAD */ /* Code */
4 ?>
```

code1-ok.php

```
1 <?php
2 $str = addslashes( $_GET['text'] );
3 eval( 'echo "' . $str . '";' );  /* OK */
4 ?>
```

code2-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 eval( 'echo "' . $str . '";' );  /* BAD */ /* Code */
4 ?>
```

code2-ok.php

```
1 <?php
2 $str = "Hello World!";
3 eval( 'echo "' . $str . '";' );  /* OK */
4 ?>
```

B.2.4 SQL injection

sql1-bad.php

```
1 <?php
2 $id = $_GET['id'];
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 mysql_query( $query );  /* BAD */ /* SQL */
5 ?>
```

sql1-ok.php

```
1 <?php
2 $id = mysql_real_escape_string( $_GET['id'] );
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 mysql_query( $query );  /* OK */
5 ?>
```

sql2-bad.php

```
1 <?php
2 $id = mysql_real_escape_string( $_GET['id'] );
3 $query = "SELECT * FROM users WHERE id = " . $id;
4 mysql_query( $query );  /* BAD */ /* SQL */
5 ?>
```

sql2-ok.php

```
1 <?php
2 $id = (int)$_GET['id'];
3 $query = "SELECT * FROM users WHERE id = " . $id;
4 mysql_query( $query ); /* OK */
5 ?>
```

sql3-bad.php

```
1 <?php
2 $id = $_GET['id'];
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 pg_query( $query ); /* BAD */ /* SQL */
5 ?>
```

sql3-ok.php

```
1 <?php
2 $id = pg_escape_string( $_GET['id'] );
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 pg_query( $query ); /* OK */
5 ?>
```

sql4-bad.php

```
1 <?php
2 $id = $_GET['id'];
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 mysqli_query( $db, $query ); /* BAD */ /* SQL */
5 ?>
```

sql4-ok.php

```
1 <?php
2 $id = mysqli_real_escape_string( $db, $_GET['id'] );
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 mysqli_query( $db, $query ); /* OK */
5 ?>
```

sql5-bad.php

```
1 <?php
2 $id = $_GET['id'];
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 mysqli_query( $db, $query ); /* BAD */ /* SQL */
5 ?>
```

sql5-ok.php

```
1 <?php
2 $id = $_GET['id'];
3 $stmt = mysqli_prepare($db, "SELECT * FROM users WHERE id = ?");
4 mysqli_stmt_bind_param($stmt, "s", $id);
5 mysqli_stmt_execute($stmt);      /* OK */
6 ?>
```

sql6-bad.php

```
1 <?php
2 $id = $_GET['id'];
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 sqlite_query( $db, $query );  /* BAD */ /* SQL */
5 ?>
```

sql6-ok.php

```
1 <?php
2 $id = sqlite_escape_string( $_GET['id'] );
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 sqlite_query( $db, $query );  /* OK */
5 ?>
```

B.2.5 Server-side include

ssi1-bad.php

```
1 <?php
2 echo include $_GET['action'];    /* BAD */ /* SSI */
3 ?>
```

ssi1-ok.php

```
1 <?php
2 $actions = array( '../include_safe.php' );
3 $action = $_GET['action'];
4 if ( in_array( $action, $actions ) )
5 {
6         echo include $action;    /* OK */
7 }
8 ?>
```

ssi2-bad.php

```
1 <?php
2 echo include '../include_' . $_GET['action'] . '.php'; /* BAD */ /*
      SSI */
3 ?>
```

ssi2-ok.php

```
1 <?php
2 $actions = array( 'safe' => '../include_safe.php' );
3 $includefile = $actions[$_GET['action']];
4 echo include $includefile; /* OK */
5 ?>
```

B.2.6 XPath injection

xpath1-bad.php

```
1 <?php
2 $username = $_GET['username'];
3 $password = $_GET['password'];
4 $expression = "//Employee[UserName/text()='" . $username . "' And
     Password/text()='" . $password . "']";
5 xpath_eval( $xpath, $expression );      /* BAD */ /* XPath */
6 ?>
```

xpath1-ok.php

```
1 <?php
2 $username = addslashes($_GET['username']);
3 $password = addslashes($_GET['password']);
4 $expression = "//Employee[UserName/text()='" . $username . "' And
     Password/text()='" . $password . "']";
5 xpath_eval( $xpath, $expression );      /* OK */
6 ?>
```

xpath2-bad.php

```
1 <?php
2 $username = $_GET['username'];
3 $password = $_GET['password'];
4 $expression = "//Employee[UserName/text()='" . $username . "' And
     Password/text()='" . $password . "']";
5 $xpath = new DOMXPath( $doc );
6 $xpath->query( $expression );   /* BAD */ /* XPath */
7 ?>
```

xpath2-ok.php

```
1 <?php
2 $username = addslashes($_GET['username']);
3 $password = addslashes($_GET['password']);
4 $expression = "//Employee[UserName/text()='" . $username . "' And
     Password/text()='" . $password . "']";
```

```
5 $xpath = new DOMXPath( $doc );
6 $xpath->query( $expression );    /* OK */
7 ?>
```

