WEB APPLICATION VULNERABILITY PREDICTION USING MACHINE LEARNING

[1]Vignesh M, [2]Dr. K. Kumar
[1]PG Scholar, [2]Assistant Professor
Department of Computer Science and Engineering,
Government College of Technology, Coimbatore
Mail id: mvignesh240@gmail.com, kkumar@gct.ac.in

Abstract-Web applications have become one of the most important communication channels between various kinds of service providers and clients. Along with the increased importance of Web applications, the negative impact of security in such applications has grown as well. Due to the limited time and resources, web software engineers need support in identifying vulnerable code. A practical approach to predicting vulnerable code would enable them to prioritize security auditing efforts. In this proposed system hybrid (static + dynamic) program attributes are used to characterize input validation and sanitation code patterns which act as a significant indicator of web application vulnerabilities. Current vulnerable prediction techniques rely on the availability of data labeled with the vulnerability information for training. For most web application, past vulnerability data is often not available or at least not complete. Hence to address both situations where labeled past data is fully available or not fully available, this approach can be used. The web program is sliced into small sinks and by using dynamic and static program analysis, input validation and sanitation attributes are generated.

Keywords – Input validation, Input sanitation, Static Program analysis, Dynamic program analysis, Machine learning.

I INTRODUCTION

Web applications play an important role in many of our daily activities such as social networking, email, banking, shopping, registrations, and so on. As web software is also highly accessible, web application vulnerabilities arguably have greater impact than vulnerabilities in other types of software. Web developers are directly responsible for the security of web applications. Unfortunately, they often have limited time to follow up with new arising security issues and are often not provided with adequate security training to become aware of state-of-the-art web security techniques. Input validation and input sanitization are two secure coding techniques that they can adopt to protect their programs from such common vulnerabilities. Input validation typically checks an input against required properties like data length, range, type, and sign. Input sanitization, in general, cleanses an input string by accepting only pre-defined characters and rejecting others, including characters with special meaning to the interpreter under consideration. Intuitively, an application is vulnerable if the developers failed to implement these techniques correctly or to a sufficient degree.

The code attributes that characterize validation and sanitization code implemented in the program could be used to predict web application vulnerabilities. Based on this hypothesis, we propose a set of code attributes called input validation and sanitization (IVS) attributes from which we build vulnerability predictors that are fine-grained, accurate, and scalable. The approach is fine-grained because it identifies vulnerabilities at program statement levels. We use both static and dynamic program analysis techniques to extract IVS attributes. Static analysis can help assess general properties of a program. Yet, dynamic analysis can focus on more

specific code characteristics that are complementary to the information obtained with static analysis. We use dynamic analysis only to infer the possible types of input validation and sanitization code, rather than to precisely prove their correctness, and apply machine learning on these inferences for vulnerability prediction. Therefore, we mitigate the scalability issue typically associated with dynamic analysis. Thus, our proposed IVS attributes reflect relevant properties of the implementations of input validation and input sanitization methods in web programs and are expected to help predict vulnerabilities in an accurate and scalable manner. Furthermore, both supervised learning and semi-supervised learning methods are used to build vulnerability predictors from IVS attributes, such that our method can also be used in contexts where there is limited vulnerability data for training.

II RELATED WORKS

N. Jovanovic, C. Kruegel, and E. Kirda [1], have proposed “Pixy: A static analysis tool for detecting web application vulnerabilities,” Pixy is the first open source tool for statically detecting XSS vulnerabilities in php code by means of data flow analysis. A flow-sensitive, interprocedural, and context sensitive edata flow analysis for PHP, targeted detecting taint-style vulnerabilities. This analysis process had to overcome significant conceptual challenges due to the untyped nature of PHP. Additional literal analysis and alias analysis are the steps that lead to more comprehensive and precise results than those provided by previous approaches. Pixy is a system that implements the proposed analysis technique, written in Java and licensed under the GPL. A straightforward approach to solving the problem of detecting taint-style vulnerabilities would be to immediately conduct a *taint analysis* on the intermediate three-address code representation generated by the front-end. This taint analysis would identify points where tainted data can enter the program, propagate taint values along assignments and similar constructs, and inform the user of every sensitive sink that receives tainted input. also perform an *alias analysis* for providing information about alias relationships. Moreover, it is very beneficial for the taint analysis to know about the literal values that variables and constants may hold at each program point. This task is performed by literal analysis.

