International Journal of Electrical and Computer Engineering (IJECE)
Vol. 10, No. 6, December 2020, pp. 6061~6070
ISSN: 2088-8708, DOI: 10.11591/ijece.v10i6.pp6061-6070

Automated server-side model for recognition of security vulnerabilities in scripting languages

Rabab F. Abdel-Kader[1], Mona Nashaat[2], Mohamed I. Habib[3], Hani M. K. Mahdi[4]
[1,2,3]Department of Electrical Engineering, Faculty of Engineering, Port Said University, Egypt
[4]Faculty of Engineering, Ain Shams University, Egypt

Article Info

Article history:

Received Dec 8, 2019
Revised May 17, 2020
Accepted May 27, 2020

Keywords:

Data flow computing
Data security
Object-oriented programming
Software protection
Software testing

ABSTRACT

With the increase of global accessibility of web applications, maintaining a reasonable security level for both user data and server resources has become an extremely challenging issue. Therefore, static code analysis systems can help web developers to reduce time and cost. In this paper, a new static analysis model is proposed. This model is designed to discover the security problems in scripting languages. The proposed model is implemented in a prototype SCAT, which is a static code analysis tool. SCAT applies the phases of the proposed model to catch security vulnerabilities in PHP 5.3. Empirical results attest that the proposed prototype is feasible and is able to contribute to the security of real-world web applications. SCAT managed to detect 94% of security vulnerabilities found in the testing benchmarks; this clearly indicates that the proposed model is able to provide an effective solution to complicated web systems by offering benefits of securing private data for users and maintaining web application stability for web applications providers.

Corresponding Author:

Rabab F. Abdel-Kader,
Department of Electrical Engineering,
Port-Said University,
Port-Said, 42523, Egypt.
Email: rababfakader@eng.psu.edu.eg

1. INTRODUCTION

Web applications are famous for security vulnerabilities that can be exploited by malicious users. According to positive technologies (PT) [1], which is one of the top ten worldwide vendors of vulnerability assessment systems, a percentage that ranges from 60% to 75% (depending on the analysis method) of the analyzed sites contained critical vulnerabilities. A big portion of the detected vulnerabilities belongs to the Cross-Site Scripting weakness and SQL injection. These kinds of vulnerabilities are caused by faulty code. For example, cross-site script insertion is caused by the lack of sanitization for data supplied from the user, code injection vulnerabilities result from the mixing of code and data. Another obvious point in these statistics is that the largest share of web application vulnerabilities belongs to the general class of taint-style vulnerabilities [2]. Taint-style vulnerabilities are a class of vulnerabilities that are a direct result of a lack of or inadequate sanitization or validation of the integrity of data that is processed by the application.

This paper presents a new static code analysis model that is targeted to spot security vulnerabilities in scripting languages. The model is also implemented in a prototype called (SCAT), which is implemented to scan the applications and detect: cross-site scripting [2], SQL injection [3], remote code execution, remote command execution, and XPath injection vulnerabilities [4]. This paper is organized as follows: the next section illustrates the background and related work, Section 3 represents a detailed description of the model implementation, and Section 4 describes the assessment methodology. Section 5 presents the empirical results, while Section 6 represents the conclusions.

2. BACKGROUND AND RELATED WORKS

Static code analysis is a well-known approach that can be used for detecting security problems in any program without the need of executing it [5]. Static code analyzers are usually used early in development, which reduces the cost of fixing any error found in the code. However, it is known that static analysis tools produce too many false positives; this is when a static analysis tool inappropriately marks a problem-free section of code as vulnerable [6]. This means that the output from a security tool usually requires human review.

There exist a considerable number of security assessment models for scripting languages; Pixy [7] is one good example for such models, it is an open source static code analyzer performs automatic scans of PHP 4 source code. Pixy takes a PHP program as input and outputs possible vulnerable points. Yu et al., [8] also used static analysis to detect vulnerabilities in PHP 4 scripts and create string signatures for these vulnerabilities. They implemented this process in Stranger, which stands for STRing AutomatoN GEneratoR. Stranger is a string analysis tool for PHP web applications [8]. However, the tool does not support a recent version of PHP.

Saner [9] is another security analyzer that uses an approach that consists of a static analysis component to identify the flows of input values from sources to sensitive sinks. Nevertheless, the tool does not support any object-oriented features in PHP. The author of RIPS [10] used an approach to build a static source code analyzer written in PHP using the built-in tokenizer functions. The last version of RIPS that was released in 2014 is implemented to find a wide range of known vulnerabilities [10].

3. PROPOSED MODEL IMPLEMENTATION

The proposed model is designed to detect the security issues in scripting languages like PHP. Figure 1 shows the underline architecture of the proposed model which was applied in SCAT. The proposed model first transforms the input program into a parse tree [11]. In the prototype, the lexical analyzer is generated from the famous lexical analyzer generator for Java (JFlex) [12]. While the parser in the prototype is built using a modified version of The Constructor of Useful Parser (CUP) v0.10 tool [13]. Some modifications had to be made in the source files of CUP, and so the production's symbol name, symbol index and length can be accessed by the rule actions.

