Tailoring Taint Analysis for Database Applications in the $\mathbb{K}$ Framework

Md. Imran Alam[1,2][a] and Raju Halder[1][b]
[1]*Indian Institute of Technology Patna, India*
[2]*Università Ca' Foscari Venezia, Italy*

Keywords: Database Applications, Taint Analysis, $\mathbb{K}$ Framework, Security.

Abstract: Maintaining the integrity of underlying databases of any information systems is one of the challenges. This could be either due to coding flaws or due to improper flow of information from source to sink in the associated database applications. Compromising this may lead to either disclosure of sensitive information to the attackers or illegitimately modification of private data stored in the databases. Taint analysis is a widely used program analysis technique that aims at averting malicious inputs from corrupting data values in critical computations of programs. In this paper, we propose `K-DBTaint`, a rewriting logic-based executable semantics for taint analysis of database applications in the $\mathbb{K}$ framework. We specify the semantics for a subset of SQL statements along with host imperative program statements. Our $\mathbb{K}$ semantics can be seen as a sound approximation of program semantics in the corresponding security type domain. With respect to the existing methods, `K-DBTaint` supports context- and flow-sensitive analysis, reduces false alarms, and provides a scalable solution. Experimental evaluation on several PL/SQL benchmark codes demonstrates encouraging results as an improvement in the precision of the analysis.

1 INTRODUCTION

Database applications are ubiquitous today and many of these applications are developed in general-purposes languages embedded with SQL statements. Vulnerabilities in database applications pose serious security and privacy threats such as the exposure of confidential data, loss of customer trust, and denial of service. According to OWASP[1], the most serious vulnerabilities are SQL injection, cross-site scripting, buffer overflow, etc. These vulnerabilities are usually caused either due to coding flaws or due to improper flow of information from source to sink in the associated database applications.

Taint analysis is a well-established technique that aims at averting malicious inputs from corrupting data values in critical computations of programs (Huang et al., 2014). Tainted data refers to data which originate from potentially malicious users and can cause security problems at vulnerable points (known as sensitive sinks) in the program. Tainted data may enter in a program through specific places and spread across the program via assignments and similar constructs. To exemplify this scenario, let us consider the code snippet, shown in Figure 1, that allows users to search a product catalog. The user input is read through the auxiliary function read() and is integrated with the SQL query at line number 5. When the query is executed, it extracts information of those products whose description IDs (*i.e.*, "Descpr") match with user-inputs (*i.e.*, "*prod*"). If a user supplies " 'tpid' OR 1 = 1; `DROP TABLE Products`" as input, the query returns all the rows of the `Products` table and drops `Products` table form the database. This way, the attackers may inject tainted input and attempt to alter the actual intent of the code. This is known as SQL Injection attack (Su et al., 2018).

```
void prodsearch( ) {
1. prod VARCHAR(15);
2. z int;
3. prod = read();
4. if prod ⩾ 1 then
5.     SELECT PName INTO z FROM Products WHERE Descpr =
         prod;
}
```

Figure 1: A motivating example.

In practice, many database applications which

[a] https://orcid.org/0000-0003-2700-4127
[b] https://orcid.org/0000-0002-8873-8258
[1] https://owasp.org/www-project-top-ten/

Alam, M. and Halder, R.
Tailoring Taint Analysis for Database Applications in the K Framework.
DOI: 10.5220/0010618603700377
In *Proceedings of the 10th International Conference on Data Science, Technology and Applications (DATA 2021)*, pages 370-377
ISBN: 978-989-758-521-0

deal with sensitive data, *e.g.* financial, healthcare, etc., do not perform proper checking of input data, and allows attackers to steal or modify weakly protected data in the underlying database to conduct, for example, credit card fraud, identity theft, or other crimes.

Approaches based on taint analysis (Jovanovic et al., 2006; Wassermann and Su, 2008; Cao et al., 2017; Tripp et al., 2009) identify potential vulnerabilities in program code. Unfortunately, these approaches suffer form false alarms due to ignorance of the control-flow, the semantics of SQL statements and constant functions. Security type-system (Huang et al., 2014) has emerged independently as a probably most popular approach to static taint analysis in a competing manner.