Y. Xie and A. Aiken [2], “Static detection of security vulnerabilities in scripting languages,” In this approach, static analysis is applied to finding security vulnerabilities in PHP. The goal is to develop a bug detection tool that automatically finds serious vulnerabilities with high confidence. An interprocedural static analysis algorithm for PHP is proposed. A language as dynamic as PHP presents unique challenges for static analysis: language constructs that allow dynamic inclusion of program code, variables whose types change during execution, operations with semantics that depend on the runtime types of the operands, and pervasive use of hash tables and regular expression matching are just some features that must be modelled well to produce useful results. Proposed static analysis algorithm is used to find SQL injection vulnerabilities. Once configured, the analysis is fully automatic. Although we focus on SQL injections in this system, the same techniques can be applied to detecting other vulnerabilities such as cross site scripting (XSS) and code injection in web applications. We parse the PHP source code into abstract syntax trees (ASTs). The parser is based on the standard open source implementation of PHP 5.0.5. Each PHP source file contains a main section and zero or more user defined functions. We store the user-defined functions in the environment and start the analysis from the main function. For each function in the program, the analysis performs a standard

conversion from the abstract syntax tree (AST) of the function body into a control flow graph (CFG). The nodes of the CFG are basic blocks: maximal single entry, single exit sequences of statements. The edges of the CFG are the jump 6 relationships between blocks. For conditional jumps, the corresponding CFG edge is labelled with the branch predicate. Each basic block is simulated using symbolic execution. The goal is to understand the collective effects of statements in a block on the global state of the program and summarize their effects into a concise *block summary*. After computing a summary for each basic block, we use a standard reachability analysis to combine block summaries into a *function summary*. The function summary describes the pre- and post conditions of a function.

D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna [3], "Saner: Composing static and dynamic analysis to validate sanitization in web applications,"A novel approach to analyze the correctness of the sanitization process is introduced. The approach combines two complementary techniques to model the sanitization process and to verify its thoroughness. More precisely, this is the first technique based on static analysis models how an application modifies its inputs along the paths to a sink, using precise modelling of string manipulation routines. This approach uses a conservative model of string operations, which might lead to false positives. Therefore, a second technique based on dynamic analysis is devised. This approach works bottom-up from the sinks and reconstructs the code used by the application to modify the inputs. The code is then executed, using a large set of malicious input values to identify exploitable flaws in the sanitization process. In this approach, a static analysis technique is used that characterizes the sanitization process by modeling the way in which an application processes input values. This helps to identify cases where the sanitization is incorrect or incomplete. A dynamic analysis technique is introduced, that is able to reconstruct the code that is responsible for the sanitization of application inputs, and then execute this code on malicious inputs to identify faulty sanitization procedures. By composing the two techniques to leverage their advantages and mitigate their disadvantages. We implemented this approach and evaluated the system on a set of real-world applications. In the process, a number of previously unknown vulnerabilities in the sanitization routines of the analyzed programs are identified.

L. K. Shar and H. B. K. Tan [4], "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns," An application that accesses database via a SQL language is vulnerable if an unrestricted input is used to build the query string because an attacker might craft the input value to have unauthorized access to the database and perform malicious actions. This security issue is called SQLI vulnerability. An application that sends HTTP response data to a web client is vulnerable if an unrestricted input is included in the response data because an attacker might inject a malicious JavaScript code in the input value. The injected code when executed by the client's browser could perform malicious actions to the client. This security issue is called XSS vulnerability. Web developers generally implement input sanitization schemes to prevent these two vulnerabilities. Input sanitization code attributes which can be statically collected. From these attributes, we aim to build SQLI and XSS vulnerability predictors which provide high recalls and low false alarm rates so that the predictors can be used alternatively or in combination with existing taint-based approaches. Compared to current vulnerability prediction approaches, we only use static code attributes and we target vulnerable code at statement level.

