Software Qual J (2018) 26:455–489
DOI 10.1007/s11219-017-9359-5

Estimating software robustness in relation to input validation vulnerabilities using Bayesian networks

Ekincan Ufuktepe[1] **· Tugkan Tuglular**[1]

Published online: 28 March 2017

Abstract Estimating the robustness of software in the presence of invalid inputs has long been a challenging task owing to the fact that developers usually fail to take the necessary action to validate inputs during the design and implementation of software. We propose a method for estimating the robustness of software in relation to input validation vulnerabilities using Bayesian networks. The proposed method runs on all program functions and/or methods. It calculates a robustness value using information on the existence of input validation code in the functions and utilizing common weakness scores of known input validation vulnerabilities. In the case study, ten well-known software libraries implemented in the JavaScript language, which are chosen because of their increasing popularity among software developers, are evaluated. Using our method, software development teams can track changes made to software to deal with invalid inputs.

Keywords Robustness · Input validation vulnerabilities · Bayesian networks

1 Introduction

Robustness is a quality attribute, which is defined by the IEEE standard glossary of software engineering terminology (1990) as the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. Usually, robustness is considered under the dependability quality attribute of software systems (Shahrokni and Feldt 2013). For instance, Avizienis et al. (2001) define robustness as dependability with respect to erroneous input. With buffer overflow and SQL injection attacks, developers and software companies have become more considerate of robustness. A buffer overflow attack occurs when too much data copied into a fixed size buffer, causes the data to

✉ Tugkan Tuglular
tugkantuglular@iyte.edu.tr

Ekincan Ufuktepe
ekincanufuktepe@iyte.edu.tr

1 Department of Computer Engineering, Izmir Institute of Technology, Izmir, Turkey

overwrite into adjacent memory locations, and may affect the behavior of the program (Kuperman et al. 2005). SQL injection attacks allow attackers to obtain access to the databases used by the applications, where input is treated as SQL code, and if not validated, the SQL code can be submitted to the database (Halfond et al. 2006). Both buffer overflow and SQL injection attacks are data attacks and may be successful if validation is not applied to the entered data.

Existing static analysis tools do not provide any robustness estimation. Therefore, companies try to address software robustness through testing, for which there are tools available. These tools do not provide an estimation for robustness, but instead provide success rate with respect to the test suite used.

Quality estimations play an important role in evaluating software as a product. One use of quality estimations is to provide a snapshot of software product at time t and to use the result to decide whether to continue improving the product or not. Another use is to compare two versions of the same product to see how the changes affected the software product. In this paper, we propose a method for estimating the robustness of the software under consideration against invalid input.

Input to software should be validated where it is entered (Jourdan 2008). Otherwise, entered data may exploit existing input validation vulnerabilities in software. Input validation vulnerability can be defined as vulnerable parts of the system receiving unintended input, which may result in altered control flow, arbitrary control of a resource, or arbitrary code execution (http://cwe.mitre.org/). Various input validation vulnerabilities, such as SQL injection and Cross-Site Scripting, have been identified as some of the most dangerous encountered vulnerabilities by the Open Web Application Security Project (OWASP) (Smithline 2013) and National Vulnerability Database (NVD 2014). These vulnerabilities are introduced during the design and implementation phases of the software development lifecycle (SDLC), when the developer fails to design, code, and test for validation of inputs. NVD only provides report for each single vulnerability. In this work, input validation-related vulnerability reports are collected one by one from NVD (NVD 2014). Our calculations have shown that the number of input validation vulnerabilities identified and published between Jan. 2000 and Jan. 2015 is 17,317.

The following six input validation vulnerabilities defined by Common Weakness Enumeration (CWE https://cwe.mitre.org/) are in the scope of this work:

- CWE-22: Improper limitation of a pathname to a restricted directory (“Path Traversal”)
- CWE-78: Improper neutralization of special elements used in an OS command (“OS Command Injection”)
- CWE-79: Improper neutralization of input during Web page generation (“Cross-site Scripting”)
- CWE-89: Improper neutralization of special elements used in an SQL command (“SQL Injection”)
- CWE-120: Buffer copy without checking size of input (“Classic Buffer Overflow”)
- CWE-134: Uncontrolled format string

Their descriptions are given in Appendix 1. Although there are more input validation vulnerabilities, these are chosen because Common Weakness Scoring System (CWSS https://cwe.mitre.org/cwss/) considers them as effective vulnerabilities, meaning widely spread and having considerate impact if exploited, and presents calculated vulnerability scores for them.

An application with client-side and/or application side JavaScript code can have data input that may flow to a Web site and/or a database. The possibility of such a data flow and at the same time the possibility of data being an invalid data are the motivation behind this work. Due to this probabilistic nature of applications including libraries and their environments, we propose a robustness estimation based on Bayesian inference. When the proposed method calculates a robustness estimation for an application and/or library, it performs static analysis on the source of the software under consideration without checking its connection to a Web site and/or to a database. The reason is that our method is designed with idea of enabling developers to check even partially developed code for robustness. With current agile software development practices, even the developer might not know whether the code written will have a Web site and/or database connection.

We follow risk-based approach, since our idea of robustness measure stems from vulnerabilities and their impact. Fenton and Neil (2012) state that Bayesian networks are useful for risk assessment. Moreover, fault trees, which are similar to our input validation vulnerabilities tree, are used in Bayesian network construction (Bobbio et al. 2001).

We propose a method for the estimation of robustness of a given software. The proposed method employs an intermediate language for input validation called Input Validation Language for Robustness (IVL4R), a static analyzer that utilizes IVL4R, and a Bayesian network (BN) stemmed from input validation vulnerabilities utilizing accumulated data about vulnerabilities for robustness estimation. The novelty of the proposed method is as follows:

- The proposed static analyzer checks the source code for input validation code that checks validity of inputs, which can exploit input validation vulnerabilities. The proposed static analyzer does not look for input validation vulnerabilities. However, the static analyzers that accept rules from outside can be configured to act like our proposed checker for input validation code.
- A Bayesian network constructed using input validation vulnerability tree is used to calculate weakness of given software with respect to robustness. To the extent of our knowledge, our proposal to estimate robustness using a Bayesian network is new. Its novelty lies in employing accumulated data about vulnerabilities and analyzing results of all functions of given software with respect to IVL4R.
- Our proposed approach does not search for input validation vulnerabilities. Instead, it checks existence of input validation code that possibly prevents exploitation of input validation vulnerabilities. The assumption here is that input validation vulnerabilities exist in software. A vulnerability that is not known at the time being can be discovered later, but it may have been residing in the code since the date of development.

The result of calculation gives software developers an estimation that lies between 0 and 1 and being close to 1 means software is in better shape in terms of robustness. This way, software developers can track changes with respect to robustness between different versions of software. Moreover, to be able to estimate robustness will result in better awareness and goal setting for robustness. Since our proposed BN utilizes accumulated data about vulnerabilities (starting from 2000 to the current time of calculation), which includes not only vulnerability distributions but also their impacts, the measurement also indicates the software's relative risk with respect to robustness under currently (at the time of calculation) known circumstances.

A "Cross-site Scripting" vulnerability existed in jQuery up until version 1.6.2 (2011) but patched in the later versions. Somehow, it re-appeared in version 1.9.1 (http://blog.mindedsecurity.com/search/label/jQuery 2013). According to MindedSecurity, when there is a call to jQuery function with an argument, such as jQuery (location. hash), the jQuery, or its alias "$," method tries to understand if the argument contains some tags. It means that if the arguments of jQuery function contain some kind of tag, it will be rendered using innerHTML, resulting in a potential DOM Based Cross-site Scripting attack. Other input validation vulnerability examples are given in Appendix 2. Therefore, as running example, we prefer to work on jQuery. For case study, we applied our proposed method to ten well-known software libraries including jQuery implemented in the JavaScript language. We selected both client-side and server-side JavaScript libraries.

The paper is organized as follows. Section 2 gives a summary of related work. Section 3 explains the process of detecting the existence of input validation code in the source code using IVL4R. Section 4 introduces our method for constructing a BN to estimate software robustness in relation to input validation vulnerabilities. Section 5 presents the case study. Section 6 explains the tools used in this work. Section 7 concludes the paper.

2 Related work

Bayesian networks are also known as belief networks. They are graphical representation of interactions between causes and effects. Bayesian network is a directed acyclic graphical model, of which nodes are encoded with probabilistic relationships. They are used for diagnostic purposes or for predictions if evidence is supplied. Without any evidence supplied, Bayesian networks provide structured probability information between causes and effects. "In particular, each node in the graph represents a random variable, while the edges between the nodes represent probabilistic dependencies among the corresponding random variables. These conditional dependencies in the graph are often estimated by using known statistical and computational methods" (Ben-Gal 2007).

Frigault and Wang (2008) explored the causal relationships between vulnerabilities encoded in an attack graph. However, the evolving nature of vulnerabilities and networks was largely ignored. They proposed a dynamic Bayesian network (DBN)-based model to incorporate temporal factors, such as the availability of exploit codes or patches. Starting from the model, they studied two concrete cases to demonstrate potential applications. This novel model provides a theoretical foundation and a practical framework for continuously measuring network security in a dynamic environment.

Kondakci (2010) proposed a network security risk assessment model using Bayesian belief networks (BBNs). He introduced a generic threat model that can also be applied to risk computation of various types of IT assets and dependable computing environments. He modeled the classification of information security threats (human-related, internal, and external) as a compound structure with four dependable parameters. He also developed a new risk propagation model using the conditional probability method and average score scheme, by which risk levels can easily be estimated and quantified for different assessment systems. The reason for using a BN is given as its ability to represent knowledge and develop automated reasoning systems. Traditional inference methods are difficult to apply in determining posterior distributions of risk factors in dynamically changing IT environments.

Therefore, dependence analysis in large-scale networks can easily be performed by applying Bayesian approaches.

Wagner (2010) used BNs to assess and predict software quality by using activity-based quality models. To construct the BN, Wagner defined three types of nodes and followed four steps. The first step involves creating a goal-based derivation of relevant activities and their indicators, while the facts and sub-activities are identified in the second step. The third step identifies suitable indicators for the facts. In the fourth step, node probability tables are defined to show quantitative relationships. The nodes are created by a map of the "Situations and Activities" of the software. The "Activities" map holds information that has an influence on an activity, i.e., anything that is done with the system. For example, *maintenance* and *use* are high-level activities. The "Situations" map contains, for example, the *system*, its *environment*, and the development *organization*. For each situation that has a positive or negative effect on an activity, a node is added to the facts. If the situation does not have any effect on the activity, no node is added.

Guarnieri and Livshits (2010) stated that static analysis is a useful technique in a variety of applications, ranging from program optimization to bug finding. Their solution utilizes staged static analysis as a means to analyze streaming JavaScript programs. They suggested use of combined offline-online static analysis as a way to accomplish fast, online analysis at the expense of a more thorough and costly offline analysis of the static code. The offline stage may be performed on a server ahead of time, whereas the online analysis is typically integrated into the Web browser. Through a wide range of experiments on both synthetic and real-life JavaScript code, they found that in normal use, where updates to the code are small, they could update static analysis results within the browser fast enough to be acceptable for everyday use. They demonstrated the advantages of this kind of a staged analysis approach in a wide variety of settings, especially in the context of mobile devices.

Jensen et al. (2009) presented a static analysis program for JavaScript called TAJS that performs type analysis on JavaScript code. The type analysis is performed on a lattice of "values" and from transfer functions that have been derived from a data flow analysis and flow graph. For example, if a function is called with a "*toString()*" function, they inferred the data type of "string." In this work, they claimed that their type analyzer is the first sound and detailed tool for JavaScript code. The use of a monotone framework with an elaborate lattice structure, combined with recency abstraction, results in an analysis that achieves good precision on demanding benchmarks.