In this paper, we extend Hunt and Sands's security type system (Hunt and Sands, 2006) and we propose `K-DBTaint`, a rewriting logic-based executable semantics in the $\mathbb{K}$ framework for taint analysis of database applications. The $\mathbb{K}$ framework is a rewrite logic-based framework for defining programming language semantics suitable for formal analysis and reasoning about programs and programming languages (Roşu and Şerbănuţă, 2010; Asăvoae, 2014). Inspired by rewrite-logic semantics project (Meseguer and Roşu, 2007), this framework unifies algebraic denotational semantics and operational semantics by considering them as two different view over the same object. Such semantic definitions are directly executable in a rewriting logic language, e.g. Maude (Clavel and et al., 2007), thus support a development of analysis tools at no cost. With respect to the literature, the developed prototype based on our theoretical foundation is flow-sensitive, context-sensitive, and captures integrity violations in database applications.

To summarize, our main contributions are:

- We apply the $\mathbb{K}$ framework to define taint analysis of database applications by extending Hunt and Sands's (Hunt and Sands, 2006) security type system as the basis.
- We specify $\mathbb{K}$ rewrite rules which capture semantics of both the database statements and the imperative statements of a host imperative language.
- The proposed analysis is flow-sensitive, context-sensitive, and improve the precision by handling constant functions.
- We develop a prototype tool based on our theoretical foundation which allow the users to analyse PL/SQL codes.
- We present experimental evaluation results on a set of PL/SQL benchmark codes to establish the effectiveness of our approach.

The paper is organized as follows: Section 2 discusses the related works in the literature on static taint analysis. A brief descriptions of abstract syntax of database language and $\mathbb{K}$ framework are presented in Section 3. Section 4 presents formulation of Hunt and Sands's security type system in the $\mathbb{K}$ framework. In Sections 5, we present the executable rewriting logic semantics in $\mathbb{K}$ for taint analysis. The experimental evaluation results are reported in section 6. Finally, section 7 concludes our work.

2 RELATED WORKS

Taint analysis, a form of information-flow analysis, detects integrity violations in database applications (Cao et al., 2017; Huang et al., 2014; Tripp et al., 2009; Medeiros et al., 2015; Wassermann and Su, 2008; Jovanovic et al., 2006; Maskur and Asnar, 2019; Halim and Asnar, 2019; Vijayalakshmi and Syed Mohamed, 2021; Jana et al., 2018). The authors in (Tripp et al., 2009) present TAJ, an analysis tool for industrial applications. WAP-TA (Medeiros et al., 2015) and PIXY (Jovanovic et al., 2006) apply taint analysis to detect vulnerabilities in server-side scripts written in PHP through an inter-procedural context-sensitive data flow analysis. However, these analyses are imprecise as they do not support constant functions. Static taint analysis for detecting cross-site scripting vulnerabilities in JavaScript codes is proposed in (Wassermann and Su, 2008). Unfortunately, due to ignorance of control dependencies, the proposed technique is unable to capture the indirect influence of taint information on other variables due to implicit flow. Control flow graph-based fine-grained taint analysis of PHP scripts is proposed in (Cao et al., 2017; Halim and Asnar, 2019). Pattern-based taint analysis of web application is proposed in (Maskur and Asnar, 2019). The proposed approaches are not context-sensitive and generate false alarms with function calls. Moreover, the above approaches fail to address false positives in presence of constant functions, such as $x := 0 \times x$, $x := y - y$, etc. Observe that these approaches are not directly applicable to database applications due to the presence of external database states along with program's internal states. Intuitively, precise taint analysis of database applications requires handling of both database and program states on the basis of semantics foundation. Ignoring database states and treating database statements as a black box, of course, provide a pathway for possible database-specific security threats.

Table 1 reports a summary of the state-of-the-art tools and techniques in the line of static taint analysis,

Table 1: A Comparative Summary (☑ denotes partially successful at this stage).