IJSER

This proposal could be easily extended to address other web application vulnerabilities such as buffer overflow, path traversal, and URL redirects/forwards. These vulnerabilities are caused by the common weakness of web applications in handling user inputs properly. Classifiers as the base data miners for building vulnerability prediction models are used. Based on different characteristics of classification algorithms, classifiers can be grouped into different categories such as tree-based approaches, neural networks, support vector machines, nearest-neighbor approaches, statistical procedures, and ensembles.

E. Arisholm, L. C. Briand, and E. B. Johannessen [5], "A systematic and comprehensive investigation of methods to build and evaluate fault prediction models," This paper describes a study performed in an industrial setting that attempts to build predictive models to identify parts of a Java system with a high fault probability. The system under consideration is constantly evolving as several releases a year are shipped to customers. Developers usually have limited resources for their testing and would like to devote extra resources to faulty system parts. The main research focus of this paper is to systematically assess three aspects on how to build and evaluate fault-proneness models in the context of this large Java legacy system development project: (1) compare many data mining and machine learning techniques to build fault-proneness models, (2) assess the impact of using different metric sets such as source code structural measures and change/fault history (process measures), and (3) compare several alternative ways of assessing the performance of the models, in terms of (i) confusion matrix criteria such as accuracy and precision/recall, (ii) ranking ability, using the receiver operating characteristic area (ROC), and (iii) our proposed cost-effectiveness measure (CE).

III EXISTING SYSTEM

SQL injection (SQLI), cross site scripting (XSS), remote code execution (RCE), and file inclusion (FI) are among the most common and serious web application vulnerabilities threatening the privacy and security of both clients and applications nowadays. From the perspective of web developers, input validation and input sanitization are two secure coding techniques that they can adopt to protect their programs from such common vulnerabilities. Input validation typically checks an input against required properties like data length, range, type, and sign. Input sanitization, in general, cleanses an input string by accepting only pre-defined characters and rejecting others, including characters with special meaning to the interpreter under consideration. Intuitively, an application is vulnerable if the developers failed to implement these techniques correctly or to a sufficient degree. To address these security threats, many web vulnerability detection approaches, such as static taint analysis, dynamic taint analysis, modeling checking, symbolic and concolic testing, have been proposed. Static taint analysis approaches are scalable in general but are ineffective in practice due to high false positive rates. Dynamic taint analysis, model checking, symbolic and concolic testing techniques can be highly accurate as they are able to generate real attack values, but have scalability issues for large systems due to path explosion problem. There are also scalable vulnerability prediction methods. But the granularity of current prediction approaches is coarse-grained: they identify vulnerabilities at the level of software modules or components.

IV PROPOSED SYSTEM

Input validation and input sanitization are two secure coding techniques that they can adopt to protect their programs from such common

vulnerabilities. Input validation typically checks an input against required properties like data length, range, type, and sign. Input sanitization, in general, cleanses an input string by accepting only pre-defined characters and rejecting others, including characters with special meaning to the interpreter under consideration. Intuitively, an application is vulnerable if the developers failed to implement these techniques correctly or to a sufficient degree. We hypothesize that code attributes that characterize validation and sanitization code implemented in the program could be used to predict web application vulnerabilities. Based on this hypothesis, we propose a set of code attributes called input validation and sanitization (IVS) attributes from which we build vulnerability predictors that are fine-grained, accurate, and scalable. The approach is fine-grained because it identifies vulnerabilities at program statement levels. We use both static and dynamic program analysis techniques to extract IVS attributes. Static analysis can help assess general properties of a program. Yet, dynamic analysis can focus on more specific code characteristics that are complementary to the information obtained with static analysis. We used dynamic analysis only to infer the possible types of input validation and sanitization code, rather than to precisely prove their correctness, and apply machine learning on these inferences for vulnerability prediction. Therefore, we mitigate the scalability issue typically associated with dynamic analysis. Thus, our proposed IVS attributes reflect relevant properties of the implementations of input validation and input sanitization methods in web programs and are expected to help predict vulnerabilities in an accurate and scalable manner. Furthermore, we use both supervised learning and semi-supervised learning methods to build vulnerability predictors from IVS attributes, such that our method can also be used in contexts where there is limited vulnerability data for training.