Franke et al. (2011) described Bayesian decision support model, designed to help enterprise IT system decision-makers to evaluate the consequences of their decisions by analyzing various scenarios. Their model is based on expert elicitation from multiple experts on IT systems availability. They have obtained this information through an electronic survey. The Bayesian model they have proposed uses a leaky Noisy-OR method to weigh together the expert opinions on 16 factors affecting systems availability. Using this model, the effect of changes to a system is estimated prior, providing decision support for improvement of enterprise IT systems availability. The Bayesian model thus obtained is then integrated within a standard, reliability block diagram-style, mathematical model for assessing availability on the architecture level. The IT systems play the role of building blocks in their Bayesian model. The overall assessment framework thus addresses measures to ensure high availability both on the level of individual systems and on the level of the entire enterprise architecture.

Okutan and Yıldız (2012) mentioned that there are lots of different software metrics discovered and used for defect prediction. They have proposed a practical and easy approach through which they could determine the set of metrics that are most important and focus on them more to predict defectiveness, instead of dealing with many metrics. They have used Bayesian networks to determine the probabilistic influential relationships among software metrics and defect proneness. Nevertheless to the metrics they used in Promise data repository, they have defined two more metrics, i.e., number of developers (NOD) and lack of coding quality (LOCQ) for the source code quality. They have extracted these metrics by inspecting the source code repositories of the selected Promise data repository data sets. At the end of their modeling, they have learned the marginal defect proneness probability of the whole software system, the set of most effective metrics, and the influential relationships among metrics and defectiveness. Their experiments on nine open source Promise data repository data sets have shown them that response for class (RFC), lines of code (LOC), and LOCQ are the most effective metrics, while coupling between objects (CBO), weighted method per class (WMC), and lack of cohesion of methods (LCOM) are less effective metrics on defect proneness. In addition, the number of children (NOC) and the depth of inheritance tree (DIT) have shown them that they have very limited effect and are untrustworthy. On the other hand, with their experiments on Poi, Tomcat, and Xalan data sets, they have observed that there is a positive correlation between the NOD and the level of defectiveness.

Weber et al. (2012) presented a bibliographical review over the last decade on the application of Bayesian networks to dependability, risk analysis, and maintenance. They have shown an increasing trend of the literature related to these domains. This trend is because of the benefits that Bayesian networks provide in contrast with other classical methods of dependability analysis for instance: Markov Chains, Fault Trees, and Petri Nets, etc. Some of the benefits of these methods are that they have the capability to model complex systems, make predictions and diagnostics, compute the exact occurrence probability of an event, update the calculations according to evidences, represent multi-modal variables, and help modeling user-friendly via a graphical and compact approach. Their review is based on an extraction of 200 specific references in dependability, risk analysis, and maintenance applications among a database with 7000 Bayesian network references.

Dejaeger et al. (2013) indicated that software testing is a crucial activity during software development and fault prediction models assist practitioners herein by providing an upfront identification of faulty software code by drawing upon the machine learning literature. While especially the Naive Bayes classifier is often applied in this regard, citing predictive performance and comprehensibility as its major strengths, a number of alternative Bayesian algorithms that boost the possibility of constructing simpler networks with fewer nodes and arcs remain unexplored. Their study contributes to the literature by considering 15 different Bayesian network (BN) classifiers and comparing them to other popular machine learning techniques. In addition, they have investigated the applicability of the Markov blanket principle for feature selection, which is a natural extension to BN theory. They have tested results using the statistical framework of Demšar, both in terms of the area under the ROC curve (AUC) and the recently introduced H-measure. In conclusion, the simple and comprehensible networks with less nodes can be constructed using BN classifiers other than the Naive Bayes classifier. Furthermore, they have found that the development context is an item that should be taken into account during model selection and the aspects of comprehensibility and predictive performance need to be balanced out.

Holm et al. (2014) has mentioned the importance of managing software vulnerabilities with publicly available exploits for both developers and users. However, this is a difficult matter to address as time is limited and vulnerabilities are frequent. Therefore, Holm et al. (2014)

presented a Bayesian network-based model that can be used by enterprise decision makers to estimate the likelihood that a professional penetration tester is able that obtains knowledge of critical vulnerabilities and exploits for these vulnerabilities for software under different circumstances. In their approach, they have gathered the data on the activities in the model from the previous empirical studies, vulnerability databases, and a survey with 58 individuals who all have been credited for the discovery of critical software vulnerabilities. The model they have proposed described 13 states related by 17 activities, and a total of 33 different datasets. In conclusion, the model they have proposed can be used to support decisions regarding what software to acquire or what measures to invest in during software development projects.

Perkusich et al. (2015) have mentioned that there are several software process models and methodologies such as waterfall, spiral, and agile. However, the rate of successful software development projects is low. The software is the major output of software processes, and through this inference, they describe that by increasing software process, management quality should increase the project's chances of success. In addition, organizations have invested to adapt software processes to their environments and the characteristics of projects to improve the productivity and quality of the products. Due to these problems, Perkusich et al. (2015) presented a procedure to detect problems of processes in software development projects by using Bayesian networks. Their procedure was successfully applied to Scrum-based software development projects. The Bayesian network was successfully validated through simulated scenarios, and their procedure was successfully validated in two Scrum-based software development projects.

3 Validation of input data

Every input should be validated before use. This is similar to the rule stating that "every variable should be defined before it is used." This rule introduced define-use (*du*) pairs and enabled their detection using static analysis. If a define-use pair is not found for a variable by static analyzer, an anomaly flag is raised. Similarly, by our proposal, we extend define-use pairs to define-validate-use (*dvu*) triples. It follows that if a define-validate-use triple is not found for an input (variable), then an anomaly flag should be raised.

3.1 Input validation language for robustness

In this work, we consider arguments to the function as inputs of the function. Argument definition is the define part of the define-validate-use triple, and before use, there should be a validation part, which is usually described by precondition(s). Preconditions can be implemented in various ways, such as using if, switch, while, and for statements as well as assert and try-catch structures. Since precondition implementation is language dependent, it is better to use an intermediate language to represent precondition implementation, i.e., input validation. We extended intermediate representation for input validation and sanitization developed by Alkhalaf (2014) to input validation language for robustness by including preconditions to be checked in the input so that it is possible to prevent exploitation of input validation vulnerabilities, assuming that they exist in software. The following Table 1 shows the proposed preconditions. The *length* function returns the length of input. The *indexof* function checks if the given argument exists in the input and returns −1 if does not exist and the location if exists. In the intermediate representation, all the inputs are considered as string.

Table 1 Preconditions able to prevent input validation vulnerabilities from being exploited

Input validation vulnerability	Precondition
CWE-22: Improper limitation of a pathname to a restricted directory ("Path Traversal")	indexof(Var , "../") indexof (Var , "..\")
CWE-78: Improper neutralization of special elements used in an OS command ("OS Command Injection")	indexof (Var , ".exe") indexof (Var , "/bin/")
CWE-79: Improper neutralization of input during Web page generation ("Cross-site Scripting")	indexof (Var , "<script>")
CWE-89: Improper neutralization of special elements used in an SQL command ("SQL Injection")	indexof (Var , " OR ") indexof (Var , " AND ") indexof (Var , " IS NULL ")
CWE-120: Buffer copy without checking size of input ("Classic Buffer Overflow")	length(Var)
CWE-134: Uncontrolled format string	indexof (Var , "%")

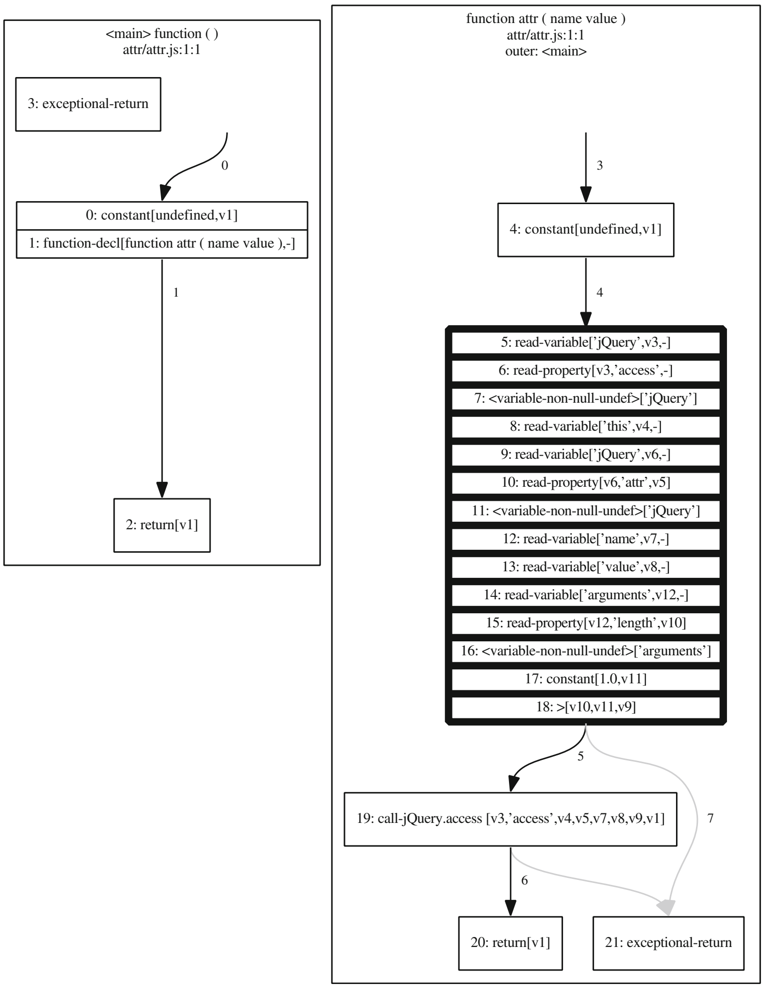
The BNF representation of IVL4R is as follows:

Function → function(Var [, Var]){ Block }*
Var → <identifier>
*Block → Stmt [; Stmt]**
Stmt → Var := Exp | accept | reject
    | if (Pred) { Block }[else { Block }]
Exp → "String" | Var | ValFunc
Pred → Pred && Pred | Pred || Pred | !Pred | (Pred)
    | ValFunc RelOp <integer>
RelOp → < | <= | > | >= | == | !=
ValFunc → indexof(Var, "String")
    | length(Var)
Integer → Digit | Integer Digit
String → Character | String Character
Digit → 0 | 1 | ... | 9
Character → 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z' | '%' | '/' | '\' | '<' | '>' | ... | '.'

Reject statement corresponds to negative exit, meaning that if input variable is not validated, the function exits. Accept statement indicates positive exit, meaning that input variable is validated. ValFunc stands for validation functionality with respect to the chosen input validation vulnerabilities. Each function in the software under consideration is transformed to this intermediate language and then evaluated with respect to preconditions given in Table 1. The evaluation results are fed to Bayesian network.

3.2 Implementation of input validation language for robustness

We selected jQuery's version 1.9.1 library as the running example. jQuery is a frequently used library that is preferred by Web application developers because it is fast, small, and feature-rich. It contains many functions that help developers handle events, perform manipulations, HTML



<main> function ( )
attr/attr.js:1:1
3: exceptional-return
0
0: constant[undefined,v1]
1: function-decl[function attr (name value),-]
1
2: return[v1]
function attr (name value)
attr/attr.js:1:1
outer: <main>
3
4: constant[undefined,v1]
4
5: read-variable['jQuery',v3,-]
6: read-property[v3,'access',-]
7: <variable-non-null-undef>['jQuery']
8: read-variable['this',v4,-]
9: read-variable['jQuery',v6,-]
10: read-property[v6,'attr',v5]
11: <variable-non-null-undef>['jQuery']
12: read-variable['name',v7,-]
13: read-variable['value',v8,-]
14: read-variable['arguments',v12,-]
15: read-property[v12,'length',v10]
16: <variable-non-null-undef>['arguments']
17: constant[1.0,v11]
18: >[v10,v11,v9]
5
19: call-jQuery.access [v3,'access',v4,v5,v7,v8,v9,v1]
7
6
20: return[v1]
21: exceptional-return

Fig. 1 Output of TAJS's extended flow graph on detecting missing input validation code with respect to input validation vulnerabilities

document traversal, and animation. Furthermore, it is used by many popular Web sites such as Amazon, Microsoft, WordPress, Reddit, Instagram, Stack Overflow, the Guardian, Fox News, and many others. We focus on three main steps in collecting information about functions.