	K-DBTaint	Pixy	WAP-TA	Su et al.	TAJ	Cao et al.
Semantics/Security Type System	✓	✗	✗	✗	✓	✓
Explicit Flow	✓	✓	✓	✓	✓	✓
Implicit Flow	✓	✗	✓	✗	✗	✓
Constant Functions	☑	✗	✗	✗	✗	✗
Flow-Sensitivity	✓	✓	✓	✗	✓	✓
Context-Sensitivity	✓	✓	✓	✓	✓	✗
SQL Semantics	✓	✗	✗	✗	✗	✗
Language Supported	Database Applications	PHP	PHP + SQL	JavaScript	Java	PHP

as compared with K-DBTaint. A recent case study on detection of vulnerabilities in web applications using taint analysis is reported in (Vijayalakshmi and Syed Mohamed, 2021).

3 PRELIMINARIES

In this section, we first recall the abstract syntax of the database language (Alam et al., 2021; Alam and Halder, 2021) under consideration and then describe the $\mathbb{K}$ framework (Roşu and Şerbănuţă, 2010; Asăvoae, 2014) in brief.

Table 2: Abstract Syntax of the Database Language (Alam et al., 2021; Alam and Halder, 2021).

E	::=	$n \mid id \mid E\ ap\ E \mid (E)$, where $ap \in \{+, -, \times, /\}$
B	::=	$true \mid false \mid E\ rel\ E \mid \neg B \mid B$ AND $B \mid B$ OR B, where $rel \in \{\geqslant, \leqslant, <, >, ==\}$
τ	::=	$int \mid float \mid char \mid bool$
D	::=	$\tau\ id$
A	::=	$id := E \mid id := read()$
Q_{sel}	::=	SELECT $\vec{E}$ INTO $\vec{rs}$ FROM id WHERE B;
Q_{ins}	::=	INSERT INTO id ($\vec{id}$) VALUES ($\vec{E}$);
Q_{upd}	::=	UPDATE id SET $\vec{id} = \vec{E}$ WHERE B;
Q_{del}	::=	DELETE FROM id WHERE B;
Q	::=	$Q_{sel} \mid Q_{ins} \mid Q_{upd} \mid Q_{del}$
C	::=	$skip; \mid Q \mid D; \mid A; \mid defun\ id(\vec{D})\{C\} \mid call\ id(\vec{E}); \mid return; \mid return\ E; \mid if\ B\ then\ \{C\} \mid if(B)\ then\ \{C_1\}\ else\ \{C_2\} \mid while(B)\ do\ \{C\}$
$\mathcal{P}$	::=	$C \mid C\ ;\ \mathcal{P}$

3.1 Abstract Syntax of Database Language

We consider a generic database language scenario where SQL statements are embedded in a high-level imperative language. Its abstract syntax is shown in Table 2. An identifier denoted by *id* represents either a program variable or a database attribute or a database table name. For simplicity, we assume that all attributes names in the database schema are distinct. The arithmetic and boolean expressions are denoted by E and B respectively. By convention, $\vec{id}$ stands for a sequence of identifiers $\langle id_1, id_2, \ldots, id_n \rangle$ and $\vec{E}$ stands for a sequence of arithmetic expressions $\langle E_1, E_2, \ldots, E_m \rangle$. The declaration and assignment statements in imperative language are denoted by D and A. We consider a subset of database manipulation statements, namely SELECT (Q_{sel}), INSERT (Q_{ins}), UPDATE (Q_{upd}), and DELETE (Q_{del}).