Finally, in data flow analysis, the constructed parse tree is transformed into a control flow graph (CFG) for each encountered function [14]. The proposed model enforces a list of standards that must be satisfied by the performed data flow analysis. First, the output CFG must maintain the flow of types of each program point during execution, such requirement is necessary due to the dynamically typed nature of PHP [15]. Second, it is required to collect information about the complete program putting all function calls in considerations; this is the main role of the Inter-procedural data flow analysis phase [16]. Finally, the data flow analysis collects all associated information for each node [17].

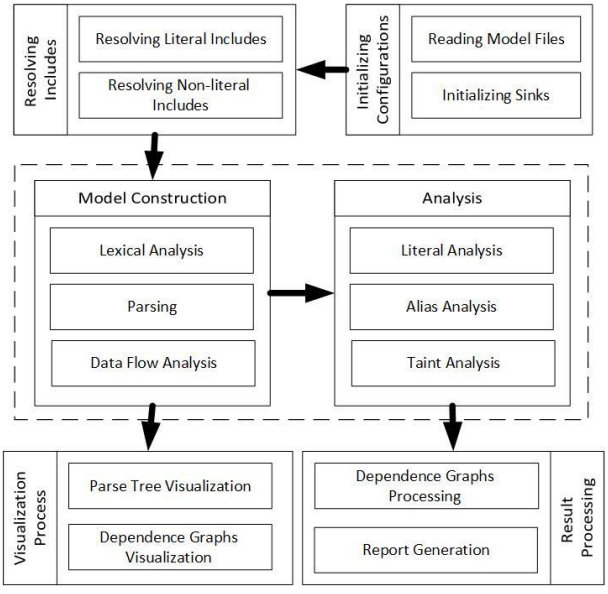

Figure 1. The proposed model system architecture

The produced information from the data flow analysis step is now ready for the taint analysis step [19]. Taint analysis simply determines for each program point whether it may hold a tainted value or not. In order to improve the capability of the analysis phase, alias analysis is performed first; alias analysis is concerned with collecting the alias relationships for all variables in the input program [18].

The parse tree is first generated into DOT language, and then the Dot file is transformed into a visualized tree using Graphviz [19]. We use Graphviz class libraries to create a graphical representation for parse tree and dependence graphs for program points that may receive tainted data during execution time. Figure 2 shows a representation of the implementation of these functionalities within the proposed model structure.

The last phase is result processing and report generating. For each sensitive sink that can receive tainted data during execution, the model generates a vulnerability record. The record shows the file name that contains the sensitive sink with the tainted data, the line number and the type of the detected vulnerability. The proposed model also creates dependence graphs for the tainted variable [9].

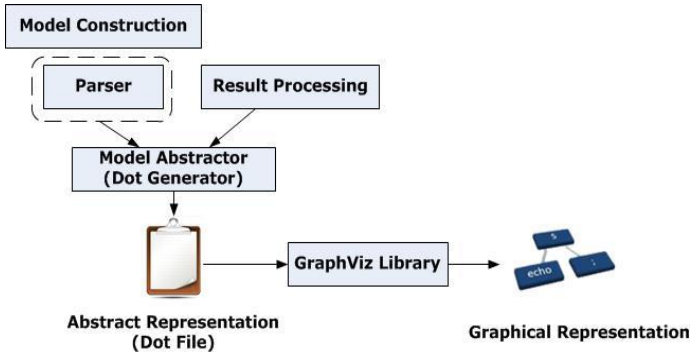

Figure 2. Visualization features implementation

Creating a security model for scripting languages like PHP requires giving extra attention to a list of language features such as dynamic includes, dynamic typing, and dynamic object reference.

a. Dynamic includes: PHP allows dynamic file inclusion, in which the file name and path is formed dynamically in execution time. The proposed model performs recursive literal analysis phase in order to resolve dynamic-included files. Figure 3 shows a simplified form of the algorithm for resolving dynamic inclusion.
b. Dynamic Typing: the proposed model determines the flow of types for each variable in the program. It keeps track of each point in the program that may result in changing variable type such as assignment statements, calling to functions and Set and Unset functions [20]. In each of the aforementioned cases, type analysis investigates the corresponding CFG and update related variables types.
c. Dynamic object reference: The problem with many existing approaches is the lack of understanding any OOP features in scripting languages, for example, pixy marks any custom object as a tainted program point. Similarly, it marks all user-defined method return values as tainted. The proposed model applies an algorithm to both user-defined classes. The algorithm main function is to simulate stack and heap data structures for custom classes, object references, user-defined methods and variables, namespaces, and interfaces. Thus, the algorithm can maintain relations between all defined objects, their custom classes, methods, and variables. During the analysis phase each custom object is resolved with its class definition, this helps to detect vulnerabilities in user-defined objects and methods.

```
LiteralAnalysis.analyasze ( )
allIncludeNodes= LiteralAnalysis.CollectIncludeNodes ( )
foreach CfgIncludeNode in AllIncludeNodes
        IncLiteral=a.GetLiteral (CfgIncludeNode.GetIncludeinfo)
        Result r=TryToResolve (IncludeLiteral, CfgIncludeNode)
        If (r==Resolved)
                resolvedInclude.add (CfgIncludeNode)
        else If (r==More)
                CyclicInclude.add (CfgIncludeNode)
        else If (r==NotFound)
                NotFoundInclude.add (CfgIncludeNode)
End foreach
```

