12-2013

Towards a hybrid framework for detecting input manipulation vulnerabilities

Sun DING
Nanyang Technological University

Hee Beng Kuan TAN
Nanyang Technological University

Lwin Khin SHAR
Singapore Management University, lkshar@smu.edu.sg

Bindu Madhavi PADMANABHUNI
Nanyang Technological University

Citation

DING, Sun; TAN, Hee Beng Kuan; SHAR, Lwin Khin; and PADMANABHUNI, Bindu Madhavi. Towards a hybrid framework for detecting input manipulation vulnerabilities. (2013). *2013 20th Asia-Pacific Software Engineering Conference (APSEC): Bangkok, December 2-5: Proceedings*. 363-370. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/4837

Published in Published in 2013 20th Asia-Pacific Software Engineering Conference (APSEC): Bangkok, December 2-5: Proceedings. pp. 363-370.
https://doi.org/10.1109/APSEC.2013.56

Towards a Hybrid Framework for Detecting Input Manipulation Vulnerabilities

Sun Ding
School of Electrical and Electronic Engineering, Nanyang Technological University,Singapore
ding0037@e.ntu.edu.sg

Hee Beng Kuan Tan
School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore
ibktan@e.ntu.edu.sg

Lwin Khin Shar
School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore
shar0035@e.ntu.edu.sg

Bindu Madhavi Padmanabhuni
School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore
padm0010@e.ntu.edu.sg

***Abstract*—Input manipulation vulnerabilities such as SQL Injection, Cross-site scripting, Buffer Overflow vulnerabilities are highly prevalent and pose critical security risks. As a result, many methods have been proposed to apply static analysis, dynamic analysis or a combination of them, to detect such security vulnerabilities. Most of the existing methods classify vulnerabilities into safe and unsafe. They have both false-positive and false-negative cases. In general, security vulnerability can be classified into three cases: (1) provable safe; (2) provable unsafe; (3) unsure. In this paper, we propose a hybrid framework—Detecting Input Manipulation Vulnerabilities (DIMV), to verify the adequacy of security vulnerability defenses for input manipulation vulnerabilities by integrating formal verification with vulnerability prediction in a seamless way. The verification part takes into account sink predicates and effect of domain and custom specifications for detecting input manipulation vulnerabilities. Proving from specification is used as far as possible. Cases that cannot be proved are then predicted from the signatures mined. Our evaluation shows the practicality of the proposed framework.**

Keywords*— Vulnerability detection; framework; formal verification; prediction; data mining; input validation; specification; verification; input manipulation vulnerabilities

I. Introduction

Security vulnerabilities, such as SQL Injection (SQLI), Cross-Site-Scripting (XSS), and Buffer Overflow are mistakes in software that can be directly used by a hacker to gain access to a system or network [1]. They have resulted in enormous losses due to information leakage or customer dissatisfaction.

Security vulnerabilities are usually caused by programming errors or inadequate security defenses. Most of the security attacks are result of code injection from external output where the input data is crafted carefully to be interpreted as code rather than data to achieve devious objectives. Hackers look for presence of vulnerable code to carry out exploits. For example, buffer overflow vulnerabilities occur due to failure to check if the string to be copied is larger than the allocated size of the destination buffer. Format string exploits occur when external inputs are passed to string formatting functions without adequate validation. SQL Injection and cross site scripting attacks are due to failure to validate or sanitize inputs for presence of characters that have special meaning in their respective domain. Path traversal, command injection and HTTP response splitting are few other examples of input manipulation vulnerabilities. Adequate input validation and sanitization is the only way to thoroughly prevent illegal input manipulation in a software system.

In this paper, we propose a hybrid framework integrating formal verification with vulnerability prediction for automated detection of input manipulation vulnerabilities. Formal verification helps in proving or disproving correctness of software expressed in terms of formal specifications by mathematical logic. By specifying security rules as formal specifications for each of security critical statements and modules in a system and verifying them formally we can conclude on the adequacy of security defense implemented in the system. Formal proof classifies software to be “safe” or “unsafe” where it can be proved. But, for cases when either the security property itself cannot be proved inherently or the theorem prover lacks information to prove it from the context supplied to it nothing can be said about their security defense adequacy. To complement this, we propose using vulnerability prediction by mining static code attributes representing input validation and sanitization regimes for such “unsure” cases.

The paper is organized as follows. Section II provides the background relevant for the proposed approach. We present our proposed hybrid framework for detecting input manipulation vulnerabilities in Section III. The approach is evaluated in Section IV. We discuss the works related to ours in Section V and give our conclusions in Section VI.

II. Background

Formal verification and data mining are two kinds of techniques that used in vulnerability detection [2, 18].

On one hand, formal verification uses mathematical proof to prove system design integrity. For vulnerability detection, security assertions are specified and the system is verified to check if the assertions hold on all paths leading to the potential vulnerable statement. The emphasis is on correctness, rigorousness and precision, but huge effort is required for defining formal specifications and the computational model used itself may have some limitations.