For all the functions, we check the function parameters to see whether they are validated before use. To ascertain whether parameters are validated, we use TAJS's (Jensen et al. 2009) flow graph feature and extend TAJS to detect automatically whether there is a validation block in the function. With our extensions to TAJS's flow graph feature, we can automatically detect whether a function's parameters are validated before use. We differentiate validated and not validated parameters in the flow graph.

On the right-hand side of Fig. 1, we see the output of our TAJS extension. The bold thick frames denote that function *attr()*'s parameters "name" and "value" are used before they are validated or the function does not include any validation block. To realize this process, we initially constructed a tree of the source code consisting of four levels (see Fig. 2). The first

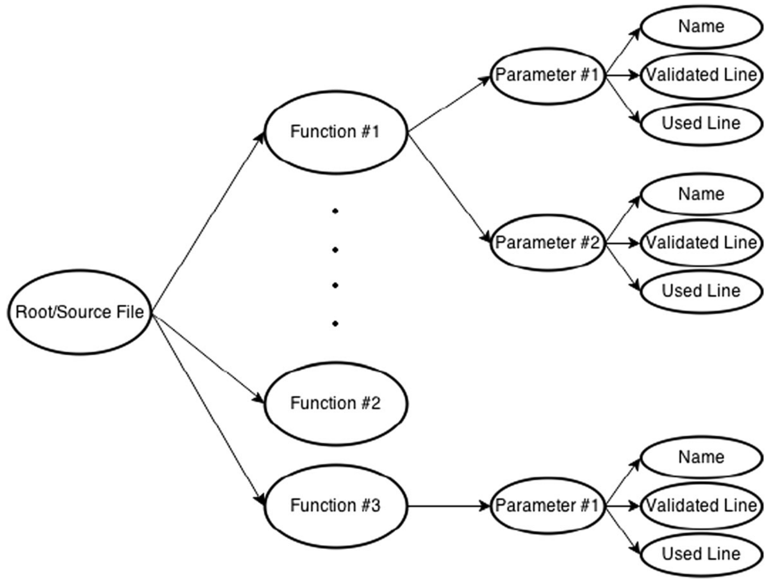


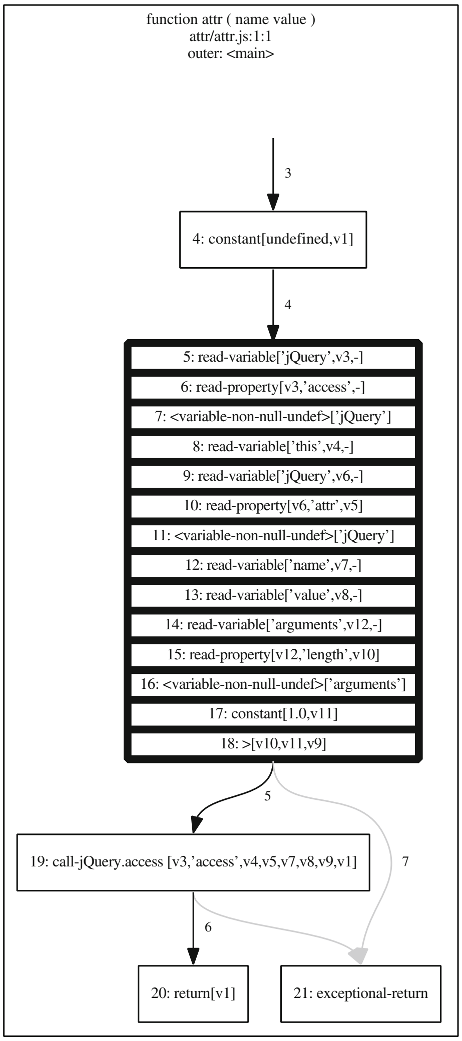
Fig. 2 Tree of function and parameter information

level represents the root, which is also the root of the source file. In the second level, we have the functions defined in the source code. The third level contains the parameters of the functions. The fourth and final level consists of parameter information containing three main components: parameter name, first used line, and validated line, which are indicated by line numbers in TAJS flow graph.

In Fig. 3, we outline the algorithm using the example of how to detect a parameter that is a potential input validation vulnerability through the flow graph of the *attr()* function extracted from TAJS. Before we check whether an input validation is performed, it is important to note that we consider every parameter of the function as an input. Then, we check if the parameters are validated before they are used. Having constructed our tree of function and parameter information, we already have the "*validated line*" and "*used line*" attribute values as assigned parameters after analyzing the function's flow graph. Therefore, we define the constraint for invalidated data as "*validated line > used line*." Parameters that satisfy the constraint "*validated line < used line*" are assumed validated input. In Fig. 3, we can see two bold thick frames, denoting the function's parameters. In the bold thick frames, we see that a read operation is performed on the parameters, which means that the function starts using the parameters at that point. Therefore, we check the code above the bold thick frames to see whether the parameters are validated. Using the flow graph, we do not find any validations before the variables are used. In Fig. 4, we see another version of the same function, but with validated parameters/inputs. Once again, we see that the parameters are used in both bold thick frames and dotted thick frames. However, in the dotted thick frame, the parameter is read to validate the input in a simple "if-block." If the parameter does not match the constraint, it is directed to another situation to prevent any harm to the software.

Current scope of "Input Validation Language for Robustness" does not cover float numbers. The scope of "Input Validation Language for Robustness" is determined by the selection of input validation vulnerabilities, which is 6 for the current design and implementation. When another input validation vulnerability is decided to be included to the scope, first "Input Validation Language for Robustness" must be extended along with its implementation if

Fig. 3 Example flow graph of a function parameter without validation

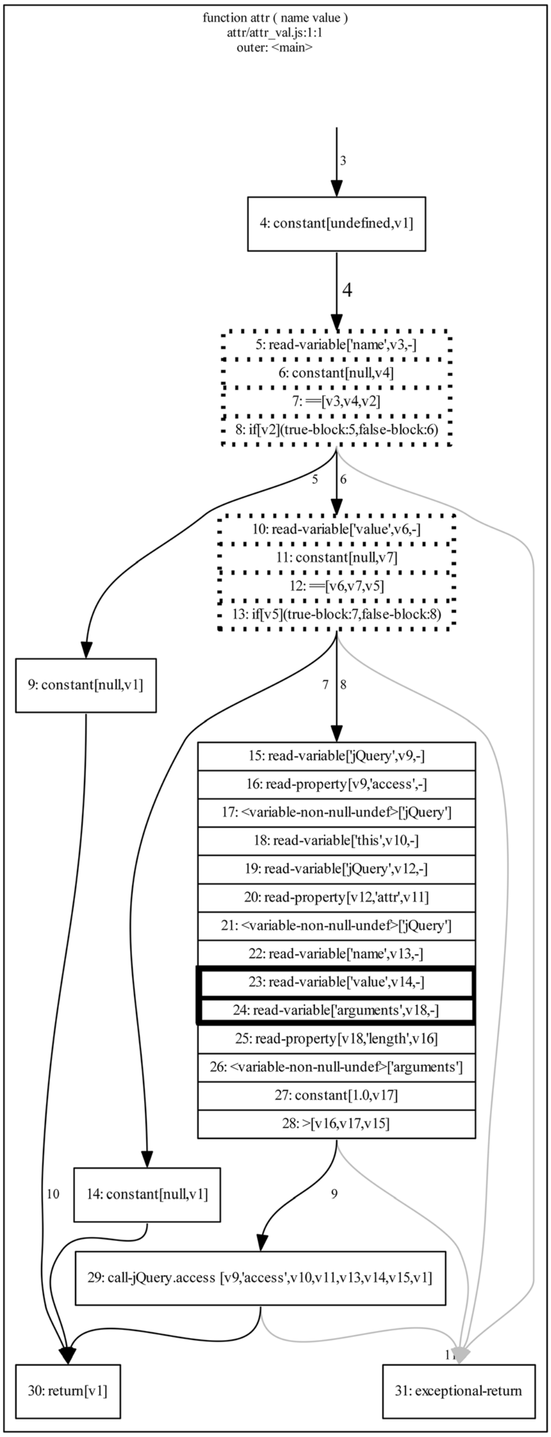

function attr (name value)
attr/attr.js:1:1
outer: <main>
3
4: constant[undefined,v1]
4
5: read-variable['jQuery',v3,-]
6: read-property[v3,'access',-]
7: <variable-non-null-undef>['jQuery']
8: read-variable['this',v4,-]
9: read-variable['jQuery',v6,-]
10: read-property[v6,'attr',v5]
11: <variable-non-null-undef>['jQuery']
12: read-variable['name',v7,-]
13: read-variable['value',v8,-]
14: read-variable['arguments',v12,-]
15: read-property[v12,'length',v10]
16: <variable-non-null-undef>['arguments']
17: constant[1.0,v11]
18: >[v10,v11,v9]
5
7
19: call-jQuery.access [v3,'access',v4,v5,v7,v8,v9,v1]
6
20: return[v1]
21: exceptional-return

necessary and related data for weight and score of this new vulnerability must be collected and added to the Bayesian network.

4 Bayesian network construction

The reason we used a BN as the basis of our methodology is that BNs offer consistent semantics for representing uncertainty and an intuitive graphical representation of interactions between various causes and their effects. BNs are useful when information about the past and/ or the current situation is vague, incomplete, conflicting, or uncertain (Heckerman 1996). The

Fig. 4 Example flow graph of a function parameter with validation


function attr (name value)
attr/attr_val.js:1:1
outer: <main>
3
4: constant[undefined,v1]
4
5: read-variable['name',v3,-]
6: constant[null,v4]
7: ==[v3,v4,v2]
8: if[v2](true-block:5,false-block:6)
5
6
10: read-variable['value',v6,-]
11: constant[null,v7]
12: ==[v6,v7,v5]
13: if[v5](true-block:7,false-block:8)
9: constant[null,v1]
7
8
15: read-variable['jQuery',v9,-]
16: read-property[v9,'access',-]
17: <variable-non-null-undef>['jQuery']
18: read-variable['this',v10,-]
19: read-variable['jQuery',v12,-]
20: read-property[v12,'attr',v11]
21: <variable-non-null-undef>['jQuery']
22: read-variable['name',v13,-]
23: read-variable['value',v14,-]
24: read-variable['arguments',v18,-]
25: read-property[v18,'length',v16]
26: <variable-non-null-undef>['arguments']
27: constant[1.0,v17]
28: >[v16,v17,v15]
10
14: constant[null,v1]
9
29: call-jQuery.access [v9,'access',v10,v11,v13,v14,v15,v1]
11
30: return[v1]
31: exceptional-return

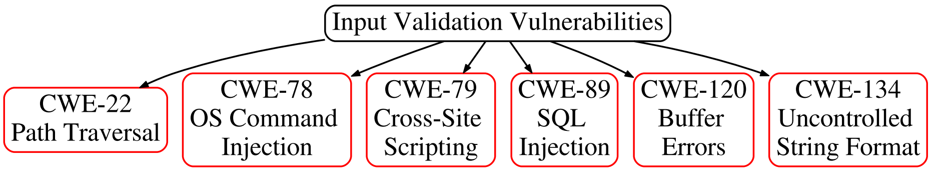

Input Validation Vulnerabilities
CWE-22 Path Traversal
CWE-78 OS Command Injection
CWE-79 Cross-Site Scripting
CWE-89 SQL Injection
CWE-120 Buffer Errors
CWE-134 Uncontrolled String Format

Fig. 5 Input validation vulnerability tree

data used by the Bayesian network comes from three sources: (1) source code, (2) NVD statistics, and (3) CWSS scores. Data extracted from source code with respect to "Input Validation Language for Robustness" design and implementation along with TAJS is complete and precise. However, there is no indication or statement that NVD statistics are complete and precise. Similarly, CWSS scores are calculated using expert judgment, which might be considered as having some uncertainty. The nature of the problem fits to Bayesian networks.