Figure 3. Resolving dynamic inclusions

4. RESEARCH METHOD

4.1. Sub evaluation procedure

Evaluating the proposed model is mainly based on finding out how well the prototype SCAT confirms to static code analysis tools requirements such as accuracy, robustness, usability, and responsiveness [21, 22]. For this purpose, two different sets of benchmark tests were performed. The evaluation process presented here adapted the same structure used by Poel [23]. However, this structure is extended by computing the evaluation metrics for each tool. Evaluation metrics computed for each tool include precision, recall, specificity and $F_{measure}$ [24].

- Precision: is the ratio of the number of true positives (*TP*) over the number of reported errors, which include the reported true positives and false positives (*TP+FP*).

$$Precision = TP / (TP + FP) \quad (1)$$

- Recall: is the ratio of the number of true positives (*TP*) over the number of actual errors, which is the sum of reported true positives, and false negatives that were not detected (*TP+FN*).

$$Recall = TP / (TP + FN) \quad (2)$$

- Specificity: is the ratio of the number of true negatives (*TN*) over the sum of true negatives and false positives (*TN+FP*).

$$Specificity = TN / (TN + FP) \quad (3)$$

- $F_{measure}$: provides an aggregate measure for precision and recall.

$$F_{measure} = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (4)$$

Two others commonly used $F_{measures}$ are the $F_{2\text{-}measure}$, which weights recall higher than precision and the $F_{0.5\text{-}measure}$, which puts more emphasis on precision than recall [23]. The formula for $F_{\beta\text{-}measure}$ is:

$$F_{\beta-measure} = \frac{(1+\beta^2)(Precision \times Recall)}{\beta^2 \times Precision + Recall} \quad (5)$$

$F_{measures}$ ranges between 0 and 1 for a given tool, the three measures can be used to introduce a ranking of the performance of several tools.

The methodology of the evaluation process is introduced in Figure 4, the process starts with choosing a group of related static analysis tools, and then each tool within the group is used to analyze both sets of benchmarks: Intra-Benchmark tests and Inter-Benchmark tests, finally the results obtained by each tool are manually analyzed in order to compute the evaluation metrics. Implementation codes are available through the link https://sourceforge.net/p/scat-static-analysis/code/ci/master/tree/. Before the empirical results reviewed, both benchmarks tests and the group of related tools involved in the evaluation process are further explained in the next three subsections.

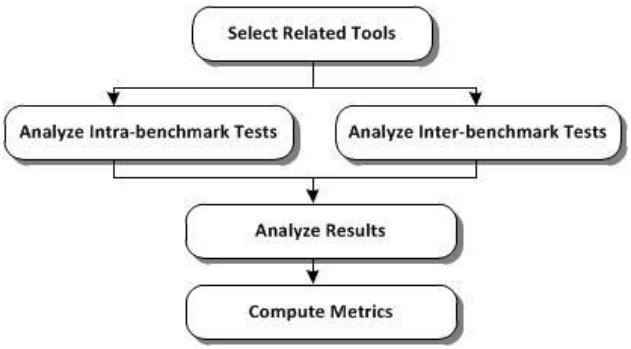

Figure 4. Evaluation process methodology

4.2. Benchmarks tests

4.2.1. Intra-benchmarks tests

The Intra-benchmark tests consist of real-world web applications written in PHP. These applications are chosen with variety in its size, PHP supported version, coding style, and code complexity. The complete list of these applications is shown in Table 1. For each application, the table shows its name, the application version that was used in the experiments, the application type and code size of each application measured by the number of code lines (LOC), the code size was calculated using PHPLoc Pear package.

Some of the tested applications are deliberately vulnerable web-applications that are provided as a target for web-security scanners. These applications are Exploit.co.il, Mutillidae and Damn Vulnerable Web App (DVWA). The rest of the tested applications are real-world applications written in PHP like PBL Guestbook 1.32, MyBloggie 2.1.6, WordPress 1.5.1.3, and MyEasyMarket 4.1.

Intra-benchmark tests boil down to running a static code analysis on each one of these applications, then the results obtained by each tool are manually analyzed to gather basic information about each tool such as the total analysis time, the total number of spotted vulnerabilities (TP) and the number of false positives (FP) [6]. The experiments focus on a set of taint-style vulnerabilities, which are XSS, SQL Injections, Command Injection, and Code Injection, as these are the most frequently detected vulnerabilities by the selected set of static code analysis tools [25-27].