On the other hand, formal verification provides qualitative analysis whereas prediction is quantitative. So prediction can be used when qualitative analysis fails to reach a conclusion and the result is not quantifiable. Vulnerability prediction analyses can be categorized into statistical prediction and data mining. Some statistical approaches as in [2] look at version histories and corresponding known vulnerabilities of software for prediction. This deviates from our scope as we don't consider the version histories for detecting vulnerabilities in software. Others use code or execution complexity [3] or lines of code as metrics for prediction. In both cases the purpose is to identify vulnerable components for testing and fixing. Another prediction attempt is to use more comprehensive data mining techniques. Data mining automates and sometimes fastens the detection process but is only as effective as the correctness of distinguishing patterns of safe and vulnerable patterns mined and might generate many false positives and false negatives. Data mining approaches which take into account code semantics and domain knowledge [4, 5] rather than just metrics as used by statistical prediction ones [3] are more suitable for detecting presence and/or adequacy of security defenses in the code. The mined patterns may help in vulnerability location identification which is not possible in statistical prediction using code metrics.

Prediction by data mining complements formal verification by its generalization capabilities and uses patterns mined to classify a new instance. Therefore, a hybrid approach of formal verification and prediction using data mining has the advantages of qualitative and quantitative capabilities for comprehensive vulnerability detection.

III. THE PROPOSED APPROACH

Input manipulation vulnerabilities occur when the program does not manage to validate or sanitize the illegal input. Therefore, to examine the potential vulnerable statement, we can focus on the part of the program that validate or sanitize the external input before it is used to influence the data processed by the potential vulnerable statement. Intuitively, we call the part of program, **input validation and sanitization semi-slice.**

In proposed framework, formal verification will be applied on the software first to classify it as 'safe' or 'unsafe' or 'unsure'. For each potential vulnerable statement that is classified 'unsure', we propose mining its code attributes for predicting vulnerabilities. Formal proof and data mining whole system software of real-life applications is very complex. In order to guide the proving and mining process to only relevant code that is affected by the input we propose to restrict the proving and mining to input validation and sanitization semi-slice since all the statements relevant to input manipulation vulnerability detection are captured in it.

A. Input Validation and Sanitization Semi-Slice and its Extraction

Before proceeding to a more formal introduction of input validation and sanitization semi-slice, we introduce some terms. We shall adopt the formalism of control flow graph from [6].

In a control flow graph (CFG), a node *w* **post-dominates** a node *u* if and only if every path from *u* to the exit node contains *w*. In a CFG, a node *y* is **control dependent on** a node *x* if and only if *x* has successors *x'* and x'' such that *y* post-dominates *x'* but *y* does not post-dominate *x''*. We say that node *y* is transitively control dependent on node *v* if there exists a sequence of nodes, $v_0 = v, v_1, v_2, ..., v_n = y$, in the control flow graph such that $n \geq 1$ and v_j is control dependent on v_{j-1} for all j, $1 \leq j \leq n$. In a CFG, a node *y* is **data dependent on** a node *x* if there exists a variable *v* such that *x* defines *v*, *y* uses *v* and there is a path from *x* to *y* along which *v* is not redefined.

Additionally, we call a variable in a program as an **input variable** if it is not defined in the program solely from constants and variables. Let *v* be an input variable defined at a node *x* in a CFG. A node *y* in the CFG **transitively references** to *v* defined at *x* if there is a path from *x* to *y*, a sequence of nodes, $y_0 = x, y_1, ..., y_n = y$ in the path and a sequence of variables $v_0 = v, v_1, ..., v_n$, such that $n \geq 1$, for each j, $1 \leq j \leq n$, y_j defines the variable v_j and references to variable v_{j-1}, and the sub-path from y_{j-1} to y_j is a definition clear path with respect to v_{j-1}.

A **sensitive sink** *k* is a statement in a program such that the execution of *k* may lead to harmful or wrong operation if the values of variables that *k* references are not restricted properly. Correspondingly, the node that represents a sensitive sink is called a **sensitive sink node**. Let *k* be a sensitive sink node in a CFG. A predicate node *d* in the CFG is called an **input validation node** of *k* if the following properties hold:

i. Both *k* and *d* transitively reference to a common input variable defined at the same node
ii. Node *k* is transitively control dependent on *d*

The branch condition controlling the branch that k is on is called a **validation node constraint**.

A sequence *V* of statements that validate the input processed at *k,* is extracted from *P* according to the following steps:

1. Include *k* in *V*
2. Include all nodes that define variables that are referenced at *k* in *V*
3. Include all input validation nodes of *k* in *V*
4. *P* is iterated from the last statement to the first statement statement-by-statement to extract further statements from *P* in the following way until *V* is invariant upon iteration – that is, until no new statements from *P* is included in an iteration. In each iteration, for each statement $t \neq k$ in *P*, if there is a statement in *V* that is control or data dependent on *t*, then *t* is included in *V*

These statements are included in *V* in the same order as they appear in *P*. We call the sequence *V*, the **input validation and sanitization semi-slice** of *k*.