B.2.7 Cross-site scripting

xss1-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 echo $str;   /* BAD */ /* XSS */
4 ?>
```

xss1-ok.php

```
1 <?php
2 $str = htmlspecialchars( $_GET['text'] );
3 echo $str;   /* OK */
4 ?>
```

xss2-bad.php

```
1 <html>
2 <body>
3 <?=$_GET['text'] /* BAD */ /* XSS */?>
4 </body>
5 </html>
```

xss2-ok.php

```
1 <html>
2 <body>
3 <?=htmlspecialchars($_GET['text']) /* OK */?>
4 </body>
5 </html>
```

xss3-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 printf( "%s", $str );    /* BAD */ /* XSS */
4 ?>
```

xss3-ok.php

```
1 <?php
2 $str = htmlspecialchars( $_GET['text'] );
3 printf( "%s", $str );    /* OK */
4 ?>
```

B.3 Language semantics

B.3.1 Aliasing

alias1-bad.php

```
1  <?php
2  function clean( $a )
3  {
4      $a = htmlspecialchars( $a );
5  }
6
7  $str = $_GET['text'];
8  clean( $str );
9  echo $str;  /* BAD */ /* XSS */
10 ?>
```

alias1-ok.php

```
1  <?php
2  function clean( &$a )
3  {
4      $a = htmlspecialchars( $a );
5  }
6
7  $str = $_GET['text'];
8  clean( $str );
9  echo $str;  /* OK */
10 ?>
```

alias2-bad.php

```
1  <?php
2  function clean( $a )
3  {
4      $a = htmlspecialchars( $a );
5  }
6
7  $str = $_GET['text'];
8  clean( $str );
9  echo $str;  /* BAD */ /* XSS */
10 ?>
```

alias2-ok.php

```
1  <?php
2  function clean( $a )
3  {
4      $a = htmlspecialchars( $a );
```

```
5 }
6
7 $str = $_GET['text'];
8 clean( &$str );
9 echo $str;  /* OK */
10 ?>
```

alias3-bad.php

```
1 <?php
2 $a = $_GET['text'];
3 $b = $a;
4 $b = htmlspecialchars( $b );
5 echo $a;          /* BAD */ /* XSS */
6 ?>
```

alias3-ok.php

```
1 <?php
2 $a = $_GET['text'];
3 $b = &$a;
4 $b = htmlspecialchars( $b );
5 echo $a;          /* OK */
6 ?>
```

alias4-bad.php

```
1 <?php
2 $b = &$a;
3 $b = $_GET['text'];
4 echo $a;          /* BAD */ /* XSS */
5 ?>
```

alias4-ok.php

```
1 <?php
2 $b = &$a;
3 $b = htmlspecialchars( $_GET['text'] );
4 echo $a;          /* OK */
5 ?>
```

B.3.2 Arrays

array1-bad.php

```
1 <?php
2 $arr[] = $_GET['text'];
3 echo $arr[0];     /* BAD */ /* XSS */
4 ?>
```

array1-ok.php

```
1 <?php
2 $arr[] = htmlspecialchars( $_GET['text'] );
3 echo $arr[0];    /* OK */
4 ?>
```

array2-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 $arr = compact( 'str' );
4 echo $arr['str'];        /* BAD */ /* XSS */
5 ?>
```

array2-ok.php

```
1 <?php
2 $str = htmlspecialchars( $_GET['text'] );
3 $arr = compact( 'str' );
4 echo $arr['str'];        /* OK */
5 ?>
```

B.3.3 Constants

constant1-bad.php

```
1 <?php
2 define( 'CONSTANT', $_GET['text'] );
3 echo CONSTANT;    /* BAD */ /* XSS */
4 ?>
```

constant1-ok.php

```
1 <?php
2 define( 'CONSTANT', htmlspecialchars( $_GET['text'] ) );
3 echo CONSTANT;    /* OK */
4 ?>
```

constant2-bad.php

```
1 <?php
2 define( '...', $_GET['text'] );
3 echo constant( '...' ); /* BAD */ /* XSS */
4 ?>
```

constant2-ok.php