Table 1. List of Intra-benchmark applications

Application Name	Version	Application Type	LOC
Mutillidae	2.3.7	Vulnerable Web Application	103114
DVWA	1.0.7	Vulnerable Web Application	32315
Exploit.co.il	1.0.0	Vulnerable Web Application	5109
PBLGuestbook	1.32	Guest Book Application	1566
WordPress	1.5.1.3	Content Management System	31010
MyEasyMarket	4.1	Shopping Cart Application	2569
MyBloggie	2.1.6	Weblog System	9461

4.2.2. Intre-benchmarks tests

The inter-benchmark consists of 110 small php test cases stating 55 test cases, these cases are divided into three categories, which are language support, vulnerability detection, and sanitization routine support. Nico L. De Poel [24] used these test cases to evaluate a collection of commercial and open source static code analyzers. Each test case consists of a vulnerable program that includes a security problem, and a resolved program that resolves the vulnerability problem. The evaluation process focuses on both true positive and false positive situations, so for each test case, a given tool is said to pass the test if it succeeded to detect the vulnerability in the vulnerable file and did not fire alarm within the resolved file.

4.3. Selected tools

A wide range of related tools was investigated in order to choose the tools which are eligible to engage in the evaluation process. These tools must allow comparing their performance, usability and the range of covered vulnerabilities. This was the main reason for choosing open source tools, as they offer full access to the source code, which helps in understanding the evaluation results. However, some tools were discarded such as Ardilla [28] and IPAAS [29] since they do not provide source code yet and TAP [30] which is a recent tool to detect vulnerability using deep learning.

The set of selected tools includes Pixy, RIPS and Yet another Source Code Analyzer (YASCA). Pixy is the first and popular open source static code analysis tool targeted for PHP [7]. The second tool in the set is RIPS, which is a static code analyzer that was developed by Johannes Dahse. It is written in PHP and developed to detect a wide range of taint style vulnerabilities. The third tool in the set is YASCA which was initially created by Michael V. Scovetta [31]. It can scan source code written in PHP and other languages.

5. EMPIRICAL RESULTS

5.1. Analysis Time-Based Comparison

The analysis time is computed for each intra-benchmark test, while the analysis time in inter-benchmark tests was ignored, as it was significantly small. Table 2 shows the analysis time for intra-benchmark tests. SCAT took a noticeably long time in some applications; a significant part of this time returns to file inclusion resolution phase; however, this delay should be acceptable comparing the eminent number of the vulnerabilities detected by SCAT in these applications.

Table 2. Analysis time-based comparisons

Application	SCAT	Pixy	RIPS	YASCA
WordPress	20.403	Parse Error	7.223	14.09
Exploit	5.971	Parse Error	1.157	17.61
Mutillidae	27.563	Parse Error	46.479	60.14
MyBloggie	31.531	38.040	8.921	27.05
PBLGuestbook	16.981	7.260	0.211	45.12
MyEasyMarket	16.560	22.420	0.943	14.56
DVWA	3.879	Parse Error	7.220	21.98

5.2. Vulnerability detection-based comparison

5.2.1. Vulnerability detection in intra-benchmark tests

Table 3 shows the number of vulnerabilities detected by each tool in intra-benchmark tests. The results show that for most applications, SCAT managed to achieve better results than other tools. For example, in XSS detection, SCAT succeeded to detect XSS vulnerabilities that other tools failed to detect. Also, in WordPress application, SCAT managed to detect XSS vulnerability in "searchform.php" file in which WordPress allows remote attackers to inject arbitrary web script or HTML via the PHP_SELF portion of a Uniform Resource Identifier (URI) to "index.php". On the other hand, RIPS kept firing false alarms in files such as "archive.php" and "index.php" in which "searchform.php" file is included. While Pixy failed to parse WordPress among other applications that use some advanced PHP 5 features.