B. Detecting Input Manipulation Vulnerabilities

In general, for any type of security vulnerability, it is impossible to prove all cases to be either safe or unsafe. To address this problem, we propose a hybrid framework—**D**etecting **I**nput **M**anipulation **V**ulnerabilities (**DIMV**), which integrates formal proof and data mining techniques in a seamless way:

1. First, based on given specifications for certain vulnerabilities, DIMV proves cases to be ‘safe’ or ‘unsafe’ as far as possible.
2. Second, safe and unsafe patterns in terms of code static attributes are then used to predict the ‘unsure’ cases.

Figure 1 gives an overview of the proposed approach. Next, we shall introduce the details of DIMV.

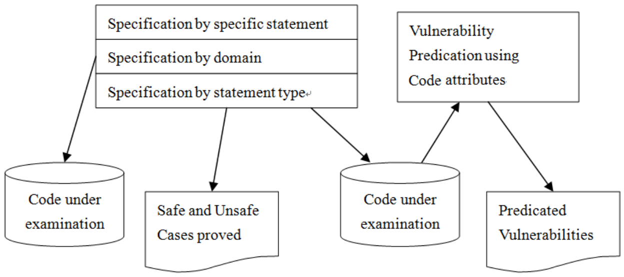

Specification by specific statement
Specification by domain
Specification by statement type
Vulnerability Predication using Code attributes
Code under examination
Safe and Unsafe Cases proved
Code under examination
Predicated Vulnerabilities

Fig. 1. Overview of the Proposed Framework

1) *Vulnerability Dectection from Verification*

One major setback to formally prove required properties for program is the huge effort involved in defining a formal specification. Security vulnerabilities of code occur at some specific types of statements. If in the code, user/external inputs are not properly checked to ensure that the required properties are met before processed by these statements, the inputs might change the original intention of these statements to perform harmful operations. The required properties are invariant for each type of sink statements.

We therefore propose a three-level specification based approach for vulnerability detection. A specification can be specified by:

1. Statement type in general
2. Statement type by domain
3. Specific statement in a program

Each specification attached to a statement at these levels will contain two elements:

- Type of vulnerability with safe and unsafe formal properties specified if there are formal properties available
- Prediction rules or patterns

```
Data Type    T           ::=  int | float | p*| obj
Variable     v           ::=  id: T
Constant     c           ::=  Num | Str
Predicate    predicate   ::=  ==|!=|<|>|≥|≤|contains()
Operation    op          ::=  n_op |s_op|p_op|b_op
             n_op        ::=  +|-|*|/|<<|>>|%
             s_op        ::=  escape()
             p_op        ::=  ++|--|length ()|addr()
             b_op        ::=  AND | OR | NOT
Expression   exp         ::=  v|v_i (op v_j)*
Var_Bound    bound       ::=  (v|exp) predicate  (v|exp|c)
Property     Property    ::=  {bound}
```

Fig. 2. Syntax for Property Specification

The three-leveled approach caters to vulnerability detection centered at language specific, domain specific and application / business logic vulnerabilities. The safe and unsafe formal properties for each type of sensitive sink statement are defined as predicates and stored in a **sink properties database**. The effort for defining these properties is only once. These properties can be refined and expanded and are reusable.

DIMV introduces the syntax in Fig. 2 to define properties. Each property is a set of boundaries which constraint the variables referenced by the corresponding sink. Violating the property may lead to sink nodes exploited. In Fig. 2, besides those regular notations, DIMV is also built on another four important functions:

- *length(v)*: this function returns the length of a given pointer or object.
- *addr(v)*: this function returns the memory address of a given pointer or object.
- *contains*(v_1, v_2): this functions judges whether a string v_1 contains any strings matched by the regular expression pattern v_2.
- *escape (n, v_1)*: this function encodes a given string v_1 by converting certain special characters. DIMV maintains a list of standard escaping functions. The parameter *n* is the ID, set to invoke these functions for different use cases.

Table I illustrates a part of standard functions maintained by DIMV. Table II lists a set of sample predefined properties in DIMV.

Table I. STANDARD ESCAPE FUCNTIONS MAINTAINED BY DIMV

ID	Host Language	Standard Escape Function
1	PHP	htmlentities ()
2	PHP	htmlspecialchars()
3	PHP	mysql_real_escape ()
4	Javascript	escape ()
……		

Table II. SINK PROPERTIES DATABASE IN DIMV

Sensitive Sink	Host Language	Safe Formal Property
Buffer Overflow Vulnerability		
void * memmove(void * dst, const void * src, size_t num)	C/C++	(length (dst) ≥ num) &&(length (src) ≥ num)
void * memset(void * ptr, int value, size_t num)	C/C++	length (ptr) ≥ num
Cross Site Scripting (XSS) Vulnerability		
echo var	PHP	escape (1,var) ==var /* the output content *var* should not contain any special characters*/
SQL Injection Vulnerability		
mysql_query(sql)	PHP	escape (3, sql) ==sql /* the script *sql* should not contain any special characters*/
…….		