The Q_{sel} statement filters a set of tuples from the target table id based on the satisfiability of B and stores them in the resultset program variable $\vec{rs}$. Similarly, the Q_{upd} updates the attributes $\vec{id}$ of table id by new values of $\vec{E}$, indicated by $\vec{id} = \vec{E}$, when the corresponding rows satisfy B. More specifically, the term $\vec{id} = \vec{E}$ denotes the sequence of assignments $id_1 = E_1, \ldots, id_n = E_n$, where id_i denotes the i^{th} attribute name, and E_i represents i^{th} expression. To exemplify this, let us consider the statement "UPDATE Product SET $Item_1$:= 'apple', $Item_2$:= 'orange' WHERE $Pid = uspid$;". Its abstract syntax $\vec{id} = \vec{E}$ is denoted by $\langle Item_1, Item_2 \rangle = \langle$ 'apple', 'orange' $\rangle$ and B is denoted by "$Pid = uspid$". The Q_{ins} appends a new data record (denoted by VALUES($\vec{E}$)) to a table id. The Q_{del} statement deletes records from table id, that satisfy the condition B. The other imperative statements supported by the language are assignment, function, conditional, looping, and return. A database program $\mathcal{P}$ consists of sequence of statements C.

3.2 The $\mathbb{K}$ Framework

In this section, we briefly describe the $\mathbb{K}$ framework (Roşu and Şerbănuţă, 2010). The $\mathbb{K}$ framework is a rewrite logic-based framework for defining programming language semantics suitable for formal reasoning about programs and programming languages.

Any formal semantics of a language requires first a formal syntax. In $\mathbb{K}$ framework, language syntax is defined using a variant of the familiar BNF notation, with terminals enclosed in quotes and non-terminals starting with capital letters. For example, the syntax declaration:

$$\text{syntax } E ::= \texttt{id} \mid E \text{ "}*\text{" } E \quad [\texttt{strict}]$$

defines a syntactic category E, containing the program variables and a basic arithmetic operation on expressions of database language. Each production can have a space-separated list of attributes which can be specified in square brackets at the end of the production.

Specifying language semantics using $\mathbb{K}$ framework, consists of three parts: providing evaluation strategies that conveniently (re)arrange computations (*computations*), giving the structure of the configuration to hold program states (*configuration*), and writing $\mathbb{K}$ rules to describe transitions between configurations ($\mathbb{K}$ *rewrite rules*).

Evaluation strategies serve as a link between syntax and semantics, by specifying how the arguments of a language construct should be evaluated. For example, consider the following syntax for arithmetic expression:

$$\text{syntax } E ::= E_1 \text{ "+" } E_2 \quad [\texttt{strict}]$$

The attribute `strict` allows E_1 and E_2 to evaluate in any order, thus enforces a non-determinism. The annotation above corresponds to the following four heating/cooling rules:

$$\langle \frac{E_1 + E_2}{E_1 \curvearrowright \square + E_2} \cdots \rangle_k \mid \langle \frac{E_1 + E_2}{E_2 \curvearrowright \square + E_1} \cdots \rangle_k \mid \langle \frac{V_1 \curvearrowright \square + E_2}{V_1 + E_2} \cdots \rangle_k \mid \langle \frac{V_2 \curvearrowright E_1 + \square}{E_1 + V_2} \cdots \rangle_k$$

Here, V_1 and V_2 are the evaluated results of the expressions Exp_1 and Exp_2 respectively. The construct $\square$ (HOLE) is a place-holder that will be replaced by the result of the evaluated term or sub-term.

Configurations represent the state of a running program/system and are structured as nested, labeled cells containing various computation-based data structures. Within the $\mathbb{K}$, configuration cells are represented using an XML-like notation, with the label of the cell as the tag name and the contents between the opening and closing tags (*i.e.*, `List`, `Map`, `Bag`, `Set`, etc.). For example, consider the following configuration with three cells:

$$\text{configuration} \equiv \langle \langle K \rangle_k \langle \texttt{Map}[id \mapsto L] \rangle_{env} \langle \texttt{Map}[L \mapsto n] \rangle_{store} \rangle_T$$

The k cell holds a list of computational tasks, that is $k : \texttt{List}\{K, \curvearrowright\}$ where K holds computational contents such as programs or fragment of programs and $\curvearrowright$ is the task sequentialization operator which sequentializes program statements. The *env* cell maps variables to their locations (i.e., $env : id \mapsto L$) and the *store* cell maps locations to values (i.e., $store : L \mapsto n$). These cells are covered by the top cell denoted by T.