Table 3. Vulnerability detection in intra-benchmark tests

Vul. Type	Benchmark	SCAT			Pixy			RIPS			YASCA		
		TP	FP	FP%	TP	FP	FP%	TP	FP	FP%	TP	FP	FP%
Cross-Site Scripting	WordPress	221	15	6.36		Null		93	9	8.82	5	0	0
	Exploit	0	1	100		Null		0	2	100	0	1	100
	Mutillidae	54	2	3.57		Null		64	123	65.78	8	1	11.11
	MyBloggie	35	4	10.26	37	4	9.76	20	27	57.45	2	3	60
	PBLGuestbook	1	0	0	0	1	100	0	0	-	1	1	50
	MyEasyMarket	16	0	0	7	0	0	0	0	-	0	0	-
	DVWA	5	1	16.67		Null		2	14	87.50	1	3	75
SQL Injection	WordPress	0	0	-		Null		0	0	-	0	0	-
	Exploit	35	0	0		Null		35	0	0	1	4	80
	Mutillidae	1	0	0		Null		0	0	-	0	0	-
	MyBloggie	1	0	0	1	4	80	0	1	100	3	0	0
	PBLGuestbook	7	0	0	7	1	12.5	1	7	87.5	0	4	100
	MyEasyMarket	29	0	0	15	0	0	0	3	100	3	1	25
	DVWA	8	1	11		Null		3	3	50	0	1	100
Command Injection	WordPress	0	0	-		Null		0	1	100	0	0	-
	Exploit	1	0	0		Null		0	0	-	0	0	-
	Mutillidae	1	0	0		Null		1	4	80	1	0	0
	MyBloggie	0	0	-	0	0	-	0	0	-	0	0	-
	PBLGuestbook	0	0	-	0	0	-	0	0	-	0	0	-
	MyEasyMarket	0	0	-	0	0	-	0	0	-	0	0	-
	DVWA	4	0	0		Null		4	2	33.3	1	5	83.3
Code Execution	WordPress	1	0	0		Null		1	6	85.7	0	0	-
	Exploit	0	0	-		Null		0	0	-	0	0	-
	Mutillidae	0	0	-		Null		3	27	90	0	0	-
	MyBloggie	16	5	23.8	0	0	-	0	3	100	12	0	0
	PBLGuestbook	0	0	-	0	0	-	0	0	-	0	0	-
	MyEasyMarket	0	0	-	0	0	-	0	0	-	0	0	-
	DVWA	0	0	-		Null		0	2	100	0	0	-

In order to standardize the results, Precision value for the detected vulnerabilities is calculated for each tool. Table 4 shows these calculated values, the results are categories by vulnerability type, the value calculated for each tool shows the average of the precision values achieved by each tool in the tested applications. The table clearly indicates that SCAT achieved the highest precision for XSS vulnerabilities.

The precision values for SQL Injection vulnerabilities (Precision) for each tool are shown in the second row of the table; the results attest that SCAT also achieved the highest percentage among comparing tools. The precision values for command execution and code injection vulnerabilities for each application are illustrated in the third and fourth rows, Although SCAT achieved the highest value, there was a considerable drop in the overall percentage values, this due to the absence of these types of vulnerabilities in most of the chosen benchmarks.

Table 4. Precision values for detected vulnerabilities in intra-benchmark tests

Vul. Type	SCAT	Pixy	RIPS	YASCA
XSS	0.804	0.272	0.2578	0.4341
SQL Injection	0.84	0.4107	0.4643	0.1934
Command Injection	0.4286	0.00	0.1238	0.1667
Code Injection	0.2541	0.00	0.0633	0.00

5.2.2. Vulnerability detection in inter-benchmark tests

The results for inter-benchmark tests are grouped in Table 5. The table is divided into three grouped sets or rows; the first column of the table shows the category name, the second column shows the subject name, each subject includes a set of test cases. The number of test cases in each subject is showed in the third column, the rest of columns display the results of true positives (TP tests), false positives (FP tests) and the success percentage of the tool in each subject [30]. The total percentage is calculated by dividing the total number of passed tests by the total number of tests in a given category. The equation of success percentage is shown in (6).

$$success\% = \frac{TP\_Passed + FP\_Passed}{TP\_Tests + FP\_Tests}\% \quad (6)$$

Table 5. Vulnerability detection in inter-benchmark tests

Category	Subject	No. of Tests	SCAT			Pixy			RIPS			YASCA		
			TP	FP	S%	TP	FP	S%	TP	FP	S%	TP	FP	S%
Vulnerability Detection	All	18	17	15	89	7	17	67	12	17	81	4	14	50
	Argument injection	1	0	1	50	0	1	50	0	1	50	0	1	50
	Command Injection	2	2	2	100	0	2	50	2	2	100	2	0	50
	Code injection	2	2	2	100	0	2	50	2	2	100	0	2	50
	SQL injection	6	6	4	83	2	6	67	5	6	92	0	6	50
	Server-side Include	2	2	2	100	2	1	75	2	1	75	0	2	50
	XPath injection	2	2	1	75	0	2	50	1	2	75	2	0	50
	Cross-site Scripting	3	3	3	100	3	3	100	0	3	50	3	3	50
Language Support	All	30	25	16	68	19	17	60	1	29	50	0	30	50
	Aliasing	4	4	4	100	4	0	50	0	4	50	0	4	50
	Arrays	2	2	0	50	2	0	50	0	2	50	0	2	50
	Constants	2	1	1	50	2	1	75	0	2	50	0	2	50
	Functions	5	5	1	60	5	4	90	1	4	50	0	5	50
	Dynamic Inclusion	3	1	1	33	1	1	33	0	3	50	0	3	50
	Object model	8	7	5	75	0	7	44	0	8	50	0	8	50
	Strings	3	3	3	100	3	3	100	0	3	50	0	3	50
	Variable Variables	3	2	1	50	2	1	50	0	3	50	0	3	50
Sanitization Support	All	7	6	3	64	6	3	64	1	7	57	0	7	50
	Regular expressions	2	2	0	50	2	0	50	0	2	50	0	2	50
	SQL injection	1	0	1	50	0	1	50	1	1	100	0	1	50
	Strings	2	2	0	50	2	0	50	0	2	50	0	2	50
	Cross-site Scripting	2	2	2	100	2	2	100	0	2	50	0	2	50