Following the approach proposed by Christey (2005), we classified the input validation vulnerabilities chosen for this work as an input validation vulnerability tree seen in Fig. 5.

The Bayesian network structure uses the data to update probabilities as given in formula (1). In formula (1), X variable represents the observed nodes and x_i is the data point in the given Bayesian network. The *parent*(X_i) stands for the parent nodes of node x_i. In formula (2), we have given an example of $P(x_1,...,x_n)$ which is defined from formula (1). Formula (2) explains that every *Function Input Parameter Status Nodes* (FIPS) except FIPS$_{\text{SQLI}}$. FIPS$_{\text{SQLI}}$ node is set to a "*Not Validated*" state. Therefore, if we extend formula (1), we obtain the given formula in (2).

$$P(x_1,\ldots,x_n) = \prod_{i=1}^{n} P(x_i|\text{parents}(X_i)) \tag{1}$$

$$\begin{aligned}
&P(\text{FIPS}_{\text{UFS}}, \text{FIPS}_{\text{PT}}, \text{FIPS}_{\text{XSS}}, \neg\text{FIPS}_{\text{SQLI}}, \text{FIPS}_{\text{BO}}, \text{FIPS}_{\text{OSCI}}, \text{UFS}, \text{PT}, \text{XSS}, \text{SQLI}, \text{BO}, \text{OSCI}, \text{Application}) \\
&\quad = P(\text{Application} \mid \text{UFS}\wedge\text{PT}\wedge\text{XSS}\wedge\text{SQLI}\wedge\text{BO}\wedge\text{OSCI}). \\
&\quad P(\text{UFS} \mid \text{FIPS}_{\text{UFS}}). \\
&\quad P(\text{PT} \mid \text{FIPS}_{\text{PT}}). \\
&\quad P(\text{XSS} \mid \text{FIPS}_{\text{XSS}}). \\
&\quad P(\text{SQLI} \mid \neg\,\text{FIPS}_{\text{SQLI}}). \\
&\quad P(\text{BO} \mid \text{FIPS}_{\text{BO}}). \\
&\quad P(\text{OSCI} \mid \text{FIPS}_{\text{OSCI}}). \\
&\quad P(\text{FIPS}_{\text{UFS}}).P(\text{FIPS}_{\text{PT}}).P(\text{FIPS}_{\text{XSS}}).P(\neg\text{FIPS}_{\text{SQLI}}).P(\text{FIPS}_{\text{BO}}).P(\text{FIPS}_{\text{OSCI}})
\end{aligned} \tag{2}$$

When constructing a BN, we need to define the relationships between the nodes (see Fig. 2). Each predecessor node affects the successor nodes' probabilistic table (NPT). Therefore, the number of NPTs is equal to the number of nodes in the BN. The number of node probabilistic tables in our BN is

$$\begin{aligned}
&\text{number}\ \text{of}_{\text{vulnerability nodes}} = \text{number}_{\text{of}_{\text{function input parameter status nodes}}} \\
&\text{number}_{\text{of}_{\text{NPTs}}} = \text{number}\ \text{of}_{\text{function input parameter status nodes}} + \text{number}\ \text{of}_{\text{vulnerability nodes}} + 1.
\end{aligned} \tag{3}$$

In formula (3), the value "1" represents the software included as the *Application Node*, while the number of vulnerabilities is six, corresponding to the six input validation vulnerabilities defined above. The number of *Function Input Parameter Status Nodes* is also equal to the number of vulnerability nodes. Each input parameter status node is uniquely directed to a single vulnerability node to separate the required validation codes from other vulnerabilities. In other words, there is a one-to-one correspondence between *Function Input Parameter Status Nodes* and *Input Validation Vulnerability Nodes*. However, it is important to declare that the number of *Function Input Parameter Status Nodes* depends on the number of *Input Validation Vulnerability Nodes* and the number of *Input Validation Vulnerability Nodes* can be changed through the configuration file we have created for our tool. For instance, if an application that does not include and use any SQL statements, the developer can remove the SQL injection input validation vulnerability from the configuration file. Therefore, the new Bayesian network will have five *Input Validation Vulnerability Nodes* and five *Function Input Parameter Status Nodes* instead of six. Instead of removing one input validation vulnerability, if a new type of input validation vulnerability is defined and added to the configuration file, then the number of *Input Validation Vulnerability Nodes* will be seven and following that the number of *Function Input Parameter Status Nodes* will be seven as well.

pt?>In Fig. 6, BN for estimating robustness against input validation vulnerabilities is given. The hierarchy in Fig. 6 from bottom to the top works in three steps. First, all of the function parameters are analyzed with static analysis. The static analysis checks for specific validation codes defined in the source code and pass the information to the upper nodes, which are the input validation vulnerabilities. In the second step, for each, existing validation code in the source code increases the probability of the "Containing Validation Code" of the input validation vulnerability. Then in the third step, in direct proportion, the increase of containment of validation code increases the robustness of the application. The BN can be divided into three layers: *A*, *B*, and *C*. As seen in Fig. 6, the topmost node as layer *A* represents the *Application* NPT; in layer *B*, six *Input Validation Vulnerability Nodes* contain corresponding vulnerability NPTs; and in layer *C*, the *Function Input Parameter Status Nodes* contains the respective function NPTs of the analyzed function parameters if they have a validation code related to its child node (*Input Validation Vulnerability Node*).

All the BNs developed using our proposed method have the same three layer structure. The top layer is always single node, and the remaining two layers have the same number of nodes due to the one-to-one correspondence property explained above. For the time being, we suggest the use of 6-6-1 BN structure since it has the highest coverage in terms of existing input validation vulnerabilities. If a new type of input validation vulnerability is found and defined, the new type of input validation vulnerability can be added into the BN making it 7-7-1 structure. The presented tool enables this addition through the configuration file.

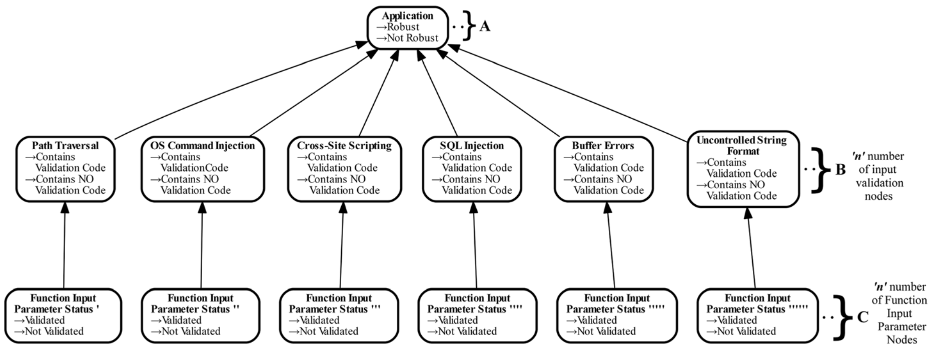

Application
→Robust
→Not Robust
A
Path Traversal
→Contains Validation Code
→Contains NO Validation Code
OS Command Injection
→Contains ValidationCode
→Contains NO Validation Code
Cross-Site Scripting
→Contains Validation Code
→Contains NO Validation Code
SQL Injection
→Contains Validation Code
→Contains NO Validation Code
Buffer Errors
→Contains Validation Code
→Contains NO Validation Code
Uncontrolled String Format
→Contains Validation Code
→Contains NO Validation Code
B
'n' number of input validation nodes
Function Input Parameter Status '
Function Input Parameter Status ''
Function Input Parameter Status '''
Function Input Parameter Status ''''
Function Input Parameter Status '''''
Function Input Parameter Status ''''''
→Validated
→Not Validated
C
'n' number of Function Input Parameter Nodes

Fig. 6 Bayesian network for measuring robustness against input validation vulnerabilities

Table 2 Function input parameter status node probabilistic table

Validation status	Function input parameter status
Validated	0.325
Not validated	0.675

As explained above, developers may propose that one input validation vulnerability is not possible for their software then their BN will have a 5-5-1 structure. For JavaScript, all input validation vulnerabilities taken into consideration in this work have their examples. Therefore, for any software written in JavaScript, the BN must have 6-6-1 structure for highest possible coverage. Hence, all the BNs constructed for the libraries investigated in the case study have 6-6-1 structure.

4.1 Calculating the node probabilistic table for function nodes

The bottom layer *C* in Fig. 6 has six *Function Input Parameter Status Nodes*. These nodes are independent from each other and parent nodes of a single *Input Validation Vulnerability Node*. Each *Function Input Parameter Status NPT* (Table 2) is calculated by checking if the function parameters have a validation code related to its child node (*Input Validation Vulnerability Node*). For example, if the *Function Input Parameter Status Node's* child node is *XSS*, then the function parameters are checked with static analysis if there is a validation code for XSS. After the checks, a probabilistic value is calculated according to usage of validation codes for input validation vulnerabilities. A detailed calculation of *Function Input Parameter Status NPT* calculation is given in algorithm 1.

In algorithm 1, the function input parameter status node probabilistic tables' values are calculated. Every input validation vulnerability could require different validations to avoid invalidated inputs. Therefore, for every vulnerability, a new Function Input Parameter Status (FIPS) node is created. All of the FIPS NPT values are calculated with the same logic. However, the only difference that makes every FIPS nodes identical is that the FIPS node that is directed to the vulnerability uses the vulnerability's validation rules. This provides an independent observation among all the vulnerabilities.

The algorithm first starts with a static analysis that collects information about function and its parameters and obtains information if the parameters are validated or not. Then, the vulnerabilities' validation BNF rules are pre-loaded. Since the functions' parameter validation status information is already extracted from static analysis, an increment is performed over their validation statuses. If the function parameter has a status *Not Validated*, then the "not validated count" is incremented. However, if the function parameter has a status *Validated*, then the validation content is checked with its directed vulnerability. If the validation content contains the corresponding vulnerability's validation rules, then the "validated count" is incremented, otherwise the "not validated count" is incremented.

After the "not validated count" and "validated count" are complete, a ratio calculation is performed. The ratio is performed over the "not validated count" and "validated count" values, divided by the total numbers of parameters. The total number of parameters is equal to the summation of "validated count" and "not validated count.” Finally, ratio values are assigned to each corresponding FIPS node’s probabilistic table.