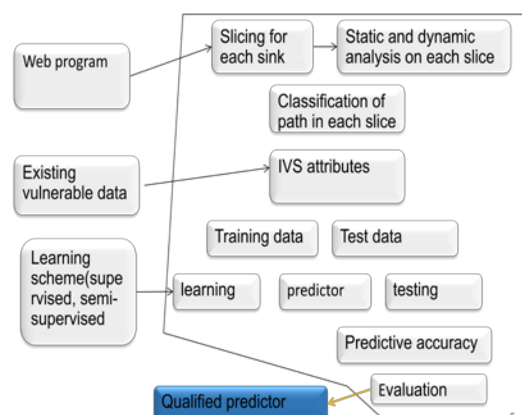


Fig. 4.1 Proposed System Diagram

DIFFERENT MODULES IN THE PROJECT:

1. Static and dynamic program analysis
2. Backward slicing
3. Hybrid program analysis
4. Slicing of each sink
5. Static and dynamic analysis on each slice
6. Classification of path in each slice
7. IVS attributes
8. Building vulnerability prediction model
   A. Data representation
   B. Data processing
9. Supervised learning
10. Semi-supervised learning
11. Final predictor

1. STATIC AND DYNAMIC PROGRAM ANALYSIS

Both static and dynamic program analysis techniques are used to extract IVS attributes. Static analysis can help assess general properties of a program. Yet, dynamic analysis can focus on more specific code characteristics that are complementary to the information obtained with static analysis. The dynamic analysis is used only to infer the possible types of input validation and sanitization code, rather than to precisely prove their correctness.

2. BACKWARD SLICING

Program slicing is a program analysis and transformation technique to decompose programs by analyzing their

data and control flow. Given an imperative program, a slice is an executable program whose behavior must be identical to the specialized subset of the original program's behavior. A program slice consists of those program statements which are (potentially) related to the values computed at some program point and/or variable, referred to as a slicing criterion.
.

3. HYBRID PROGRAM ANALYSIS

The analysis is based on the control flow graph (CFG), the program dependence graph (PDG), and the system dependence graph (SDG) of a web application program. Each node in the graphs represents one source code statement. We may therefore use program statement and node interchangeably depending on the context. A sink is a node in a CFG that uses variables defined from input sources and thus, may be vulnerable to input manipulation attacks. This allows us to predict vulnerabilities at statement levels. Input nodes are the nodes at which data from the external environment are accessed. A variable is tainted if it is defined from input nodes. As described earlier, the first step the approach is to compute a backward static program slice for each sink and the set of tainted variables used. Backward static slice with respect to slicing criterion consists of all nodes (including predicates) in the CFG that may affect the values of, subset of variables are used. We first construct the PDG for the main method of a web application program and also construct PDGs for the methods called from the main method according to the algorithm given by Ferrante et al. We then construct the SDG. A PDG models a program procedure as a graph in which the nodes represent program statements and the edges represent data or control dependences between statements. SDG extends PDG by modeling interprocedural relations between the main program and its subprograms.

4. SLICING OF EACH SINK

Program slicing is method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduce that program to a minimal form which still produces that behavior. The first step of our approach is to compute a backward static program slice for each sink and the set of tainted variable used in the sink.