In Vulnerability Detection category, the results show that SCAT detected all vulnerabilities types; except for Argument injection which is not supported by the prototype. On the other hand, RIPS failed to spot XSS and argument injection vulnerabilities, it also failed in one SQL injection test and one XPath test, and YASCA only detected command execution and XPath injection vulnerabilities.

In the results of the Language Support category, SCAT managed to detect the vulnerabilities in object model files, it passes 7 tests out of 8 tests in this subject. This result indicates that the effort spent in order to support object-oriented features in the prototype model was paid off. In Sanitization Support group, the results show that for 86% of TP tests, SCAT was able to detect good sanitization routines when it encounters it. While YASCA failed to pass any of the test cases. The only test in which SCAT failed is SQL

injection sanitization test, in this test an (htmlspecialchars) sanitization routine is used which SCAT considers as a strong sanitization method, so SCAT skips the vulnerability.

In false positive tests of Vulnerability Detected category, Pixy and RIPS remain silent for all the tests. However, false positive tests cannot be considered alone, as an evaluation of tool performance, for example, Pixy passes all tests because it is incapable to detect these vulnerabilities. This the main flaw in false positive tests; they cannot differentiate between a tool that can scan and actually take the decision to skip the resolved vulnerability and another tool that does not detect the vulnerability in the first place. SCAT came in the second place with 83% passing percentage. Table 6 shows the calculated evaluation metrics for each tool in the three categories of inter-benchmark tests In Vulnerability Detection category, Pixy managed to achieve better values in the metrics that weights false positives higher than the true positives, this is because Pixy does not cover these types of vulnerabilities. However, SCAT managed to score the highest value in $F_{measure}$ (4).

Table 6. Metrics evaluation for inter-benchmark tests

Category	Tool	Precision	Recall	Specificity	$F_{Measure}$
Vulnerability Detection	SCAT	0.85	0.94	0.83	0.89
	Pixy	0.88	0.39	0.94	0.54
	RIPS	0.92	0.67	0.94	0.78
	YASCA	0.50	0.22	0.78	0.31
Language Support Detection	SCAT	0.64	0.83	0.53	0.72
	Pixy	0.59	0.63	0.57	0.61
	RIPS	0.50	0.03	0.97	0.06
	YASCA	0.00	0.00	1.00	0.00
Sanitization Support Detection	SCAT	0.60	0.86	0.43	0.71
	Pixy	0.60	0.86	0.43	0.71
	RIPS	1.00	0.14	1.00	0.25
	YASCA	0.00	0.00	1.00	0.00

In the Language Support category, SCAT managed to score the highest value in Precision (1), Recall (2) and $F_{measure}$ metrics (4). In specificity (3) values, RIPS and Pixy managed to achieve better performance because they managed to pass more false positive tests, however, $F_{measure}$ values indicate that SCAT has a better performance. Precision (1), Recall (2) and Specificity (3) evaluation metrics in Sanitization Support category show that SCAT has the highest $F_{measure}$ value, although RIPS achieved higher value in Precision and Specificity metrics.

The results of inter-benchmark tests clearly show that SCAT scores the highest percentage in the true positives tests (recall) of the three categories with 88% detection rate. It also managed to score 94% detection rate in vulnerability detection category in particular which was the highest rate overall comparing tools. Pixy passes these three categories with 63% detection rate, while RIPS only scores 28%, YASCA came in last with 7% detection rate.

Table 7 shows the summary of results for the execution of the four tools against the inter-benchmark tests. The table presents the calculated Recall (1), Precision (2), Specificity (3) and $F_{measure}$ evaluation metrics (4, 5). The results obtained for both types of $F_{measure}$ metric show that SCAT achieved the best values.

Table 7. Evaluation metrics results

Tool	Precision	Recall	Specificity	$F_{Measure}$	$F_{0.5\ Measure}$	$F_{2\ Measure}$
SCAT	0.69	0.88	0.59	0.77	0.721	0.834
Pixy	0.69	0.63	0.64	0.62	0.677	0.641
RIPS	0.81	0.28	0.97	0.42	0.587	0.322
YASCA	0.16	0.07	0.93	0.10	0.127	0.079

6. CONCLUSION

Web applications present a major role in almost all the principal services in the daily life. However, vulnerabilities that threaten the personal data of users are discovered frequently. Therefore, this paper proposed an automated server-side model for the dynamic recognition and justification of a wide range of taint-style attacks. The proposed model is able to overcome most of the challenges in securing scripting languages like PHP. The model was implemented in a prototype called (SCAT), which performs several types of analysis to detect security vulnerabilities in the input program.