DIMV performs the algorithm *CheckVul* shown in Fig. 3 to verify a given program. For each sink node *k*, *CheckVul* gets its safe property from database and computes *k*’s semi-slice *V*. For every variables used by *k*’s safe property, *CheckVul* uses symbolic evaluation over *V* to get these variables' definitions. With symbolic evaluation and constraint solving, if the safe properties can be proved under *k*'s validation constraint, then a definite conclusion on *k*'s vulnerability would be reached. Cases that cannot be proved are highlighted as 'unsure' and are later processed by DIMV's vulnerability predication component.

Take for instance, in Fig. 4 node10 is the sink node. The sink has insufficient validation and is hence unsafe. This can be verified using our general statement specification for *memmove* depicted in Table I. Node9 is the input validation node for node10 as both of them refer to common input variable *lmov* defined at node6. Hence the validation node constraint is $cvn = (lmov \leq MAXSIZE)$. The safe property for this sink is $p =$ (length (*dst*) >= *num*) && (length (*src*) >= *num*). Therefore, a generated constraint ($cvn \rightarrow p$) will be expressed as:

$(lmov \leq MAXSIZE) \rightarrow ((\text{length}(dst) >= num) \&\& (\text{length}(src) >= num))$

After symbolic evaluation, this generated constraint would be expressed in terms of program variables as:

$(lmov \leq MAXSIZE) \rightarrow ((ldst >= lmov) \&\& (MAXSIZE >= lmov))$

The above constraint will be verified against a theorem prover and the result is invalid. Therefore, the sink node node10 is termed as "unsafe", which means it is vulnerable to buffer overflow attack.

```
Input      : $G$
Output     : $result$, // $result$ could '$safe$', '$unsafe$ 'or '$unsure$'
Algorithm: CheckVul ($G$)
begin
  for each sink node $k$ in $G$
    $p$=lookup ($k$); /* search in DIMV's sink properties database to get the
                          property for $k$.*/
    $V$=semi_slice($k$); // get the semi-slice of $k$
    for each input validation node $d$ in $V$
      get $d$'s validation node constraint $cvn$
      $C = cvn \rightarrow p$; // get an implication constraint
    end for
    for each variable $\delta$ used in $C$
      $\eta$ =symbolic_evaluate($\delta$, $V$);/* do symbolic evaluation over $V$ to get
            $\delta$'s definition */
      $C = C \wedge \eta$; // update $C$ with variables' definitions
    end for
    $r$ = constraint_solve($C$); //verify $C$ against a  theorem prover
    record ($r$, $k$, $G$);         //  record the analysis result
  end for
  return $r$;
end
```

Fig. 3. Vulnerability Verification Algorithm

```
    #define MAXSIZE 40
    int main() {
1.      char src [MAXSIZE];
2.      printf ("Enter string:\n");
3.      scanf ("%39s",src);

4.      size_t  ldst = 0;
5.      scanf("%d", &ldst);

6.      size_t  lmove = 0;
7.      scanf("%d", &lmov);

8.      char * dst = (char *) malloc (ldst);

9.      if(lmov <= MAXSIZE)                        //input validation node
10.         memmove(dst, src, lmov);               // sink node
    }
```

Fig. 4. Sample Code Violating a Sink's Safe Property

At times, the specification framework may not be able to prove or disprove the sink's safe property. This could be due to inadequate information for such verification condition generation or due to absence of validation node for the sink. Based on our previous work [4, 5], quite a number of 'unsure' cases are caused by functions with nondeterministic behaviors. To reduce the effect from these functions, DIMV maintains a **function effect database** for users to store predefined mock objects and stub functions. This database is created either on organization or project basis and can be extended as and when required.

However, even with the above effort, in certain cases, a definite outcome still cannot be reached and the sink will hence be declared 'unsure' and will be predicted using classifier models learned from code attributes collected from benchmarks with known vulnerabilities.

2) Vulnerability Dectection from Prediction

We propose using code attributes for predicting cases declared 'unsure' by formal verification component. This happens at times safe or unsafe sink properties cannot be proved as the theorem prover lacks information to prove sink properties from the system's context. Hence we complement the formal verification with prediction to detect probable vulnerabilities by taking into account the code and domain semantics. In order to predict the vulnerability of an application we propose using data mining schemes appropriate to the domain/framework under consideration. Patterns in software represent programming rules or coding style. Capturing such patterns/features ideal for distinguishing between safe and vulnerable software is the key to predict vulnerability with higher probability of accuracy when encountered with new software specimen. Our previous work in [2] shows promising results in predicting the SQL Injection and Cross site scripting (XSS) vulnerabilities from code attributes. This gives a preliminary justification of the proposed framework.

An application is vulnerable if the input is not validated properly and/or if the input sanitization regime required as needed for respective sink is either absent or is inadequate. Input can be validated implicitly by checking variable's properties (like length, range, type, sign or in case of SQL injection and cross-site scripting attacks for presence of characters with special meaning to domain under consideration) or it can be performed explicitly by invoking input sanitization schemes provided by the development framework or custom written routines.