```
1 <?php
2 define( '...', htmlspecialchars( $_GET['text'] ) );
3 echo constant( '...' ); /* BAD */ /* XSS */
4 ?>
```

B.3.4 Functions

function1-bad.php

```
1 <?php
2 function getValue()
3 {
4         return $_GET['text'];
5 }
6
7 echo getValue();          /* BAD */ /* XSS */
8 ?>
```

function1-ok.php

```
1 <?php
2 function getValue()
3 {
4         return htmlspecialchars( $_GET['text'] );
5 }
6
7 echo getValue();          /* OK */
8 ?>
```

function2-bad.php

```
1 <?php
2 function sanitize( $a )
3 {
4         return $a;
5 }
6
7 $str = sanitize( $_GET['text'] );
8 echo $str;       /* BAD */ /* XSS */
9 ?>
```

function2-ok.php

```
1 <?php
2 function sanitize( $a )
3 {
4         return htmlspecialchars( $a );
5 }
```

```
6
7 $str = sanitize( $_GET['text'] );
8 echo $str;        /* OK */
9 ?>
```

function3-bad.php

```
1 <?php
2 function getValue()
3 {
4         return $_GET['text'];
5 }
6
7 $func = 'getValue';
8 echo $func();    /* BAD */ /* XSS */
9 ?>
```

function3-ok.php

```
1 <?php
2 function getValue()
3 {
4         return htmlspecialchars( $_GET['text'] );
5 }
6
7 $func = 'getValue';
8 echo $func();    /* OK */
9 ?>
```

function4-bad.php

```
1 <?php
2 function getValue( $name )
3 {
4         return $_GET[$name];
5 }
6
7 function valueFromFunction( $func )
8 {
9         return $func('text');
10 }
11
12 $str = valueFromFunction( 'getValue' );
13 echo $str;        /* BAD */ /* XSS */
14 ?>
```

function4-ok.php

```
1 <?php
```

```
2  function getValue( $name )
3  {
4          return $_GET[$name];
5  }
6
7  function valueFromFunction( $func )
8  {
9          return htmlspecialchars( $func('text') );
10 }
11
12 $str = valueFromFunction( 'getValue' );
13 echo $str;      /* OK */
14 ?>
```

function5-bad.php

```
1  <?php
2  function f1()
3  {
4          function g()
5          {
6                  return $_GET['text'];
7          }
8  }
9
10 function f2()
11 {
12         function g()
13         {
14                 return htmlspecialchars($_GET['text']);
15         }
16 }
17
18 f1();
19 $str = g();
20 echo $str;      /* BAD */ /* XSS */
21 ?>
```

function5-ok.php

```
1  <?php
2  function f1()
3  {
4          function g()
5          {
6                  return $_GET['text'];
7          }
8  }
9
```

```
10 function f2()
11 {
12         function g()
13         {
14                 return htmlspecialchars($_GET['text']);
15         }
16 }
17
18 f2();
19 $str = g();
20 echo $str;      /* OK */
21 ?>
```

B.3.5 Dynamic inclusion

include1-bad.php

```
1 <?php
2 echo include '../include_unsafe.php';  /* BAD */ /* XSS */
3 ?>
```

include1-ok.php

```
1 <?php
2 echo include '../include_safe.php';  /* OK */
3 ?>
```

include2-bad.php

```
1 <?php
2 if ( $x == 5 )
3 {
4     echo include '../include_unsafe.php';   /* BAD */ /* XSS */
5 }
6 ?>
```

include2-ok.php

```
1 <?php
2 if ( $x == 5 )
3 {
4     echo include '../include_safe.php';   /* OK */
5 }
6 ?>
```

include3-bad.php

```
1 <?php
```

```
2  if ( $x == 5 )
3  {
4      echo include '../include_unsafe.php';    /* BAD */ /* XSS */
5  }
6  ?>
```

include3-ok.php

```
1  <?php
2  if ( false )
3  {
4      echo include '../include_unsafe.php';    /* OK */
5  }
6  ?>
```

B.3.6 Object model

object1-bad.php

```
1  <?php
2  class Object
3  {
4          var $value;
5  }
6
7  $obj = new Object();
8  $obj->value = $_GET['text'];
9  echo $obj->value;    /* BAD */ /* XSS */
10 ?>
```

object1-ok.php

```
1  <?php
2  class Object
3  {
4          var $value;
5  }
6
7  $obj = new Object();
8  $obj->value = htmlspecialchars( $_GET['text'] );
9  echo $obj->value;    /* OK */
10 ?>
```

object2-bad.php