The proposed model performs a flow-sensitive, inter-procedural and context-sensitive data flow analysis in order to collect information about the program execution. Then, the model uses the information collected in the data flow analysis phase to detect security vulnerabilities such as XSS and SQL injection. Finally, it generates a detailed report which contains a detailed explanation of each sensitive sink that represents security vulnerability in the program.

To evaluate the proposed system, an empirical evaluation procedure is conducted in which the proposed prototype SCAT analyzes several real-world applications and categorizes sets of testing benchmarks. The results demonstrate that the proposed system managed to detect 94% (recall value) of security vulnerabilities found in the testing benchmarks which is the highest detection rate compared to other systems. This clearly indicates the accuracy and robustness of SCAT. The evaluation process assesses the compatibility of SCAT with PHP features, the prototype managed to achieve the highest score by 83%, which is higher than Pixy that came in second place with only 64%. As a result, SCAT provides an effective solution to complicated web systems by offering the benefits of securing private data for users and maintaining web application stability for web applications providers.

REFERENCES

[1] A. Breeva and E. Potseluevskaya, "Most Vulnerable Web Application in 2013: XSS, PHP and Media Sites," *Positive Technology*, 2014.

[2] S. Gupta and B. B. Gupta, "Cross-Site Scripting (XSS) attacks and defense mechanisms: classification and state-of-the-art," *International Journal of System Assurance Engineering and Management*, vol. 8, pp. 512-530, 2017.

[3] M. K. Gupta, et al., "Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey," *International Conference on Recent Advances and Innovations in Engineering (ICRAIE)*, pp. 1-5, 2014.

[4] M. I. P. Salas, et al., "Security Testing Methodology for Evaluation of Web Services Robustness-Case: XML Injection," *2015 IEEE World Congress on Services*, pp. 303-310, 2015.

[5] K. G. Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Information and Software Technology*, pp. 18-33, 2015.

[6] M. Cova, et al., "Vulnerability analysis of web-based applications," *Test and Analysis of Web Services*, pp. 363-394, 2007.

[7] N. Jovanovic, et al., "Pixy: A static analysis tool for detecting web application vulnerabilities," *2006 IEEE Symposium on Security and Privacy*, pp. 6-263, 2006.

[8] F. Yu, et al., "Stranger: An automata-based string analysis tool for PHP,", "Tools and Algorithms for the Construction and Analysis of Systems," *Lecture Notes in Computer Science*, vol. 6015, pp. 154-157, 2010.

[9] D. Balzarotti, et al., "Saner: Composing static and dynamic analysis to validate sanitization in web applications," *2008. IEEE Symposium on Security and Privacy*, pp. 387-401, 2008.

[10] J. Dahse and T. Holz, "Simulation of Built-in PHP Features for Precise Static Code Analysis," *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[11] R. F. A. Kader, et al., "A Static Code Analysis Tool for Security Vulnerabilities in PHP 5 Scripts," *The 2012 World Conference on Computer Science and Information Technology*, 2012.

[12] G. Klein, et al., *Jflex-the fast scanner generator for java*, JFlex User's Manual, July 2005, [Online]. Available: http://www.jflex.de. [Accessed on: July 2015].

[13] S. E. Hudson, et al., "Cup parser generator for java," Princeton University, 1999.

[14] E. Goktas, et al., "Out of control: Overcoming control-flow integrity," *IEEE Symposium on Security and Privacy*, pp. 575-589, 2014.

[15] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," *Proceedings of the 15th conference on USENIX Security Symposium*, vol. 6, pp. 179-192, 2006.

[16] A. Rimsa, et al., "Efficient static checker for tainted variable attacks," *Science of Computer Programming*, vol. 80, pp. 91-105, 2014.

[17] L. K. Shar and H. B. K. Tan, "Auditing the defense against cross site scripting in web applications," *Proceedings of the 2010 International Conference on Security and Cryptography (SECRYPT)*, pp. 1-7, 2010.

[18] A. Rimsa, et al., "Tainted flow analysis on e-SSA-form programs," *International Conference on Compiler Construction*, pp. 124-143, 2011.

[19] J. Ellson, et al., "Graphviz and dynagraph-static and dynamic graph drawing tools," *Graph Drawing Software*, pp. 127-148, 2004.

[20] "PHP Official Documentation," *The PHP Group*, [Online]. Available: https://www.php.net/manual/en/.

[21] B. Mburano and W. Si, "Evaluation of Web Vulnerability Scanners Based on OWASP Benchmark," *26th International Conference on Systems Engineering*, pp. 1-6, 2018.

[22] R. M. Parizi, et al., "Benchmark Requirements for Assessing Software Security Vulnerability Testing Tools," *IEEE 42nd Annual Computer Software and Applications Conference*, pp. 825-826, 2018.