Algorithm 1 Function Input Parameter Status NPT value calculation.

```
1.  functions[] = getAnalyzedFunctionInfo()
2.  vulnerabilities[] = getVulnerabilities()
3.  FOR i=0 to vulnerabilities.size
4.      vulnerabilityValidationRules[i] = getVulnerabilityValidationRules(vulnerabilities[i])
5.  END FOR
6.  FOR i=0 to vulnerabilities.size
7.      countNotValidated = 0
8.      countValidated = 0
9.      FOR j=0 to functions.size
10.         FOR k=0 to function[j].parameters.size
11.             IF  function[j].parameters[k] = “Not Validated”
12.                 INCREMENT countNotValidated
13.             END IF
14.             ELSE IF function[j].parameters[k] = “Validated”
15.                 IF hasRule(function[j].parameters[k], vulnerabilityValidationRules[i])
16.                     INCREMENT countValidated
17.                 END IF
18.                 ELSE
19.                     INCREMENT countNotValidated
20.                 END ELSE
21.             END ELSE IF
22.         END FOR
23.     END FOR
24.     total = countNotValidated + countValidated
25.     functionNPT[i] = { countNotValidated/total, countValidated/total }
26.     assignFunctionValuesToNPT(functionNPT[i])
27. END FOR
```

4.2 Calculating the node probabilistic table for vulnerability nodes

Second layer B in Fig. 6 has six Input Validation Vulnerability Nodes. The *Input Validation Vulnerability Nodes* have 2 states: “*Contains Validation Code*” and “*Contains NO Validation Code*”. States “*Contains Validation Code*” and “*Contains NO Validation Code*” give a probabilistic value of the analyzed code that has specific validation codes for each corresponding vulnerability. Every *Input Validation Vulnerability Node* has a single predecessor node and every *Input Validation Vulnerability Node’s* predecessor node is different from another. An example NPT for *Input Validation Vulnerability* of SQL Injection is given in Table 3. In the NPT values for *Containing NO Validation Code* and validated function parameters, the weight of the corresponding input validation vulnerability is assigned. The weight of the vulnerability (formula 4) is calculated from the data gained from NVD (Table 3).

Table 3 Input validation vulnerability node probabilistic table

Function input parameter status	Not validated	Validated
Contains NO Validation Code	0.95	0.232026333
Contains Validation Code	0.05	0.767973667

In Table 3, the case, where the inputs are *Validated*, but there are *Containing NO Validation Code* for the SQL Injection, we say that with 0.232026333 probability, the application can encounter an SQL Injection attack. The 0.232026333 probability is obtained from NVD, which describes that, among the six input validation vulnerabilities, the application might be vulnerable against SQL Injection. Because there is a validation code detected in the source code, but the validation is not for SQL Injection directly, 0.232026333 is given. However, if there is a validation (*Validated*) and a specifically *Containing Validation Code* for SQL Injection, then 0.767973667 (1–0.232026333) is given. For not validated function parameters, static values are assigned. If the function input parameter is "*Not Validated*" for the containing vulnerability, we assign 0.95 and "*Containing Validation Code*" vulnerability 0.05 is assigned. A detailed algorithm is given in algorithm 2.

Let $\boldsymbol{\omega_i}$ be the vulnerability weight,
$\boldsymbol{r_i}$ the number of reports of the vulnerability,
$\boldsymbol{n}$ the summation of all input validation vulnerabilities' reports.

$$\omega_i = \frac{r_i}{n} \tag{4}$$

In algorithm 2, the vulnerability node's probabilistic table value calculation is given. Initially, vulnerabilities' reports in the past 15 years are obtained from NVD (Table 4). The reports give the information of how much the vulnerability is frequent, in other terms the frequency of the vulnerability. The vulnerability weights are calculated by their 15 years of individual reports divided to the total reports of input validation vulnerabilities that are used in the Bayesian network.

To fill in the NPT values, there are 4 different cases that should be considered;

- Not validated–Contains Validation Code➔ x = 0.05
- Not validated–Contains NO Validation Code ➔ 1-x = 0.95
- Validated–Contains Validation Code ➔ y
- Validated–Contains NO Validation Code ➔ 1-y

For the cases where there is no validation (not validated), we assume that there is a higher probability that the vulnerability contains the vulnerability. Therefore, constant values for *Not*

Table 4 Input validation vulnerabilities reported between 2000 and 2015, their weights, and CWSS scores

Input validation vulnerabilities	Reported between 2000 and 2015	Vulnerability weights	CWSS scores
SQL Injection	4018	0.232026333	93.8
Buffer Errors	5449	0.314661893	79.0
Uncontrolled Format String	161	0.009297222	61.0
Path Traversal	1742	0.100594791	69.3
OS Command Injections	156	0.009008489	83.3
Cross-Site Scripting (XSS)	5791	0.334411272	77.7
TOTAL	17,317		

Validated–Contains Validation Code = 0.05 and for *Not Validated–Contains NO Validation Code* = 0.95 are given. The constant values are not given directly 0 and 1, but given values close to 0 and 1 to satisfy the basic rules of a Bayesian approach. To fill in the rest of the NPT cases, *Validated–Contains Validation Code* and *Validated–Contains NO Validation Code*, we use the weight that is calculated from formula 4 and used the values from Table 4.

The Bayesian approach is based on beliefs and provides us with uncertain inferences. Uncertainty here means that it is not possible to be 100% sure of anything. Therefore, a small probability of standard deviation is given in BNs. If a probabilistic value like "1" or "0" were given, this would represent a frequentist approach, and not a Bayesian approach. The frequentist approach assigns probabilities to random events according to their frequency of occurrence or subsets of populations as proportions of the whole, whereas the Bayesian approach assigns probabilities to propositions that are uncertain (Korb and Nicholson 2003), thereby we have assigned the *Not Validated–Contains Validation Code* case as 0.05 instead of 0 and for *Not Validated–Contains NO Validation Code* case as 0.95 instead of 1. In addition to 0.95, nine more values are given and the changes of nodes are shown in Appendix 4. It has been observed that on every 0.01 change made on the NPT value of Not Validated and Containing NO Validation Code, the robustness measure and vulnerability nodes change with 0.01 as well.

Algorithm 2 Input validation vulnerability NPT value calculation.

```
//Calculated from the previous reported vulnerabilities and the weights are calculated by the
//distribution among the other input validation vulnerabilities
vulnerabilityWeights[] = calculateVulnerabilityWeights()
vulnerabilities[] = getVulnerabilities()
vulnerabilityNPTValues[]
vulnerabilityStatus[] = {"Contains No Validation Code", "Contains Validation Code"}
inputStatus[] = {"Not Validated", "Validated"}
FOR i=0 to vulnerabilityWeights.size
    FOR j=0 to inputStatus.size
        FOR k=0 to vulnerabilityStatus.size
            value = 0
            IF inputStatus[j] = "Not Validated"
                IF vulnerabilityStatus [k] = "Contains Validation Code"
                    value = 0.95
                    vulnerabilityNPTValues.add(value)
                END IF
                ELSE IF vulnerabilityStatus [k] = "Contains No Validation Code"
                    value = 1 – value
                    vulnerabilityNPTValues.add(value)
                END ELSE IF
            END IF
            ELSE IF inputStatus[j] = "Validated"
                IF vulnerabilityStatus [k] = "Contains Validation Code"
                    value = vulnerabilityWeights[i]
          vulnerabilityNPTValues.add(value)
                END IF
                ELSE IF vulnerabilityStatus [k] = "Contains No Validation Code"
                    value = 1 – value
                    vulnerabilityNPTValues.add(value)
                END ELSE IF
            END ELSE IF
        END FOR
    END FOR
END FOR
assignVulnerabilityValuesToNPT(vulnerabilityNPTValues[])
```

Table 5 Application node probabilistic table

XSS	Contains validation code	Contains validation code	….	Contains NO validation code
SQL Injection	Contains Validation Code	Contains Validation Code	….	Contains NO Validation Code
Buffer Overflow	Contains Validation Code	Contains Validation Code	….	Contains NO Validation Code
Path Traversal	Contains Validation Code	Contains Validation Code	….	Contains NO Validation Code
OS Command Injection	Contains Validation Code	Contains Validation Code	….	Contains NO Validation Code
Uncontrolled Format String	Contains Validation Code	Contains NO Validation Code	….	Contains NO Validation Code
Robust	1	0.82	….	0
Not Robust	0	0.18	….	1

4.3 Calculating the node probabilistic table for application nodes

In Fig. 6, layer *A* has only one node which is the *Application Node*. Inside the *Application Node*, there holds a *Node Probabilistic Table (NPT)*. The NPT keeps information about its predecessor nodes, and through its predecessor nodes, the *Application Node* makes its reasoning by passing the information to its defined two states: "*Robust*" and "*Not Robust.*" As it is seen in Fig. 5, the *Application Node* has 6 predecessor nodes which are the input validation vulnerabilities, and the vulnerability nodes have two states as well. Therefore, the *Application* NPT has 2^6 identical cases. Table 5 shows the structure of Application Node Probabilistic Table. Full Application Node Probabilistic Table is given in Appendix 5. "Robust" and "Not Robust" values at the bottom of the NPT (last two rows) are calculated using algorithm 3.

In Table 5 for each case, all containing vulnerability scores are summed. After calculating every case's score, the case scores are normalized (5). Formula 5 is a 0–1 scale normalization formula, where E is the set of n elements $E = \{e_1, ..., e_n\}$. e_i is the element that is going to be normalized, while E_{min} is the minimum value and E_{max} is the maximum value of set E. The vulnerability scores are obtained from Common Weakness Scoring System (CWSS). The detailed algorithm is given in algorithm 3.

$$\text{Normalized}(e_i) = \frac{e_i - E_{min}}{E_{max} - E_{min}} \tag{5}$$

In algorithm 3, the application node's probabilistic table value calculation is given. To calculate the NPT values, first the vulnerabilities in the BN are received, then the pre-calculated vulnerability scores from CWSS (Table 4) are received. To find all variations of input validation vulnerability cases, a subset finding algorithm is used. For a vulnerability to be contained, there should be no validation code for that input validation vulnerability existing in the source code. For every contained vulnerability, the vulnerability's CWSS score is added to the *caseScore*. In other words, the *caseScore* value is the summation of CWSS scores of contained vulnerabilities. All of the calculated case scores are collected in an array. Since the vulnerability CWSS score values are between [0,100], therefore the case scores could be "0" at minimum and "600" (100 × *number of vulnerabilities*) at maximum. However, in Bayesian networks, the NPTs only accept values between [0,1], so that a 0–1 scale normalization is applied on the calculated array of case scores. Then, the normalized case scores are assigned to their matching case in the NPT.

Algorithm 3 Application NPT value calculation

```
1.  vulnerabilities[] = getVulnerabilities()
2.  vulnerabilityScores[] = getVulnerabilityScores()  //CWSS Score values
3.  vulnerabilityCases[] = findAllVulnerabilityCases()
4.  allCaseScores[]
5.  FOR i=0 to vulnerabilityCases.size
6.      caseScore = 0
7.      FOR j=0 to vulnerabilities.size
8.          IF vulnerabilityCases[i] has vulnerabilities[j]
9.              caseScore = caseScore + vulnerabilityScore[j]
10.         END IF
11.     END FOR
12.     allCaseScores[i] = caseScore
13. END FOR
14. normalizedCaseScores[] = normalize(allCaseScores[])
15. assignScoresToNPT(normalizedCaseScores[])
```

4.4 Implementing an automated Bayesian network generator

A tool is required to keep the information needed to construct a BN. It should be possible to store information about the BN's connections between nodes and every node's NPT in a file. For this purpose, we use OpenMarkov (2014), which is a Java open source software tool. OpenMarkov is an active open source software tool which is compatible with up to date Java, while some similar tools only support previous old versions of Java that causes compatibility problems. Most Bayesian network APIs are drag and drop applications, where nodes and edges are added manually. Although they also provide a BN file to save and reuse it again, however, OpenMarkov provides an XML file format, which is easy to parse and write. In addition, Java has libraries to parse and write XML files. Therefore, by using OpenMarkov's XML file format, we are able to create our own BN. To ensure that the connection between calculating the NPT and file operation of the BN is smooth, we developed the BN generator in Java, because OpenMarkov is written in Java.