We suggest mining code attributes representing sink types (such as database functions, system functions, HTML output functions, string manipulation functions), input types (HTML forms, environment variables, data read from data base or files, command line inputs etc), input validation and sanitization routines implemented for the sink. Code attributes that characterize program code patterns that are sensitive to the particular security issue under consideration should also be used. For example for buffer overflow caused by format string, a code attribute "if the format string is data dependent on an input value", would come under such vulnerability relevant attribute.

Vulnerability prediction is done by training prediction models with attributes representing domain and code semantics from benchmarks with known vulnerabilities and using it to predict vulnerabilities on new software program specimens.

IV. Evaluation: A Case Study

Many approaches have been proposed for formal verification and prediction of vulnerability separately. As this paper proposes a framework for integrating verification and prediction approaches together in a seamless way, our evaluation is based on examining the integration of one approach from formal verification and another one from prediction through a case study. This case study is using DIMV to examine XSS vulnerability in PHP programs.

Table IV. STATISTICS OF THE TEST SUBJECTS

Test Subject	**Description**	**LOC**	**Security Advisories**
Schoolmate 1.5.4	A tool for school administration	8145	Vulnerability info in [29]
Faqforge 1.3.2	Document creation and management	2238	Bugtrag-43897
Utopia News Pro1.1.4	News management System	5737	Bugtrag-15027
Phorum 5.2.18	Message board software	12324	CVE-2008-1486 CVE-2011-4561
Cutesite 1.2.3	Content management framework	11441	CVE-2010-5024 CVE-2010-5025
Myadmin 3.4.4	MySQL database management	44628	PMASA-2011-14 PMASA-2011-20

Table V. DATA STATISTICS OF THE TEST SUBJECTS

Data Set	**#HTML sinks**	**#Actual Vul. Sinks to XSS**	**#Principal Components**
schoolmate-html	172	138	7
faqforge-html	115	53	7
utopia-html	86	17	9
phorum-html	237	9	9
cutesite-html	239	40	10
myadmin-html	305	20	9

A. *Experiment Setting*

Let us recall Fig. 1. Test subjects will be fed to DIMV and experienced a two-stage checking: vulnerability verification and vulnerability prediction.

- The first stage would capture cases with definite conclusions. We implemented the symbolic evaluation with Pixy [7] — a PHP analysis program. We used Microsoft Z3 [8] to solve numeric constraints; and used HAMPI [9] to solve string constraints.

- The second stage would predict the vulnerability of those 'unsure' cases. We also used Pixy to collect required code attributes. Table III (appendix) shows the attributes we suggested for XSS vulnerability prediction.

The experiment was conducted over six real-world PHP-based web applications obtained from SourceForge [10]. Table IV shows the relevant statistics for these test subjects. To better evaluate DIMV, we carried out a preliminary study with manual effort to find out the actual vulnerability statistics in each systems. Table V shows such data.

B. *Result of Vulnerability Verification*

Table VI shows the verification result. The Column *Actual Sinks* records the confirmed vulnerabilities with each data set. The Columns *Safe*, *Unsafe* and *Unsure* record the number of detected 'safe' and 'unsafe' cases and also the number of 'unsure' cases. In the current implementation, we set a timeout as 1 minute for verifying each sink, including both symbolic evaluation and constraint solving. If the timer exceeded, that case will be concluded as 'unsure'. This setting is to capture cases beyond theorem provers' solvability.

As shown by Table VI, within the total 1154 Sinks, 801 (69.41%) sinks are detected with definite conclusions while the rest 353(30.59%) cases fell in unsolvable situations.

Table VI. STATISTICS OF THE TEST SUBJECTS

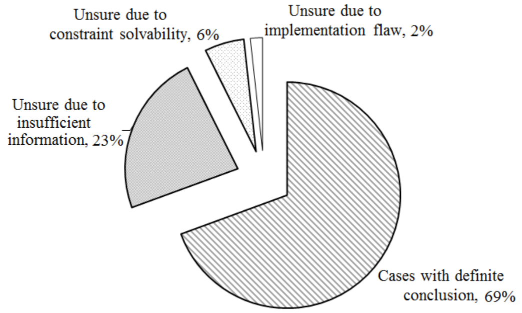
Data Set	**#HTML sinks**	**#Detectable**		**#Unsure**
		#Safe	**#Unsafe**	
schoolmate-html	172	26	95	51
faqforge-html	115	47	35	33
utopia-html	86	34	14	38
phorum-html	237	159	6	72
cutesite-html	239	153	19	67
myadmin-html	305	202	16	87
Total	1154	613	188	353(30.59%)
		801 (69.41%)		

We investigated the 'unsure' cases and discovered the unsolvable situations mainly include:

- Insufficient information in DIMV's function effect database. For example, when encountered native functions which are without function effect database, DIMV will conclude the case as 'unsure'. There are 268 of such cases.

- Complex string constraint. Solving string constraints is expensive and time consuming. When DIMV run out of time, the case will be concluded as 'unsure'. There are 65 of such cases.