$\mathbb{K}$ *rules* describe how a running configuration evolves by advancing the computation and potentially altering the state/environment. $\mathbb{K}$ *rewrite rules* are classified into two types: *computational rules*, that may be interpreted as transition in a program execution, and *structural rules*, that rearrange a term to enable the application of computational rule. For better understanding, let us consider the following rule, considering two cells k and *env*, for a variable declaration:

$$\langle \frac{\tau\ id}{.} \cdots \rangle_k \langle \frac{\rho}{\rho[id \leftarrow \texttt{T} : Type]} \rangle_{env}$$

This specifies that the next task to evaluate is a variable declaration, which is replaced by an empty computation "." and a type `T` is assigned to the variable *id* in the environment cell *env*.

By keeping rules compact and less redundant, it is less likely that a rule will need to be changed as the configuration is changed or new constructs are added to the language.

4 FORMULATING SECURITY TYPE SYSTEM IN THE $\mathbb{K}$ FRAMEWORK

In this section, we first extend Hunt and Sands's security type system (Hunt and Sands, 2006) to the case of database applications, and then we formulate their typing judgments and rules in the $\mathbb{K}$ framework.

Extending Hunt and Sands's Security Type Systems: As Hunt and Sands's security type system is flow-sensitive, we extend this for the purposes of our taint analysis with context-sensitivity in case of inter-procedural database code. This is depicted in Figure 2. We consider two security types *taint* and *untaint* with the semi-join lattice of the type domain as $SD = \langle \mathbb{S}, \sqsubseteq, \sqcup \rangle$, where $\mathbb{S} = \{taint, untaint\}$ and the partial order relation defined as $untaint \sqsubseteq taint$.

Formulating the Type System: Let us now formulate the typing judgements and rules in the $\mathbb{K}$ framwork. Let us first consider the typing judgement $pc \vdash \Gamma\{C\}\Gamma'$ which specifies that the security environment Γ' is derived by executing the statement C on the security environment Γ under the program's security context *pc*. To formulate this, we first define a configuration with three cells namely, k cell , *env* cell, and *context* cell as follows: $\langle \langle K \rangle_k \langle \texttt{Map} \rangle_{env} \langle \texttt{Map} \rangle_{context} \rangle_T$. Then we define following $\mathbb{K}$ rule to formulate the type judgement $pc \vdash \Gamma\{C\}\Gamma'$:

$$\langle \frac{C}{.} \cdots \rangle_k \langle \frac{\Gamma}{\Gamma'} \rangle_{env} \langle pc \mapsto \_ \rangle_{context}$$

The symbol "…" appearing in the k cell represents remaining computations. As a result of the execution of C which eventually be consumed (denoted by "."), the previous environment Γ in the *env* cell will be updated by the modified environment Γ' (implicitly) influenced by the current value (denoted by "_") of the security context *pc* in the *context* cell.

Each security type rule is written as $\frac{\Gamma_i \vdash \zeta_i}{\Gamma \vdash \zeta}$, where judgements $\Gamma_i \vdash \zeta_i$ denotes number of *premise* and judgement $\Gamma \vdash \zeta$ denotes a single *conclusion*. For example, the type rule for assignment statement $\frac{\Gamma \vdash E : \mathtt{T}}{pc \vdash \Gamma\ \{id := E\}\ \Gamma[\![id \mapsto pc \sqcup \mathtt{T}]\!]}$ is defined by the corresponding $\mathbb{K}$ rule $\langle \frac{id\ :=\ \mathtt{T} : Type}{\mu(pc) \sqcup \mathtt{T} : Type} \ldots \rangle_k\ \langle \ldots\ \rho[id \mapsto \ldots] \ldots \rangle_{env}\ \langle \mu \rangle_{context}$. Considering this as basis, in the following section we discuss $\mathbb{K}$ rewrite rules for static taint analysis of database language in the abstract security type domain $\mathbb{S}$.