```
1  <?php
2  class Object
3  {
```

```
 4         var $value;
 5
 6         function sanitize()
 7         {
 8                 $this->value = htmlspecialchars( $this->value );
 9         }
10 }
11
12 $obj = new Object();
13 $obj->value = $_GET['text'];
14 echo $obj->value;    /* BAD */ /* XSS */
15 ?>
```

object2-ok.php

```
 1 <?php
 2 class Object
 3 {
 4         var $value;
 5
 6         function sanitize()
 7         {
 8                 $this->value = htmlspecialchars( $this->value );
 9         }
10 }
11
12 $obj = new Object();
13 $obj->value = $_GET['text'];
14 $obj->sanitize();
15 echo $obj->value;    /* OK */
16 ?>
```

object3-bad.php

```
1 <?php
2 $obj->value = $_GET['text'];
3 echo $obj->value;        /* BAD */ /* XSS */
4 ?>
```

object3-ok.php

```
1 <?php
2 $obj->value = htmlspecialchars( $_GET['text'] );
3 echo $obj->value;        /* OK */
4 ?>
```

object4-bad.php

```
1 <?php
```

```
2 $obj = (object)array( 'value' => $_GET['text'] );
3 echo $obj->value;         /* BAD */ /* XSS */
4 ?>
```

object4-ok.php

```
1 <?php
2 $obj = (object)array( 'value' => htmlspecialchars( $_GET['text'] ) );
3 echo $obj->value;         /* OK */
4 ?>
```

object5-bad.php

```
1  <?php
2  class Object
3  {
4          var $value;
5
6          function Object()
7          {
8                  $this->__construct();
9          }
10
11         function __construct()
12         {
13                 $this->value = $_GET['text'];
14         }
15 }
16
17 $obj = new Object();
18 echo $obj->value;         /* BAD */ /* XSS */
19 ?>
```

object5-ok.php

```
1  <?php
2  class Object
3  {
4          var $value;
5
6          function Object()
7          {
8                  $this->__construct();
9          }
10
11         function __construct()
12         {
13                 $this->value = htmlspecialchars( $_GET['text'] );
14         }
```

```
15 }
16
17 $obj = new Object();
18 echo $obj->value;          /* OK */
19 ?>
```

object6-bad.php

```
1 <?php
2 $obj = (object)$_GET['text'];
3 echo $obj->scalar;          /* BAD */ /* XSS */
4 ?>
```

object6-ok.php

```
1 <?php
2 $obj = (object)htmlspecialchars( $_GET['text'] );
3 echo $obj->scalar;          /* OK */
4 ?>
```

object7-bad.php

```
1 <?php
2 class Object
3 {
4         var $value;
5 }
6
7 $obj = new Object();
8 $obj->duck = $_GET['text'];
9 echo $obj->duck;       /* BAD */ /* XSS */
10 ?>
```

object7-ok.php

```
1 <?php
2 class Object
3 {
4         var $value;
5 }
6
7 $obj = new Object();
8 $obj->duck = htmlspecialchars( $_GET['text'] );
9 echo $obj->duck;       /* OK */
10 ?>
```

object8-bad.php

```
1 <?php
```

```
2  class Object
3  {
4          var $value;
5
6          function __toString()
7          {
8                  return $this->value;
9          }
10 }
11
12 $obj = new Object();
13 $obj->value = $_GET['text'];
14 echo $obj;      /* BAD */ /* XSS */
15 ?>
```

object8-ok.php

```
1  <?php
2  class Object
3  {
4          var $value;
5
6          function __toString()
7          {
8                  return htmlspecialchars( $this->value );
9          }
10 }
11
12 $obj = new Object();
13 $obj->value = $_GET['text'];
14 echo $obj;      /* OK */
15 ?>
```

B.3.7 Strings

string1-bad.php

```
1  <?php
2  $text = $_GET['text'];
3  $str = "You entered: $text";
4  echo $str;      /* BAD */ /* XSS */
5  ?>
```

string1-ok.php

```
1  <?php
2  $text = htmlspecialchars( $_GET['text'] );
3  $str = "You entered: $text";
```

```
4 echo $str;        /* OK */
5 ?>
```

string2-bad.php

```
1 <?php
2 $str = "You entered: $_GET[text]";
3 echo $str;        /* BAD */ /* XSS */
4 ?>
```

string2-ok.php

```
1 <?php
2 $str = htmlspecialchars( "You entered: $_GET[text]" );
3 echo $str;        /* OK */
4 ?>
```

string3-bad.php

```
1 <?php
2 $str = "You entered: {$_GET['text']}";
3 echo $str;        /* BAD */ /* XSS */
4 ?>
```

string3-ok.php