- Implementation flaw. A precise and scalable symbolic evaluation procedure requires decent engineering effort. The current implementation of DIMV may generate a small amount of runtime exceptions. There are 20 of such cases.



- Fig. 5. Cases Encountered in DIMV Verification Stage

Figure 5 concludes the above finds. These difficulties are the bottlenecks for many existing formal verification methods. To overcome such bottlenecks, DIMV further processed these unsure cases with data mining techniques.

C. Result of Vulnerability Prediction

Prediction Measure: The goal of this experiment is to test whether it is possible to predict vulnerability for those 'unsure' cases. In the current implementation of DIMV, the prediction is performed by supervised classification. To measure the performance with standard metrics, we followed Table VII to compute recall of detection (*pd*), probability of false alarm (*pf*), and precision (*pr*).

Recall (*pd*) measures a classifier's ability in finding actual vulnerable sinks. Precision (*pf*) measures the actual vulnerable sinks that are correctly predicted in terms of a percentage of total number of sinks predicted as vulnerable. False alarm (*pd*) measures cost of using the classifier: an increasing *pd* indicates more false alarms or decreasing precision. Ideally, the classifier should neither miss any actual vulnerabilities (*pf* ~ 1) nor throw false alarms (*pf* ~ 0 , *pr* ~ 1).

Table VII. PERFORMANCE MEASUREMENT

		Actual	
		Vulnerable	Not-Vulnerable
Predicted	Vulnerable	True positive(*tp*)	False positive(*fp*)
	Not-Vulnerable	False negative(*fn*)	True negative(*tn*)
Recall	***pd*** = *tp* / (*tp*+*fn*)		
False alarm	***pf*** = *fp* / (*fp*+*tn*)		
Precision	***pr*** = *tp* / (*tp*+*fp*)		

Data Preprocessing: Let us recall the verification result in the sub section B. The verification stage confirmed 613 cases as 'safe' and 188 cases as 'unsafe'. Guided by Table III (appendix), we collected the attributes related with these (613+188=801) cases as the sample data to train classifiers.

Thereafter, we obtained data of 20 numeric attributes and 2 binary attributes. From our preliminary study, it is observed that different numeric attributes are defined on different scales and their distributions are highly skewed. This may cause bias toward to some attributes (e.g. attributes with large scale values), especially in the context of clustering where similarity measurement combines multiple attributes scales. To overcome this problem, we used a *min-max* method [11] to normalize the collected data.

After normalization, we further processed the data with Principle Component Analysis (PCA). PCA results in a new set of attributes (principle components), each of which is a linear combination of some the original attributes. PCA helps eliminate the attributes' inter-dependency. The new attributes set is usually much smaller, and therefore could be more efficiently analyzed by classifiers. In our experiments, we applied PCA to every data set (after min-max normalization) and used a subset of principal components as attributes such that the selected explain at least 95% of the data variance. The last column in Table V shows the numbers of principal components selected

Classifiers: we used supervised learning methods and chose two efficient classifiers for this experiment: Logistic Regression (LR) and Multi-Layer Perceptron (MLP). These classifiers were benchmarked as among the top classifiers in recent studies [12]. MLP is a type of neural networks. LR is a type of statistical regression models. Details about these classification techniques are provided by Witten and Frank [11]. We used two very different techniques in an attempt to optimize accuracy. The implementation of the two classifiers are from Weka [13], a data mining toolkit box.

Validation and Training: We first validated the classifiers with 10-fold cross validation setup. The data is divided into ten sets. A classifier is trained on nine sets and then tested on the remaining set. This process is repeated ten times; each time testing on a different set. The order of training and test set is randomized. This test design overcomes the ordering effects due to randomization. This is important to avoid a malignant increase in performance by a certain ordering of training and test data. Isolating a test set from the training set also conforms to hold-out test design which is important to evaluate the classifier' capability to predict new vulnerabilities [11].

Table VIII shows the validation result. On average, the two classifier both showed good performances with high detection rate(LP=86%, MLP=78%) low false alarm rate (LP=3%, MLP=3%). But on data set *phorum-html*, MLP could not identify certain cases whereas LP is more stable. Therefore, we chose LP to build a predictor to predict vulnerabilities.

Table VIII VALIDATION RESULT OF VULNERBILITY PREDICTOR

Data Set	Classifier	*pd*	*pf*	*pr*
schoolmate-html	LP	99	3	98
	MLP	99	0	100
faqforge-html	LP	89	5	94
	MLP	91	5	94
utopia-html	LP	94	1	94
	MLP	94	2	89
phorum-html	LP	78	1	70
	MLP	33	0	100
cutesite-html	LP	68	9	61
	MLP	78	8	67
myadmin-html	LP	85	1	89
	MLP	75	1	83
Average results on XSS prediction	**LP**	**86**	**3**	**84**
	MLP	**78**	**3**	**89**

Vulnerability Prediction: we used the built predictor to predict vulnerabilities for those 'unsure' cases. The results are shown in Table XI. The column *#unsure sinks* records the number of 'unsure' sinks. The column *#Predicted* records the number of predicted vulnerable sinks. The column *#Correct* records the number of actual vulnerable sinks. The last two columns *#FP* and *#FN* report the number of false positives and false negatives. Therefore, the prediction stage captures another 79 vulnerable sinks with a false positive rate at 14.44% and a false negative rate at 2.22%. So we can conclude that the performance of the vulnerability prediction is good.