$$[\mathtt{Expression}]\quad \frac{}{\Gamma \vdash E :\ \sqcup_{x \in \mathtt{FV}(E)} \Gamma(x)} \qquad [\mathtt{skip}]\quad \frac{}{pc \vdash \Gamma\{skip\}\Gamma}$$

$$[\mathtt{Declaration}]\quad \frac{}{pc \vdash \Gamma\{\tau\ id\}\ \Gamma[\![id \mapsto pc \sqcup untaint]\!]}$$

$$[\mathtt{Read}]\quad \frac{}{pc \vdash \Gamma\{id := read()\}\ \Gamma[\![id \mapsto pc \sqcup taint]\!]}$$

$$[\mathtt{Assignment}]\quad \frac{\Gamma \vdash E :\ \mathtt{T}}{pc \vdash \Gamma\ \{id := E\}\ \Gamma[\![id \mapsto pc \sqcup \mathtt{T}]\!]}$$

$$[\mathtt{DELETE}]\quad \frac{\Gamma \vdash B : \mathtt{T}}{pc \vdash \Gamma\{\mathtt{DELETE\ FROM}\ id\ \mathtt{WHERE}\ B\}\Gamma}$$

$$[\mathtt{SELECT}]\quad \frac{\Gamma \vdash B : \mathtt{T} \qquad pc \sqcup \mathtt{T} \vdash \Gamma\{\vec{rs} := \vec{E}\}\Gamma'}{pc \vdash \Gamma\ \{\mathtt{SELECT}\ \vec{E}\ \mathtt{INTO}\ \vec{rs}\ \mathtt{FROM}\ id\ \mathtt{WHERE}\ B\}\ \Gamma \sqcup \Gamma'}$$

$$[\mathtt{INSERT}]\quad \frac{pc \vdash \Gamma\{\vec{id} := \vec{E}\}\Gamma'}{pc \vdash \Gamma\ \{\mathtt{INSERT\ INTO}\ id(\vec{id})\ \mathtt{VALUES}(\vec{E})\}\ \Gamma \sqcup \Gamma'}$$

$$[\mathtt{UPDATE}]\quad \frac{\Gamma \vdash B : \mathtt{T} \qquad pc \sqcup \mathtt{T} \vdash \Gamma\{\vec{id} := \vec{E}\}\Gamma'}{pc \vdash \Gamma\ \{\mathtt{UPDATE}\ id\ \mathtt{SET}\ \vec{id} := \vec{E}\ \mathtt{WHERE}\ B\}\ \Gamma \sqcup \Gamma'}$$

$$[\mathtt{Function\ Call}]\quad \frac{\begin{array}{c} defun\ id(\vec{D})\{C\} \\ \Gamma \vdash \vec{\mathtt{E}} : \vec{\mathtt{T}} \quad \vec{X} = getParam(\vec{D}) \quad \frac{pc \vdash \Gamma'\ \{\mathtt{C}\}\ \Gamma''}{pc \vdash \Gamma'\ \{defun\ id(\vec{D})\{C\}\}\ \Gamma''} \\ \Gamma[\![\vec{X} \mapsto \vec{\mathtt{T}}]\!] \equiv \Gamma' \end{array}}{pc \vdash \Gamma\ \{call\ id(\vec{E})\}\ \Gamma''}$$

$$[\mathtt{if}]\quad \frac{\Gamma \vdash B : \mathtt{T} \qquad pc \sqcup \mathtt{T} \vdash \Gamma\{C\}\Gamma'}{pc \vdash \Gamma\ \{if\ B\ then\ C\}\ \Gamma \sqcup \Gamma'}$$

$$[\mathtt{if\text{-}else}]\quad \frac{\Gamma \vdash B : \mathtt{T} \qquad pc \sqcup \mathtt{T} \vdash \Gamma\{C_1\}\Gamma' \qquad pc \sqcup \mathtt{T} \vdash \Gamma\{C_2\}\Gamma''}{pc \vdash \Gamma\ \{if\ B\ then\ \{C_1\}\ else\ \{C_2\}\}\ \Gamma' \sqcup \Gamma''}$$

$$[\mathtt{while}]\quad \frac{\begin{array}{ccc} \Gamma'_i \vdash B : \mathtt{T}_i & pc \sqcup \mathtt{T}_i \vdash \Gamma'_i\{\mathtt{C}\}\Gamma''_i & 0 \le i \le k \\ \Gamma'_0 = \Gamma & \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma & \Gamma'_{k+1} = \Gamma'_k \end{array}}{pc \vdash \Gamma\ \{while\ B\ do\ \{C\}\}\ \Gamma'_k}$$