5. STATIC AND DYNAMIC ANALYSIS ON EACH SLICE

The developers will implement adequate input validation and sanitization methods but yet, they may fail to recognize all the data that could be manipulated by external users, thereby missing some of the inputs for validation. Therefore, in security analysis, it is important to first identify all the input sources. The reason for classifying the inputs into different types is that each class of inputs causes different types of vulnerabilities and different security defense schemes may be required to secure these different classes of inputs.

6. CLASSIFICATION OF PATH IN EACH SCLICE

For each sink, a backward static program slice is computed with respect to the sink statement and the variables used in the sinks. Each path in the slice is analyzed using hybrid (static and dynamic) analysis to extract its validation and sanitization effects on those variables. The path is then classified according to its input validation and sanitization effects inferred by the hybrid analysis.

7. IVS ATTRIBUTES

These attributes characterize various types of program functions and operations that are commonly used as input validation and sanitization procedures to defend against web application vulnerabilities.

Using these attributes, functions and operations are classified according to their security-related properties. Hybrid analysis-based attributes are attributes to be extracted combining static analysis and dynamic analysis. The reason for including input sources in our classification scheme is that most of the common vulnerabilities arise from the misidentification of inputs. That is, developers may implement adequate input validation and sanitization methods but yet, they may fail to recognize all the data that could be manipulated by external users, thereby missing some of the inputs for validation. Therefore, in security analysis, it is important to first identify all the input sources.

This hybrid analysis-based classification is applied for validation and sanitization methods implemented using both standard security functions and nonstandard security functions. If there are only standard security functions to be classified, we classify them based on their security-related information else dynamic analysis is used. Various input validation and sanitization processes may be implemented using language built-in functions and/or custom functions. Since inputs to web applications are naturally strings, string replacement/ matching functions or string manipulation procedures like escaping are generally used to implement custom input validation and sanitization procedures. A good security function generally consists of a set of string functions that accept safe strings or reject unsafe strings. These functions are clearly important indicators of vulnerabilities, but we 19 need to analyze the purpose of each validation and sanitization function since different defense methods are generally required to prevent different types of vulnerabilities. It is important to classify these methods implemented in a program path into different types because, together with their associated vulnerability data, our vulnerability predictors can learn this information and then predict future vulnerabilities.

8. BUILDING VULNERABILITY PREDICTION MODEL

Many machine learning techniques can be used to build vulnerability predictors. Regardless of the specific technique used, the goal is to learn and generalize patterns in the data associated with sinks, which can then be efficiently used for predicting vulnerability for new sinks. As more sophisticated security attacks are being discovered, it is important for a vulnerability analysis approach to be able to adapt. With machine learning, it is possible to adapt to new vulnerability patterns via re-training.

A. DATA REPRESENTATION

Our unit of measurement, an instance in machine learning terminology, is a path in the slice of a sink and we characterize each path with IVS attributes. The attribute values may range from zero to an upper bound that depends on the number of classified program operations or functions. Since 33 IVS attributes are proposed, each path would be represented by a 33-dimensional attribute vector.

B. DATA PROCESSING

In most of our datasets, the proportion of vulnerable sinks to non-vulnerable ones is small. This is an imbalanced data problem and should be expected in many such vulnerability datasets. Prior studies have shown that imbalanced data can significantly affect the performance of machine learning classifiers, because some of the data might go unlearned by the classifier due to their lack of representation, thus leading to induction rules which tend to explain the majority class data and favouring its predictive accuracy. Since for our problem, the minority class data capture the ‘vulnerable’ instances, we need a high predictive accuracy for this class as missing vulnerability is far more critical than

reporting a false alarm. To address this problem, we use a sampling method called adaptive synthetic oversampling. It balances the (unbalanced) data by generating synthetic, artificial data for the minority class instances, thus reducing the bias introduced by the class imbalance problem. It does not require modification of standard classifiers and thus, can be conveniently added as an additional data pre-processing step.