Table XI VULNERABILITY PREDICTION

Data Set	**#unsure sinks**	**#Vulnerable sinks**			
		#Predicted	**#Correct**	**#FP**	**#FN**
schoolmate	51	36	42	7	1
faqforge	33	15	17	3	1
utopia	38	3	3	0	0
phorum	72	2	3	1	0
cutesite	67	19	21	2	0
myadmin	87	4	4	0	0
Total	348	79	90	13 (14.44%)	2 (2.22%)

V. RELATED WORK

Security vulnerabilities may result in great loss due to system failure or information leakage. A vast number of proposals have been made to mitigate the threats of vulnerabilities. Existing methods or tools could be mainly categorized into four types: (1) vulnerability detection with program analysis; (2) runtime attack prevention; (3) vulnerability prediction with data mining; (4) hybrid framework.

Methods of program analysis focus on detecting vulnerabilities in source code or scripts using taint analysis techniques. Earlier methods usually uses static analysis to identify tainted inputs received from external data sources, track the dataflow of tainted data, and check if any reached sinks such as buffer writing, SQL or HTML output statements [14, 15]. Recently, dynamic analysis techniques are integrated to enhance the detection precision. For example, a team led by Adam Kiezun used *concolic* (concrete+symbolic) *execution* to capture program path constraints and a constraint solver to generate test inputs that explored various program paths [16]. Upon reaching the sinks, they exercised two sets of inputs—one of ordinary valid strings and the other of attack strings from a library [17]—and checked the differences between the resulting program behaviors.

Methods of runtime attack prevention adopt run-time defense to prevent exploiting potential vulnerabilities of any installed programs. These approaches aim to provide an extra protection regardless of what the source program is. For instance, StakeGuard [18] is such a tool that creates virtual variables to simulate function's return address to prevent buffer overflow attacks. Suspicious code will be redirected to act over the virtual variables first.

Methods of vulnerability prediction belong to a branch that attempts to reveal the association between vulnerabilities and certain program attributes. A classic prediction model is from Shin et al. [19]. They used code complexity, code churn, and developer activity attributes to predict vulnerable programs. Their assumption was that, the more complex the code, the higher the chances of vulnerability. Recent novel approaches also include [2, 4, 5, 20].

In practice, it is found that a standalone approach from any of the above categories cannot fully resolve the threads of various vulnerabilities. Therefore, a number of hybrid vulnerability detection frameworks have been proposed [18-19]. Some of these for vulnerability detection approaches integrate more than one analysis for detection purposes. Most of the vulnerability detection proposals, in general, classify vulnerabilities as safe or unsafe, there by incurring false positives and false negatives. By contrast, we classify them as 'safe' or 'unsafe' or 'unsure' which is a more accurate classification.

Bitblaze [21], a binary analysis platform is one of such hybrid vulnerability detection frameworks. It integrates static and dynamic analysis and performs symbolic evaluation on path constraints from an execution trace to automatically generate inputs to traverse different program paths and used it for detecting vulnerabilities. Code Auditor [22] is another vulnerability detection framework. It is based on constraint analysis and model checking and can be used for detecting buffer overflow vulnerabilities in C source code but incurs high false positives (around 23%).

However, all these frameworks are catered to either deal with specific language/ domain. Our proposed framework is generic in nature. For example, specifications can be written for any programming language constructs or assembly code and since our three level specifications accounts for domain as well as custom specifications it can be used to detect web application vulnerabilities too, where the sanitization plays a major role and since all these databases are extensible and reusable, we can detect a wide variety of vulnerabilities using our single proposed framework as opposed to having a specialized framework for each vulnerability or application programming language or domain.

VI. CONCLUSION

Existing vulnerability detection approaches classify software as 'safe' or 'unsafe' and suffer from either false positives or false negatives but there should be another class 'unsure', which is highly ignored. Hence we classify software as 'safe' or 'unsafe' or 'unsure'. One of our major contributions in the paper is the proposal for a hybrid framework seamlessly integrating formal verification with prediction for security vulnerability detection. The results from the proposed approach can be used as an input for the security testing which is crucial to many software systems. In the proposed approach, first, we use formal verification to classify a sensitive sink as 'safe' or 'unsafe' if safe or unsafe properties can be proved. Other vulnerable sensitive sinks will be classified as "unsure" cases. We proposed using data mining code attributes to predict the vulnerabilities in 'unsure' cases. Another contribution of the paper is our proposal for three level formal specification of statement's safe/unsafe usage. Writing formal specification for all statement constructs in the programming language is a huge task. Since security vulnerabilities only happen at sensitive sinks we limit the specification only to such statements. Our three level specifications capture the semantics of programming language statement in general, domain specific requirements and custom specifications to cater to domain and business logic implications on vulnerabilities too. These specifications are extensible and reusable. To evaluate our proposed approach, we applied a case study on XSS vulnerability detection over 6 test subjects. The results prove the practicality of the proposed framework.