[23] N. L. de Peol, "Automated security review of PHP web applications with static code analysis," Master's thesis, 2010.

[24] G. Chatzieleftheriou and P. Katsaros, "Test-driving static analysis tools in search of C code vulnerabilities," *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW)*, pp. 96-103, 2011.
[25] G. Díaz and J. R. Bermejo, "Static analysis of source code security: assessment of tools against SAMATE tests," *Information and Software Technology*, vol. 55, no. 8, pp. 1462-1476, 2013.
[26] U. Sarmah, et al., "A survey of detection methods for XSS attacks," *Journal of Network and Computer Applications*, vol. 118, pp. 113-143, Sep. 2018.
[27] M. A. Kausar, et al., "SQL injection Detection and Prevention Techniques in ASP.NET Web Applications," *International Journal of Recent Technology and Engineering (IJRTE)*, vol. 8, no. 3, pp. 7759-7766, Sep. 2019.
[28] A. Kieyzun, et al., "Automatic creation of SQL injection and cross-site scripting attacks," *IEEE 31st International Conference on Software Engineering (ICSE 2009)*, pp. 199-209, 2009.
[29] T. Scholte, et al., "Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis," *2012 IEEE 36th Annual Computer Software and Applications Conference (COMPSAC)*, pp. 233-243, 2012.
[30] Y. Fang, et al., "TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology," *PLoS ONE*, vol. 14, no. 11, pp. e0225196, 2019.
[31] A. A. Neto and M. Vieira, "Trustworthiness Benchmarking of Web Applications Using Static Code Analysis," *2011 Sixth International Conference on Availability, Reliability and Security (ARES)*, pp. 224-229, 2011.

BIOGRAPHIES OF AUTHORS

Rabab Farouk Abdel-Kader she received her B.S. from the Electrical Engineering Department Suez Canal University in 1998. She received her Ph.D. degree from the department of Computer Science and Software Engineering at Auburn University, Auburn, AL in 2007 and the MS degree in Electrical Engineering from Tuskegee University with high honors in 2002. Since 2008 she is working as an Assistant Professor in the Electrical Engineering department, Faculty of Engineering, Port-Said University, Egypt. Her main research interests include image processing, parallel computing, and software Engineering.

Mona Nashaat, **s**he received her B.S. degree in Electrical Engineering, Computer and Control division from Faculty of Engineering Suez Canal University, Port-Said, Egypt in 2008 and her M.S. degree from Electrical Engineering, Computer and Control division, Faculty of Engineering Port-Said University, Port-Said, Egypt in 2013. Currently she is a demonstrator at the Electrical Engineering Department, Computer and Control division, Port-Said University, Egypt. Since 2016 she is perusing her PhD degree from the University of Alberta, Canada. Her Main research line concerns include application line security, Programming Language Theory, and Software Engineering.

Mohamed Ibrahim Habib received the B.S. degree in Electrical Engineering Computer and Control from Suez Canal University, Port Said, Egypt, in 2000, and M.S. and Ph.D. degrees in Computer Engineering from Suez Canal University, in 2004 and 2009, respectively. Since 2009 he has been assistant professor in the Department of Computer Engineering, Suez Canal University. From 2014, he is a visiting assistant professor in Saudi Electronic University. His major research interests are Data Mining, Machine Learning, Deep Learning, and Artificial Intelligence.

Hani Mohamed Kamal Mahdi is an IEEE Life Senior Member. He was awarded in 2015 the Distinguished University Award of Ain Shams University, Cairo, Egypt. Hani Mahdi is a Computer Systems Professor, Computer and Systems Engineering Department, Faculty of Engineering, Ain Shams University, Cairo, Egypt. From this university he is graduated in 1971 and got his M.Sc. in Electrical Engineering in 1976. He got his Doctor from Technische Universitaet, Braunschweig, West Germany, in 1984. He was a Post-Doctoral Research Fellow at the Electrical and Computer Engineering Department, The Pennsylvania State University, Pennsylvania (1988-1989), and at the Computer Vision and Image Processing (CVIP) Lab, Electrical Engineering Department, University of Louisville, Kentucky (2001-2002). He was on a leave of absence to work with Al-Isra University (Amman, Jordan), El-Emarat University (El Ain, United Arab Emirates), and Technology Collage (Hufuf, Saudi Arabia).Prof. Mahdi was the Head of the Computer and Systems Engineering Department (2008-2009), the Director of Information Network (2006-2008) in the Faculty of Engineering, Ain Shams University. He was the Director of Ain Shams University - Information Network (2007-2008). His main research line concerns the Computational Intelligence (Artificial Intelligence) and its applications to pattern recognition, software engineering, data mining, and computer communication. He supervised many software projects in Egypt as a consultant for governmental establishments and private companies. Prof. Mahdi is the Head of conference committees of the International Conference on Electrical, Electronic and Computer Engineering ICEEC'09.