9. SUPERVISED LEARNING

Classification is a type of supervised learning methods because the class label of each training instance has to be provided. In this study, we build logistic regression and Random Forest (RF) models from the proposed attributes. LR is a type of statistical classification model. It can be used for predicting the outcome (class label) of a dependent attribute based on one or more predictor attributes. The probabilities describing the possible outcomes of a given instance are modelled. Logistic regression analysis is flexible in terms of the types of monotonic relationships it can model between the probability of vulnerability and predictor attributes. RF is an ensemble learning method for classification that consists of a collection of tree-structured classifiers. In many cases the predictive accuracy is greatly enhanced as the final prediction output comes from an ensemble of learners, rather than a single learner. Given an input sample, each tree casts a vote (classification) and the forest outputs the classification having the majority vote from the trees.

10. SEMI-SUERVISED LEARNING

As ensemble learning works by combining individual classifiers, it typically requires significant amounts of labeled data for training. In certain industrial contexts, relevant and labeled data available for learning may be limited. Semi-supervised methods [39] use, for training, a small amount of labeled data together with a much larger amount of unlabeled data. This method that exploits unlabeled data can enable ensemble learning when there are very few labeled data. Combining semi supervised learning with ensembles has many advantages. Unlabeled data is exploited to help enrich labeled training samples allowing ensemble learning: Each individual learners improved with unlabeled data labeled by the ensemble consisting of all other learners. A few different types of semi-supervised methods, such as EMbased, clustering-based, and disagreement-based learning, have been proposed in literature. But none of these techniques has been explored for vulnerability prediction so far. Hence, based on these motivations, we explore the use of an algorithm called Co Forest, Co-trained Random Forest (CF), which applies semi-supervised learning on RF. It is a disagreement-based, semi-supervised learner. CF uses multiple, diverse learners, and combines them to exploit unlabeled data (semi supervised learning), and maintains a large disagreement between the learners to promote the learning process.

11. FINAL PREDICTOR

A qualified web application vulnerability predictor can be built with the help of the input validation and sanitation attributes and the machine learning techniques. By using the above attributes we will be able to generate a web application predictor which is highly accurate, fine-grained and scalable.

V IMPLEMENTATION

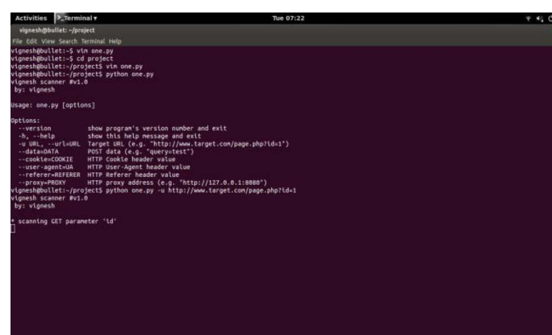
A. DERIVATION OF IVS ATTRIBUTES

The code attributes that characterize validation and sanitization code implemented in the program could be used to predict web application vulnerabilities. Based on this hypothesis, we propose a set of code attributes called input validation and sanitization (IVS) attributes from which we build vulnerability predictors that are fine-grained, accurate, and scalable. The

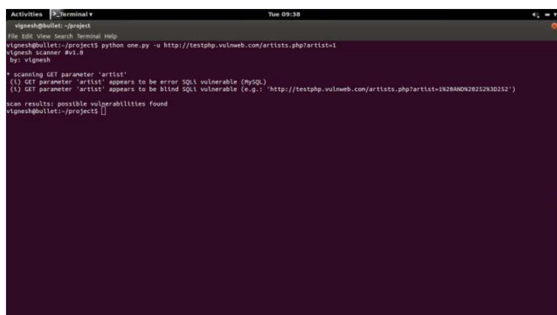
approach is fine-grained because it identifies vulnerabilities at program statement levels. We use both static and dynamic program analysis techniques to extract IVS attributes. Static analysis can help assess general properties of a program. Yet, dynamic analysis can focus on more specific code characteristics that are complementary to the information obtained with static analysis. We use dynamic analysis only to infer the possible types of input validation and sanitization code, rather than to precisely prove their correctness.