We want our BN to be more dynamic, automated, and flexible before we develop the BN generator, so that developers can modify the BN according to their JavaScript application as desired. To achieve this, we prepare a configuration file that is first parsed by our tool. Using this approach, developers are able to add or remove functions and vulnerability nodes. Another benefit of the configuration file is to store information about functions and vulnerabilities. This supports keeping our tool updated. If changes occur related to the statistics of vulnerabilities or functions, the configuration file can be modified to adapt to current values.

The size of the function part of the configuration file grows linearly with the number of functions that the software under consideration has. Similarly, vulnerability part of the configuration file grows linearly with the number of vulnerabilities taken into consideration, which is 6 for the current implementation. The size of the current vulnerability part of the configuration file is 954 bytes.

Our algorithms (evaluating NPTs for each vulnerability and function) are integrated in the tool. OpenMarkov is not directly integrated with our tool and not modified. In addition, one of the reasons that we have chosen and used OpenMarkov is that it is a non-commercial tool. OpenMarkov is added as a jar project to our tool. OpenMarkov uses XML files to run, create, and modify its BNs. Furthermore, XML files are well structured, easy to parse, and write in

Table 6 Robustness estimation for five selected functions from five different libraries

	Robust
Analysis on Raw Code	0.06209
Analysis on Validation Added Code	0.82710

Java, due to the XML libraries that exists in Java. Thereby, our tool creates an XML for OpenMarkov, and the XML file becomes a bridge between our tool and OpenMarkov. Our tool triggers OpenMarkov to run the BN file that is created by our tool. Next, we start building our BN by first defining the nodes. Then, we establish the connections and relationships between nodes. We define the architecture of our BN, which is constructed in three levels: "*Application*," "*Input Validation Vulnerability*," and "*Function Input Parameter Status*." When including the relationships of our nodes in our tool, we direct all vulnerability nodes to the application node and each *Function Input Parameter Status Node* to a single *Input Validation Vulnerability* node. Finally, we assign the calculated NPTs to the nodes.

4.5 Proof of concept

In Table 6, the overall robustness is calculated for two cases: first, the raw format of selected five functions from five different JavaScript libraries and second, same selected five functions from five different JavaScript libraries but with added validation code by us (see Appendix 3). When the first case is analyzed, the robustness has been calculated as 0.06209 (Table 6). In Table 7, we see that there is some validation code containing for *Buffer Overflow* in the raw format of the analyzed code. The validation code for functions *extractElementNode* and *d3_selection_each*, due to the validation usage length. However, for the second case, where validation code is added to the five functions, our tool has detected validation codes for input validation vulnerabilities. The NPT values are calculated as 1 (Table 8), because all of the defined input validation rules for input validation vulnerabilities are defined for every function parameter. Therefore, the second case's robustness has been calculated as 0.82710 (Table 6) by the Bayesian network.

Preconditions could be stored in a function and called from every function that accepts input data as argument. Our method considers this possibility as well and through the flow diagram it detects whether inputs are checked or not in another function before they are used.

5 Case study

As case study, we selected 10 JavaScript libraries. Before giving their robustness estimation and their evaluation, we present application of our method to jQuery version 1.9.1, where all

Table 7 Analysis results on five selected functions from five different libraries

	Format string vulnerability	Path traversal	XSS	SQL Injection	Buffer overflow	OS Command Injection
Contains Validation Code	0	0	0	0	0.11111	0
Contains NO Validation Code	1	1	1	1	0.88889	1

Table 8 Analysis results on five selected functions with added input validation from five different libraries

	Format string vulnerability	Path traversal	XSS	SQL Injection	Buffer overflow	OS Command Injection
Contains Validation Code	1	1	1	1	1	1
Contains NO Validation Code	0	0	0	0	0	0

functions of jQuery are covered. In Fig. 7, a screenshot of jQuery's Bayesian network for estimating its robustness in relation to input validation vulnerabilities is given. At the rightmost part of the BN, we can see the *Function Input Parameter Status* (*FIPS*) *Nodes*. On the left side of them, their child nodes of *Input Validation Vulnerability Nodes* are given. Among all of the *FIPS Nodes*, only one node that is connected to the "*Buffer Overflow*" vulnerability has shown different values than the others. This means that some validation code has been detected by the static analysis for *Buffer Overflow* vulnerabilities. Therefore, the *Buffer Overflow* node and its parent *FIPS Node* have shown different probabilities than the others, and its value for *Validated* has been calculated as 0.00362. This has affected the *Buffer Overflow* node's *Contains Validation Code* as 0.05232. The static analysis cannot find any other validation code for the other *FIPS Nodes*, thereby their *Not Validated* status probabilities are given as 1. In conclusion for jQuery and with its few containment of validation code for *Buffer Overflow*, the robustness has been calculated as 0.05039.

Descriptions of selected JavaScript libraries for the case study are given in Table 9. They are well-known and regularly used JavaScript libraries. Six of them are client-side, and the remaining four of them are server-side libraries. We applied our method to each of them covering all of their functions. Their robustness estimations are presented in Table 10.

A BN is generated for each library. When BNs are investigated, it is observed that these libraries have only very few validations for Buffer Overflow vulnerabilities. Further analysis has shown that there are no validation codes for other input validation vulnerabilities. Out of ten

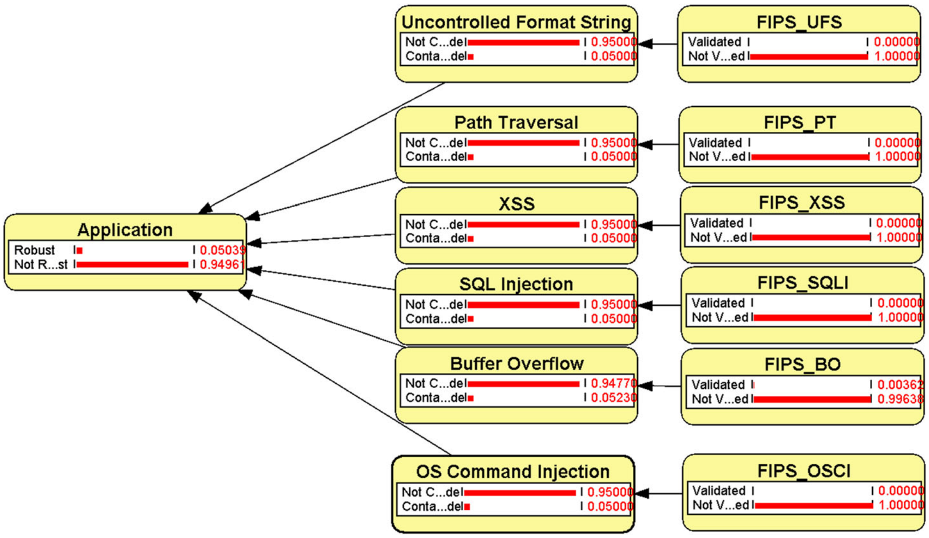


Fig. 7 Generated Bayesian network for estimating jQuery's robustness in relation to input validation vulnerabilities

Table 9 Description of JavaScript libraries evaluated in the case study

Location	Description
client-side	AngularJS (https://angularjs.org/) is a web application and GUI related JavaScript library that lets you write client-side web applications. It lets you use the old HTML as your template language and lets you extend HTML's syntax to express your application's components clearly and sufficiently. It automatically synchronizes data from user interface with developer's JavaScript objects through 2-way data binding.
client-side	D3 (http://d3js.org/) is a graphical/visualization JavaScript library for manipulating documents based on data. It uses HTML, SVG, and CSS on data for visualization. D3 follows a data-driven approach to DOM manipulation.
client-side	Joose (http://joose.it/) is a JavaScript library that provides "keywords" for class declaration, object construction, inheritance and more. These keywords become a part of the language and help the developer not to care about the implementation details of all these concepts.
client-side	jQuery (http://jquery.com/) is a DOM oriented JavaScript library that performs manipulation, event handling, animation and document traversal.
client-side	Prototype JavaScript Framework (http://prototypejs.org/) is a DOM and web application oriented JavaScript library that takes the complexity out of client-side web programming. It adds useful extensions to the browser scripting environment and provides elegant APIs around the interfaces of DOM and Ajax.
client-side	Zepto (http://zeptojs.com/) is a JavaScript library for modern browsers compatible with jQuery API.
server-side	Backbone (http://backbonejs.org/) is a web application related library that gives structure to web applications by providing models with key-value binding and custom events, collections with a rich API of enumerable functions, views with declarative event handling and connects it to all of existing API over a RESTful JSON interface.
server-side	Chaplin (http://chaplinjs.org/) is a web application related JavaScript library and an architecture for JavaScript applications using the Backbone library. Chaplin addresses Backbone library's limitations by providing a lightweight and flexible structure that features well-proven design patterns.
server-side	Dojo (http://dojotoolkit.org/) is a DOM oriented JavaScript library that scales with the development process, using web standards as its platform. It is also a toolkit for building desktop and mobile web applications.
server-side	Handlebars (http://handlebarsjs.com/) is a template system JavaScript library that provides to build semantic templates effectively. It is an extension to another JavaScript library Mustache templating language.

libraries, D3 has the most input validation code; therefore, D3 has the highest robustness measure as 0.05116 among all libraries. On the other hand, Backbone does not have any input validation code; therefore, it has the lowest robustness value of all other JavaScript libraries. In an overall review of JavaScript libraries, it is seen that JavaScript libraries have very few precautions against input validation vulnerabilities. The only validation code that libraries include are related Buffer

Table 10 Robustness estimation of JavaScript libraries evaluated in the case study

No.	JavaScript Library	Type	Version	Robustness
1	jQuery	Client-side	1.9.1	0.05039
2	Joose	Client-side	2.1	0.05064
3	Zepto	Client-side	1.1.6	0.05103
4	Backbone	Server-Side	1.1.2	0.05
5	AngularJS	Client-side	1.4.2	0.05061
6	D3	Client-side	3.5.5	0.05116
7	Dojo	Server-Side	1.10.4	0.05105
8	Chaplin	Server-Side	1.0.1	0.05033
9	Handlebars	Server-Side	3.0.0	0.05052
10	Prototype JavaScript Framework	Client-side	1.7.2	0.05047

Table 11 JavaScript libraries robustness measurement runtime table

No.	JavaScript library	Number of functions	SLOC	Min runtime (s)	Max runtime (s)	Average runtime (s)
1	jQuery	579	9597	11	29	16.12
2	Joose	361	3802	4	10	6.88
3	Zepto	255	1587	3	8	4.68
4	Backbone	123	1610	2	4	2.24
5	AngularJS	1350	28,366	27	54	37.28
6	D3	1511	9504	26	67.8	36.55
7	Dojo	974	18,566	16	58	27.52
8	Chaplin	284	3111	3	18	6.24
9	Handlebars	271	3746	5	11	6.72
10	Prototype JavaScript Framework	733	7510	10	44	20.68

Overflow, which is also very few. The average robustness of all ten JavaScript libraries has been calculated as 0.05062, which shows that JavaScript libraries are not that robust as we rely on them.

Table 11 is introduced to present the number of functions each library we investigated has. In Table 11, the performance of runtime evaluation over ten JavaScript libraries is given. The runtime values are gathered by running our tool 25 times repeatedly per libraries. The performance evaluation covers the complete process from the start until to the final process of robustness calculation. This process includes static analysis, function input parameter analysis, calculation of probabilistic data for BN, BN creation, and execution (robustness calculation). It has been observed that the average runtime is directly proportional with the number of functions of the analyzed JavaScript file. The average runtimes of the analyzed JavaScript libraries take less than a minute. Among the JavaScript libraries, Backbone has the least average runtime with 2.24 s, while AngularJS has the longest runtime with 37.28 s, due to its number of functions and SLOC. The mean value of average runtimes of ten JavaScript libraries has been calculated as 16.49 s, which might be considered as a reasonable time for robustness calculation.