REFERENCES

[1] CVE — Common Vulnerabilities and Exposures. (2013). Available: http://cve.mitre.org

[2] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," presented at the Proceedings of the 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA, 2007.

[3] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," presented at the Proceedings of the Second ACM-IEEE international symposium on

Empirical software engineering and measurement, Kaiserslautern, Germany, 2008.

[4] L. K. Shar and H. B. K. Tan, "Mining input sanitization patterns for predicting SQL injection and cross site scripting vulnerabilities," presented at the Proceedings of the 2012 International Conference on Software Engineering, Zurich, Switzerland, 2012.

[5] L. K. Shar, H. B. K. Tan, and L. C. Briand, "Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis," presented at the Proceedings of the 2013 International Conference on Software Engineering, San Francisco, CA, USA, 2013.

[6] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," ACM Trans. Softw. Eng. Methodol., vol. 10, pp. 209-254, 2001.

[7] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper)," presented at the Proceedings of the 2006 IEEE Symposium on Security and Privacy, 2006.

[8] (2013). *Z3: SMT solver*. Available: http:// z3.codeplex.com/

[9] A. Kiezun, V. Ganesh, P. Guo, P. Hooimeijer, D. Akhave, S. Artzi, and M. D. Ernst, "HAMPI: A Solver for String Constraints," presented at the ACM International Symposium on Testing and Analysis, Chicago, Illinois, USA, 2009.

[10] Sourceforge. (2012). Available: http://sourceforge.net/

[11] J. W. Han, M. Kamber, and J. Pei, Data Mining: Concepts and Techniques, Third Edition: Morgan Kaufmann Publishers Inc., 2011.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings," IEEE Trans. Softw. Eng., vol. 34, pp. 485-496, 2008.

[13] Weka3. (2013). Available: www.cs.waikato.ac.nz/ml/weka/

[14] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," presented at the Proceedings of the 15th conference on USENIX Security Symposium - Volume 15, Vancouver, B.C., Canada, 2006.

[15] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," presented at the Proceedings of the 30th international conference on Software engineering, Leipzig, Germany, 2008.

[16] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic creation of SQL Injection and cross-site scripting attacks," presented at the Proceedings of the 31st International Conference on Software Engineering, 2009.

[17] OWASP. (2013). Available: http://ha.ckers.org/xss.html

[18] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," in Network and Distributed System Security Symposium(NDSS), 2003, pp. 149-162.

[19] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," IEEE Trans. Softw. Eng., vol. 37, pp. 772-787, 2011.

[20] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," presented at the Proceedings of the 4th ACM workshop on Quality of protection, Alexandria, Virginia, USA, 2008.

[21] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," presented at the Proceedings of the 4th International Conference on Information Systems Security, Hyderabad, India, 2008.

[22] W. Lei, C. Gui, W. Jianan, Z. Pengchao, and Z. Qiang, "CodeAuditor: A Vulnerability Detection Framework Based on Constraint Analysis and Model Checking," in Management and Service Science, 2009. MASS '09. International Conference on, 2009, pp. 1-4.

APPENDIX

Table III. CODE ATTRIBUTES FOR VULNERABILITY PREDICTION

Attribute ID	Attribute Name	Description
Code Attribute		
1	Client	The number of nodes which access data from HTML request parameters
2	File	The number of nodes which access data from files
3	Database	The number of nodes which access data from database
4	Text-database	Boolean value ‘TRUE’ if there is any text-based data accessed from database; ‘FALSE’ otherwise
5	Other-database	Boolean value ‘TRUE’ if there is any data except text-based data accessed from database; ‘FALSE’ otherwise
6	Session	The number of nodes which access data from persistent data objects
7	Uninit	The number of nodes which reference un-initialized program variable
8	XSS-sanitization	The number of nodes that apply standard sanitization functions for preventing XSS issues
9	Numeric-casting	The number of nodes that type cast data into a numerical type data
10	Numeric-type-check	The number of nodes that perform numeric data type check
11	Encoding	The number of nodes that encode data into a certain format
12	Un-taint	The number of nodes that return predefined information or information not influenced by external users
13	Boolean	The number of nodes that invoke functions which return Boolean value
24	Propagate	The number of nodes that propagate the tainted-ness of an input string
15	Numeric	The number of nodes that invoke functions which return only numeric characters, mathematic operators, and/or dash character
16	LimitLength	The number of nodes that invoke string-length limiting functions
17	URL	The number of nodes that invoke path-filtering functions
18	EventHandler	The number of nodes that invoke event handler filtering functions
19	HTMLTag	The number of nodes that invoke HTML tag filtering functions
20	Delimiter	The number of nodes that invoke delimiter filtering functions
21	AlternateEncode	The number of nodes that invoke alternate character encoding filtering functions
Target Attribute		
22	Vulnerable?	Target attribute which indicates a class label—Vulnerable or Not-Vulnerable