B. LIST OF IVS ATTRIBUTES

1. Client - Input accessed from HTTP request parameters such as HTTP Get
2. File - Input accessed from files such as Cookies, XML
3. Text-database - Text-based input accessed from database
4. Numeric-database - Numeric-based input accessed from database
5. Session - Input accessed from persistent data object such as HTTP Session
6. Uninit - Un-initialized program variable
7. Un-taint - Function that returns predefined information or information not influenced by external users.
8. Known-vuln-user - Custom function that has caused security issues in the past.
9. Known-vuln-std - Language built-in function that has caused security issues in the past.
10. Propagate - Function or operation that propagates partial or complete value of a string.
11. Numeric Function or operation that converts a string into a numeric
12. DB-operator Function that filters query operators such as (=)
13. DB-comment-delimiter Function that filters query comment delimiters such as (–)
14. DB-special Function that filters other database special characters different from the above, such as (\x00) and (\x1a)
15. String-delimiter Function that filters string delimiters such as (‘) and (“)
16. Lang-comment-delimiter Function that filters programming language comment delimiter characters such as (/)
17. Other-delimiter Function that filters other delimiters different from the above delimiters such as (#)
18. Script-tag Function that filters dynamic client script tags such as (<script>)
19. HTML-tag Function that filters static client script tags such as (<div>)
20. Event-handler Function that disallow the use of inputs as the values of client side event handlers such as (onload =)
21. Null-byte Function that filters null byte (%00)
22. Dot Function that filters dot (.)
23. DotDotSlash Function that filters dot-dot-slash (../) sequences
24. Backslash Function that filters backslash (\)
25. Slash Function that filters slash (/)
26. Newline Function that filters newline (\n)
27. Colon Function that filters colon (,) or semi-colon (;)
28. Other-special Function that filters any other special characters different from the above
29. Encode Function that encodes a string into a different format
30. Canonicalize Function that converts a string into its most standard, simplest form
31. Path Function that filters directory paths or URLs
32. Limit-length Function or operation that limits a string into a specific length



5.1 Implementation of vulnerability predictor



5.2 Implementation of vulnerability predictor

VI CONCLUSION

In this project, the input validation and sanitation attributes are generated. The first step of our approach is to compute a static backward slice for each sink. Both the static program analysis and dynamic program analysis are used to extract the input validation and sanitation attributes. The program analysis is based on the control flow graph, control dependence graph and system dependence graph of a web program. The input validation and sanitation attributes will act as the building blocks for the web application vulnerability predictor.

IJSER

REFERENCES

[1]. N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities,” in Proc. IEEE Symp. Security Privacy, 2006, pp. 258–263.

[2]. Y. Xie and A. Aiken, “Static detection of security vulnerabilities in scripting languages,” in Proc. USENIX Security Symp., 2006, pp. 179–192.

[3]. D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Saner: Composing static and dynamic analysis to validate sanitization in web applications,” in Proc. IEEE Symp. Security Privacy, 2008, pp. 387–401.

[4]. L. K. Shar and H. B. K. Tan, “Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns,” Inf. Softw. Technol., vol. 55, no. 10, pp. 1767– 1780, 2013.

[5]. E. Arisholm, L. C. Briand, and E. B. Johannessen, “A systematic and comprehensive investigation of methods to build and evaluate fault prediction models,” J. Syst. Softw., vol. 83, no. 1, pp. 2–17, 2010.

[6]. A. Kie_zun, P. J. Guo, K. Jayaraman, and M. D. Ernst, “Automatic creation of SQL injection and cross-site scripting attacks,” in Proc. Int. Conf. Softw. Eng., 2009, pp. 199–209.

[7]. M. Martin and M. S. Lam, “Automatic generation of XSS and SQL injection attacks with goal-directed model checking,” in Pro0c.USENIX Security Symp., 2008, pp. 31–43.