6 Tool support

In this work, a tool called TAJS has been extended and used to calculate values for Bayesian network. TAJS is a JavaScript static analysis tool that is developed by Jensen et al. (2009). It is developed to support JavaScript developers, detect errors, and help understanding the code. TAJS supports DOM and all ECMAScript languages that performs a dataflow analysis. It models the JavaScript semantic with a rich lattice structure. TAJS performs a call graph extraction, data type analysis, removal of *eval()* function, and detection of unused code through a flow graph it generates.

For this research, TAJS is extended to collect function and function parameter information through its flow graph output. This helps to initialize and create a data structure for the inputs to functions (function parameters), thereby when an analysis is performed on inputs, the information that is obtained after the analysis can easily be stored into this data structure.

TAJS is extended further to check for specific type of validations on function parameters. To perform this check, TAJS's flow graph output is used. The flow graph output that TAJS provides is in *.dot* format, which is a graphical representation file. Flow graph output is a graphical representation of a source code that shows the interactions inside the code. Therefore, the extension that is made in TAJS searches for function parameters that are previously extracted from the first extension. If the function parameter is found in the flow graph, it checks for "*if[r]*" instructions, if a validation is performed on the function parameter. The next is checking the "*if[r]*" instruction's content, if the content has the function parameter in it and also has any input validation codes for input validation vulnerabilities then it collects these information to be used later in the BN.

The last extension performed on TAJS is addition of a package for creation of a BN file for OpenMarkov from its analysis results, transferring it to OpenMarkov and triggering OpenMarkov to execute the BN file. The BN file holds necessary data for the connections between the nodes of BN.

7 Conclusion

In this paper, we proposed a method for estimating software robustness in relation to input validation vulnerabilities using BN and developed a tool to construct an automated BN generator to measure robustness of JavaScript applications. We collected 15 years of reported statistics about input validation vulnerabilities. We investigated the effect of each input validation vulnerability in software. Through our investigations, we decided on six input validation vulnerabilities, of which Common Weakness Scores are available, and classified them using a tree. Performing input validations for input validation vulnerabilities is recommended in the design and development phases of the SDLC. Therefore, we chose static analysis to analyze all functions in the software to extract function call graphs and flow graphs using the Input Validation Language for Robustness.

The BN constructed is expressed in three levels: function input parameter status, vulnerability, and the application's robustness. The function input parameter status represents the function nodes we analyzed using the Input Validation Language for Robustness. The vulnerability level represents our input validation vulnerability nodes, including the common weakness scores of each vulnerability. The final top level gives us the overall estimation of the robustness of the application or library in relation to input validation vulnerabilities.

In the case study, we applied our method to JavaScript's most popular library, jQuery, owing to its widespread usage and preference by JavaScript developers. When all functions are evaluated, jQuery is estimated to be 5% robust in relation to input validation vulnerabilities. In addition to estimating the overall robustness, we also observed the probability of each input validation vulnerability occurring in the application. The case study includes nine more JavaScript libraries. We investigated in total 10 popular JavaScript libraries, including both client-side and server-side libraries, and report our findings. With the proposed method, we checked all the functions of these ten JavaScript libraries (6441 functions in total) and found that JavaScript libraries are not robust with respect to input validation vulnerabilities. In other words, they do not contain the necessary source code to withstand data attacks, such as SQL injection. The average robustness of all ten JavaScript libraries has been calculated as 0.05062, which shows that JavaScript libraries are not that robust as we rely on them. This is valuable information for the software developers, who use or plan to use these libraries.

Our investigation of related work that uses BNs shows that the distribution is usually considered as uniform. However, in our model and tool, we use statistics obtained from real data, which makes our work and estimation realistic and unique. Another advantage is that the tool we use is flexible and dynamic. Therefore, developers can easily introduce their own preconditions and scores by simply modifying the configuration file.

As a future work, we would like to compare our approach with software robustness testing. Moreover, we would like to also include languages like Java, C++, and Python.

Following the well-known quote "you can't control what you can't measure," it is almost impossible to react to or take precautions without any estimation of robustness in relation to input validation vulnerabilities. If developers become aware of the robustness estimation of their software, they can improve it accordingly.

Appendix 1. Descriptions of input validation vulnerabilities in the scope of work

CWE-79: Improper Neutralization of Input During Web Page Generation ("Cross-site Scripting").

Description Summary.

The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

http://cwe.mitre.org/data/definitions/79.html

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection").

Description Summary.

The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

http://cwe.mitre.org/data/definitions/89.html

CWE-120: Buffer Copy without Checking Size of Input ("Classic Buffer Overflow").

Description Summary.

The program copies an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer, leading to a buffer overflow.

http://cwe.mitre.org/data/definitions/120.html

CWE-22: Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal").

Description Summary.

The software uses external input to construct a pathname that is intended to identify a file or directory that is located underneath a restricted parent directory, but the software does not properly neutralize special elements within the pathname that can cause the pathname to resolve to a location that is outside of the restricted directory.

http://cwe.mitre.org/data/definitions/22.html

CWE-78: Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection").

Description Summary.

The software constructs all or part of an OS command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended OS command when it is sent to a downstream component.

http://cwe.mitre.org/data/definitions/78.html

CWE-134: Uncontrolled Format String.

Description Summary.

The software uses externally-controlled format strings in printf-style functions, which can lead to buffer overflows or data representation problems.

https://cwe.mitre.org/data/definitions/134.html

Appendix 2. Input validation vulnerability examples

CWE-78: Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection").

An OS Command Injection using Node.JS

```
child_process.exec('ls', function (err, data) {
   console.log(data);
});
```

Another example of an OS Command injection.

```
child_process.spawn('/bin/ls', ['-l', '.'])
```

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ("SQL Injection").

Assume that the user has defined a database with JavaScript.

```
var db = openDatabase('mydb', '1.0', 'Test DB', 6*1024*1024);
db.transaction(function (tx) {
            tx.executeSql('DROP TABLE IF EXISTS LOGS');
            tx.executeSql('CREATE TABLE IF NOT EXISTS LOGS (id unique, log)');
            tx.executeSql('INSERT INTO LOGS (id, log) VALUES (1, "foobar")');
  tx.executeSql('INSERT INTO LOGS (id, log) VALUES (2, "info")');
});
```

The attacker can inject an SQL as,

```
db.transaction(function (tx) {            tx.executeSql('DROP TABLE IF EXISTS LOGS');})
```

CWE-120: Buffer Copy without Checking Size of Input ("Classic Buffer Overflow").

JavaScript heap spraying example (https://crypto.stanford.edu/cs155old/cs155-spring11/lectures/03-ctrl-hijack.pdf)

```
var nop = unescape("%u9090%u9090")
while(nop.length<0x100000)
nop += nop
var shellcode = unescape("%u4343%4343%...");
var x = new Array()
for(i=0; i<1000; i++)
{
            x[i] = nop + shellcode;
}
```

Appendix 3. Five JavaScript functions for Proof of Concept

Original JavaScript functions (in regular font style) empowered with input validation code (in bold font style).

```
/**
 * jQuery version 1.9.1 Functions: remove
 * */

function remove ( selector, keepData ) {
  if(selector.indexOf(" OR ") == -1 &&
     selector.indexOf(" AND ") == -1 &&
     selector.indexOf(" IS NULL ") == -1 &&
     selector.indexOf("<script>") == -1 &&
     selector.length <= 1024 &&
     selector.indexOf("../") == -1 &&
     selector.indexOf("..%u2216") == -1 &&
     selector.indexOf("..%c0%af") == -1 &&
     selector.indexOf("..\\") == -1 &&
     selector.indexOf("%2e%2e%2f") == -1 &&
     selector.indexOf("..%255c") == -1 &&
     selector.indexOf("%") == -1 &&
     selector.indexOf(".exe") == -1 &&
     selector.indexOf("\/bin\/") == -1 &&

     keepData.indexOf(" OR ") == -1 &&
     keepData.indexOf(" AND ") == -1 &&
     keepData.indexOf(" IS NULL ") == -1 &&
     keepData.indexOf("<script>") == -1 &&
     keepData.length <= 1024 &&
     keepData.indexOf("../") == -1 &&
     keepData.indexOf("..%u2216") == -1 &&
     keepData.indexOf("..%c0%af") == -1 &&
     keepData.indexOf("..\\") == -1 &&
     keepData.indexOf("%2e%2e%2f") == -1 &&
     keepData.indexOf("..%255c") == -1 &&
     keepData.indexOf("%") == -1 &&
     keepData.indexOf(".exe") == -1 &&
     keepData.indexOf("\/bin\/") == -1
  )
  {
            var elem,
            i = 0;

            for ( ; (elem = this[i]) != null; i++ ) {
                     if ( !selector || jQuery.filter( selector, [ elem ] ).length > 0 ) {
                              if ( !keepData && elem.nodeType === 1 ) {
                                       jQuery.cleanData( getAll( elem ) );
                              }

                              if ( elem.parentNode ) {
                                       if ( keepData && jQuery.contains( elem.ownerDocument, elem ) ) {
                                                setGlobalEval( getAll( elem, "script" ) );
                                       }
                                       elem.parentNode.removeChild( elem );
                              }
                     }
            }
  }
  return this;
```

```
}


/**
 * AngularJS version: 1.4.2 Function: extractElementNode
 * */
function extractElementNode(element) {
  if(element.indexOf(" OR ") == -1 &&
    element.indexOf(" AND ") == -1 &&
    element.indexOf(" IS NULL ") == -1 &&
    element.indexOf("<script>") == -1 &&
    element.indexOf("../") == -1 &&
    element.indexOf("..%u2216") == -1 &&
    element.indexOf("..%c0%af") == -1 &&
    element.indexOf("..\\") == -1 &&
    element.indexOf("%2e%2e%2f") == -1 &&
    element.indexOf("..%255c") == -1 &&
    element.indexOf("%") == -1 &&
    element.indexOf(".exe") == -1 &&
    element.indexOf("\/bin\/") == -1
  )
  {
            for (var i = 0; i < element.length; i++) {
                    var elm = element[i];
                    if (elm.nodeType === ELEMENT_NODE) {
                            return elm;
                    }
            }
  }
}

/**
 * D3 version: 3.5.5 Function: d3_selection_each
 * */
function d3_selection_each(groups, callback) {
  if(groups.indexOf(" OR ") == -1 &&
    groups.indexOf(" AND ") == -1 &&
    groups.indexOf(" IS NULL ") == -1 &&
    groups.indexOf("<script>") == -1 &&
    groups.length <= 1024 &&
    groups.indexOf("../") == -1 &&
    groups.indexOf("..%u2216") == -1 &&
    groups.indexOf("..%c0%af") == -1 &&
    groups.indexOf("..\\") == -1 &&
    groups.indexOf("%2e%2e%2f") == -1 &&
    groups.indexOf("..%255c") == -1 &&
    groups.indexOf("%") == -1 &&
    groups.indexOf(".exe") == -1 &&
    groups.indexOf("\/bin\/") == -1 &&

    callback.indexOf(" OR ") == -1 &&
    callback.indexOf(" AND ") == -1 &&
    callback.indexOf(" IS NULL ") == -1 &&
    callback.indexOf("<script>") == -1 &&
    callback.length <= 1024 &&
    callback.indexOf("../") == -1 &&
    callback.indexOf("..%u2216") == -1 &&
    callback.indexOf("..%c0%af") == -1 &&
    callback.indexOf("..\\") == -1 &&
    callback.indexOf("%2e%2e%2f") == -1 &&
    callback.indexOf("..%255c") == -1 &&
    callback.indexOf("%") == -1 &&
    callback.indexOf(".exe") == -1 &&
    callback.indexOf("\/bin\/") == -1
  )
  {
            for (var j = 0, m = groups.length; j < m; j++) {
                    for (var group = groups[j], i = 0, n = group.length, node; i < n; i++) {
                            if (node = group[i]) callback(node, i, j);
                    }
            }
  }
  return groups;
}


/**
 * Backbone version: 1.1.2 Function: trigger
 * */
function trigger(name) {
  if(name.indexOf(" OR ") == -1 &&
    name.indexOf(" AND ") == -1 &&
    name.indexOf(" IS NULL ") == -1 &&
    name.indexOf("<script>") == -1 &&
    name.length <= 1000 &&
    name.indexOf("../") == -1 &&
    name.indexOf("..%u2216") == -1 &&
    name.indexOf("..%c0%af") == -1 &&
    name.indexOf("..\\") == -1 &&
    name.indexOf("%2e%2e%2f") == -1 &&
    name.indexOf("..%255c") == -1 &&
    name.indexOf("%") == -1 &&
    name.indexOf(".exe") == -1 &&
    name.indexOf("\/bin\/") == -1
  )
  {
            if (!this._events) return this;
            var args = slice.call(arguments, 1);
            if (!eventsApi(this, 'trigger', name, args)) return this;
            var events = this._events[name];
            var allEvents = this._events.all;
            if (events) triggerEvents(events, args);
            if (allEvents) triggerEvents(allEvents, arguments);

            return this;
  }
}

/**
 * Zepto version: 1.1.6 Function: className
 * */
function className(node, value){
  if(node.indexOf(" OR ") == -1 &&
    node.indexOf(" AND ") == -1 &&
    node.indexOf(" IS NULL ") == -1 &&
    node.indexOf("<script>") == -1 &&
    node.length <= 1024 &&
    node.indexOf("../") == -1 &&
    node.indexOf("..%u2216") == -1 &&
    node.indexOf("..%c0%af") == -1 &&
    node.indexOf("..\\") == -1 &&
    node.indexOf("%2e%2e%2f") == -1 &&
        node.indexOf("..%255c") == -1 &&
        node.indexOf("%") == -1 &&
        node.indexOf(".exe") == -1 &&
        node.indexOf("\/bin\/") == -1 &&

        value.indexOf(" OR ") == -1 &&
        value.indexOf(" AND ") == -1 &&
        value.indexOf(" IS NULL ") == -1 &&
        value.indexOf("<script>") == -1 &&
        value.length <= 1024 &&
        value.indexOf("../") == -1 &&
        value.indexOf("..%u2216") == -1 &&
        value.indexOf("..%c0%af") == -1 &&
        value.indexOf("..\\") == -1 &&
        value.indexOf("%2e%2e%2f") == -1 &&
        value.indexOf("..%255c") == -1 &&
        value.indexOf("%") == -1 &&
        value.indexOf(".exe") == -1 &&
        value.indexOf("\/bin\/") == -1
    )
    {
            var klass = node.className || '',
            svg   = klass && klass.baseVal !== undefined

            if (value === undefined) return svg ? klass.baseVal : klass
                            svg ? (klass.baseVal = value) : (node.className = value)
    }
}
```