```
1 <?php
2 $str = htmlspecialchars( "You entered: {$_GET['text']}" );
3 echo $str;        /* OK */
4 ?>
```

B.3.8 Variable variables

varvar1-bad.php

```
1 <?php
2 $unsafe = $_GET['text'];
3 $safe = htmlspecialchars( $unsafe );
4 $varname = "unsafe";
5 echo $$varname; /* BAD */ /* XSS */
6 ?>
```

varvar1-ok.php

```
1 <?php
2 $unsafe = $_GET['text'];
3 $safe = htmlspecialchars( $unsafe );
4 $varname = "safe";
```

```
5 echo $$varname; /* OK */
6 ?>
```

varvar2-bad.php

```
1 <?php
2 $unsafe = $_GET['text'];
3 $safe = htmlspecialchars( $unsafe );
4 $varname = addslashes($_GET['varname']);
5 echo $$varname; /* BAD */ /* XSS */
6 ?>
```

varvar2-ok.php

```
1 <?php
2 $unsafe = $_GET['text'];
3 $safe = htmlspecialchars( $unsafe );
4 $unsafe = null;
5 $varname = addslashes($_GET['varname']);
6 echo $$varname; /* OK */
7 ?>
```

varvar3-bad.php

```
1 <?php
2 $x = "Harmless";
3 $varname = "x";
4 $$varname = $_GET['text'];
5 echo $x;          /* BAD */ /* XSS */
6 ?>
```

varvar3-ok.php

```
1 <?php
2 $x = "Harmless";
3 $varname = "x";
4 $$varname = htmlspecialchars( $_GET['text'] );
5 echo $x;          /* OK */
6 ?>
```

B.4 Sanitization

B.4.1 Regular expressions

preg1-bad.php

```
1 <?php
```

```
2 $str = $_GET['text'];
3 if ( preg_match( "/^(.+)$/", $str ) )
4 {
5         echo $str;       /* BAD */ /* XSS */
6 }
7 ?>
```

preg1-ok.php

```
1 <?php
2 $str = $_GET['text'];
3 if ( preg_match( "/^([-\@\w\s.]+)$/", $str ) )
4 {
5         echo $str;       /* OK */
6 }
7 ?>
```

preg2-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 echo preg_replace( '/[\\x00\\\\\'"]/', '', $str );       /* BAD */ /*
    XSS */
4 ?>
```

preg2-ok.php

```
1 <?php
2 $str = $_GET['text'];
3 echo preg_replace( '/[&"<>]/', '', $str );       /* OK */
4 ?>
```

B.4.2 SQL injection

sql1-bad.php

```
1 <?php
2 $id = htmlspecialchars( $_GET['id'] );
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 mysql_query( $query );   /* BAD */ /* SQL */
5 ?>
```

sql1-ok.php

```
1 <?php
2 $id = mysql_real_escape_string( $_GET['id'] );
3 $query = "SELECT * FROM users WHERE id = '" . $id . "'";
4 mysql_query( $query );   /* OK */
5 ?>
```

B.4.3 Strings

string1-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 $search = array( "\0", '\\', '\'', '"' );
4 $replace = array( '\\0', '\\\\', '\\\'', '\\"' );
5 echo str_replace( $search, $replace, $str );     /* BAD */ /* XSS */
6 ?>
```

string1-ok.php

```
1 <?php
2 $str = $_GET['text'];
3 $search = array( '&', '"', '<', '>' );
4 $replace = array( '&amp;', '&quot;', '&lt;', '&gt;' );
5 echo str_replace( $search, $replace, $str );     /* OK */
6 ?>
```

string2-bad.php

```
1 <?php
2 $text = $_GET['text'];
3 $str = "You entered: " . $text;
4 echo $str;  /* BAD */ /* XSS */
5 ?>
```

string2-ok.php

```
1 <?php
2 $text = $_GET['text'];
3 $str = "You entered: " . $text;
4 echo substr( $str, 0, 12 );  /* OK */
5 ?>
```

B.4.4 Cross-site scripting

xss1-bad.php

```
1 <?php
2 $str = addslashes( $_GET['text'] );
3 echo $str;  /* BAD */ /* XSS */
4 ?>
```

xss1-ok.php

```
1 <?php
```

```
2 $str = htmlspecialchars( $_GET['text'] );
3 echo $str;  /* OK */
4 ?>
```

xss2-bad.php

```
1 <?php
2 $str = $_GET['text'];
3 echo $str;  /* BAD */ /* XSS */
4 ?>
```

xss2-ok.php

```
1 <?php
2 $str = (int)$_GET['text'];
3 echo $str;  /* OK */
4 ?>
```