Appendix 4. Effect of values at vulnerability node on robustness estimation

Different values for “not validated” and “contains no validation code” at vulnerability node and their effect on the robustness estimation

	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	1
Robustness	0.1004	0.0904	0.0804	0.0704	0.0604	0.0504	0.0404	0.0304	0.0204	0.0104	0.0004
Format String Vulnera-bility	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	1
Path Traversal	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	1
XSS	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	1
SQL Injection	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	1
Buffer Overflow	0.8979	0.9078	0.9178	0.9278	0.9377	0.9477	0.9577	0.9676	0.9776	0.9875	0.9975
OS Command Injection	0.9	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	1

Since there exists only input validation code for Buffer Overflow, its value is different than other vulnerability node values

Appendix 5. Full Application Node Probabilistic Table

Case 33	Case 34	Case 35	Case 36	Case 37	Case 38	Case 39	Case 40	Case 41	Case 42	Case 43	Case 44	Case 45	Case 46	Case 47	Case 48
Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code
0,87	0,69	0,7	0,52	0,67	0,49	0,5	0,32	0,7	0,52	0,53	0,35	0,5	0,32	0,33	0,15
0,13	0,31	0,3	0,48	0,33	0,51	0,5	0,68	0,3	0,48	0,47	0,65	0,5	0,68	0,67	0,85

Case 49	Case 50	Case 51	Case 52	Case 53	Case 54	Case 55	Case 56	Case 57	Case 58	Case 59	Case 60	Case 61	Case 62	Case 63	Case 64
Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code
0,72	0,54	0,55	0,37	0,52	0,34	0,35	0,17	0,55	0,37	0,38	0,2	0,35	0,17	0,18	0
0,28	0,46	0,45	0,63	0,48	0,66	0,65	0,83	0,45	0,63	0,62	0,8	0,65	0,83	0,82	1

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10	Case 11	Case 12	Case 13	Case 14	Case 15	Case 16
XSS	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code
SQL Injection	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code
Buffer Overflow	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Path Traversal	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
OS Command Injection	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code
Uncontrolled Format String	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code
Robust	1	0,82	0,83	0,65	0,8	0,62	0,63	0,45	0,83	0,65	0,66	0,48	0,63	0,45	0,46	0,28
Not Robust	0	0,18	0,17	0,35	0,2	0,38	0,37	0,55	0,17	0,35	0,34	0,52	0,37	0,55	0,54	0,72

Case 17	Case 18	Case 19	Case 20	Case 21	Case 22	Case 23	Case 24	Case 25	Case 26	Case 27	Case 28	Case 29	Case 30	Case 31	Case 32
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code
Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code	Contains Validation Code	Contains Validation Code	Contains NO Validation Code	Contains NO Validation Code
Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code	Contains Validation Code	Contains NO Validation Code
0,85	0,67	0,68	0,5	0,65	0,47	0,48	0,3	0,68	0,5	0,51	0,33	0,48	0,3	0,31	0,13
0,15	0,33	0,32	0,5	0,35	0,53	0,52	0,7	0,32	0,5	0,49	0,67	0,52	0,7	0,69	0,87

References

Alkhalaf, M.A. (2014). Automatic Detection and Repair of Input Validation and Sanitization Bugs (PhD dissertation, University Of California Santa Barbara).

Avizienis, A., Laprie, J., Randell, B. (2001). Fundamental Concepts of Dependability, Tech. Rep. 1145, University of Newcastle.

Ben-Gal, I. (2007). Bayesian networks. In F. Ruggeri, F. Faltin, & R. Kenett (Eds.), *Encyclopedia of statistics in Quality & Reliability*. New York: Wiley.

Bobbio, A., Portinale, L., Minichino, M., & Ciancamerla, E. (2001). Improving the analysis of dependable systems by mapping fault trees into Bayesian networks. *Reliability Engineering & System Safety, 71*(3), 249–260.

Christey, S. (2005). Preliminary list of vulnerability examples for researchers. *NIST Workshop Defining the State of the Art of Software Security Tools*, Gaithersburg, MD.

Dejaeger, K., Verbraken, T., & Baesens, B. (2013). Toward comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering, 39*(2), 237–257. doi:10.1109/TSE.2012.20.

Fenton, N., & Neil, M. (2012). *Risk assessment and decision analysis with Bayesian networks*. Boca Raton: CRC Press.

Franke, U., Johnson, P., König, J., & Marcks von Würtemberg, L. (2011). Availability of enterprise IT systems: an expert-based Bayesian framework. *Software Quality Journal, 20*(2), 369–394. doi:10.1007/s11219-011-9141-z.

Frigault, M., & Wang, L. (2008). Measuring network security using Bayesian network-based attack graphs. In *32nd annual IEEE international conference on computer software and applications (COMPSAC '08)* (pp. 698–703).

Guarnieri, S., & Livshits, V.B. (2010). Gulfstream: Incremental static analysis for streaming Java Script applications. In *Proc. of the USENIX Conference on Web Application Development*, accessible through http://static.usenix.org/event/webapps10/tech/full_papers/Guarnieri.pdf.

Halfond, W.G., Viegas, J., & Orso, A., (2006). A classification of SQL-injection attacks and counter-measures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (Vol. 1, pp. 13–15). IEEE.

Heckerman, D. (1996). A tutorial on learning with Bayesian networks. *Technical Report Microsoft Research*, MSR-TR-96-06, http://research.microsoft.com/apps/pubs/?id=69588.

Holm, H., Korman, M., & Ekstedt, M. (2014). A Bayesian network model for likelihood estimations of acquirement of critical software vulnerabilities and exploits. *Information and Software Technology, 58*, 304–318. doi:10.1016/j.infsof.2014.07.001.

IEEE Std 610.12-1990 (1990). IEEE Standard Glossary of Software Engineering Terminology.

Jensen, S.H., Møller, A., & Thiemann, P. (2009). Type analysis for Java Script. In *Proc. 16th International Static Analysis Symposium, SAS '09*, *LNCS* (vol. 5673, pp. 238–255). Berlin Heidelberg New York: Springer.

Jourdan, G. V. (2008). Data validation, data neutralization, data footprint: a framework against injection attacks. *Open Software Engineering Journal, 2*, 45–54.

Kondakci, S. (2010). Network security risk assessment using Bayesian belief networks. In *IEEE international conference on social computing (social com)* (pp. 952–960).

Korb, K. B., & Nicholson, A. E. (2003). *Bayesian artificial intelligence*. Boca Raton: CRC Press.

Kuperman, B. A., Brodley, C. E., Ozdoganoglu, H., Vijaykumar, T. N., & Jalote, A. (2005). Detection and prevention of stack buffer overflow attacks. *Communications of the ACM, 48*(11), 50–56.

National Vulnerability Database (2014). http://web.nvd.nist.gov/view/vuln/statistics. Accessed 5 June 2015.

Okutan, A., & Yıldız, O. T. (2012). Software defect prediction using Bayesian networks. *Empirical Software Engineering, 2*, 154–181. doi:10.1007/s10664-012-9218-8.

OpenMarkov (2014). http://www.openmarkov.org/. Accessed 5 June 2015.

Perkusich, M., Soares, G., Almeida, H., & Perkusich, A. (2015). A procedure to detect problems of processes in software development projects using Bayesian networks. *Expert Systems with Applications, 42*(1), 437–450. doi:10.1016/j.eswa.2014.08.015.

Shahrokni, A., & Feldt, R. (2013). A systematic review of software robustness. *Information and Software Technology, 55*, 1–17.

Smithline, N. (2013). OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10. Accessed 5 June 2015.

Wagner, S. (2010). A Bayesian network approach to assess and predict software quality using activity-based quality models. *Information and Software Technology, 52*, 1230–1241.

Weber, P., Medina-Oliva, G., Simon, C., & Iung, B. (2012). Overview on Bayesian networks applications for dependability, risk analysis and maintenance areas. *Engineering Applications of Artificial Intelligence, 25*(4), 671–682. doi:10.1016/j.engappai.2010.06.002.

Ekincan Ufuktepe received the B.S. degree in Computer Engineering from Izmir University of Economics in 2011 and M.S. degree in Computer Engineering from Izmir Institute of Technology in 2014. He is currently a PhD student in Computer Engineering since 2014 at Izmir Institute of Technology. He worked as an intern for six months in SAP Labs, Security & Trust Department, France in 2013. He has been working as a research assistant in Izmir Institute of Technology since 2012.

Tugkan Tuglular received the B.S., M.S., and Ph.D. degrees in Computer Engineering from Ege University, Turkey in 1993, 1995 and 1999, respectively. He worked as a research associate at Purdue University from 1996 to 1998. He has been with Izmir Institute of Technology since 2000.