Figure 2: Flow- and Context-sensitive Security Type Rules for Taint Analysis.

5 $\mathbb{K}$ SEMANTICS FOR K-DBTaint

In this section, we present $\mathbb{K}$ rewrite rules for taint analysis of our database language. The proposed analysis addresses semantics of both the database statements and the imperative program statements together. To this aim, we consider the following $\mathbb{K}$ configuration on which the semantics is defined:

$$configuration \equiv \langle \langle K \rangle_k\ \langle Map \rangle_{env}\ \langle \mathtt{Map} \rangle_{context}\ \langle\ \langle \mathtt{Map}\ \rangle_{\lambda\text{-}Def} \langle \mathtt{List} \rangle_{fstack}\ \rangle_{control} \langle \mathtt{List} \rangle_{in}\ \langle \mathtt{List} \rangle_{out}\ \rangle_T$$

Where the special cell $\langle\rangle_k$ contains the list of computation tasks, environment cell *env* maps variables or database attributes to their security types, *context* cell denotes the current program context, λ-*Def* cell supports inter-procedural feature, *in* and *out* cells denotes input-output operations, and *control* cell manages function calls using *fstack* cell. We now describe the $\mathbb{K}$ rules for imperative program statements and database statements as depicted in Figure 3. We label the defined rules by R_ for future reference. Note that these rules captures explicit- and implicit- flow sensitivity, context-sensitivity, the semantics of constant functions, and the semantic of SQL statements.

Imperative Program Statements: Due to space scarcity, here we discuss a few rules of imperative program statements. However, a reader can refer (Alam et al., 2018) for a complete set of rules of imperative programs. The first rule $\mathbf{R_{decl}}$ deals with variables declarations and initialization of variables by their initial security types (*untaint* in our case) in the environment cell *env*. Any unsanitized input gets its type tainted in the rule $\mathbf{R_{read}}$. Rule $\mathbf{R_{asg}}$ which handles assignment computations, updates the security type of *id* somewhere in the *env* cell by the least upper bound of the security types of the right hand side expression (i.e. T) and program's current security context *pc* in the *context* cell. The assignment is then replaced by an empty computation. In order to capture implicit flow of taint information in presence of conditional and loop statements, we define rules $\mathbf{R_{if}}$ and $\mathbf{R_{while}}$ which update the security context μ in the *context* cell based on the security type of B. The term $restore_c(\mu)$ restores the previous context on exiting a block guarded by B and $approx(\rho)$ provides a sound approximation of the semantics as a least upper bound of the environments obtained over all possible execution paths due to the presence of B. Note that the least fixed point solution in case of "*while*" is achieved by defining one of the following auxiliary function *fixpoint*():

$$(1)\quad \langle \frac{fixpoint(B,C,\rho_i)}{\cdot} \ldots \rangle_k \langle \rho'_i \rangle_{env} \text{ when } \rho_i = \rho'_i,$$

$$\mathbf{R_{decl}}: \langle \frac{\tau\ id}{\cdot} \ \ldots\rangle_k \langle \frac{\rho}{\rho[id \leftarrow \mathtt{T} : Type]}\rangle_{env} \quad \mathbf{R_{read}}: \langle \frac{read(\,)}{taint} \ \ldots\rangle_k$$

$$\mathbf{R_{con\text{-}func}}: \langle id_1 * id_2 \ \ldots\rangle_k = \begin{cases} \langle \frac{id_1 * id_2}{untaint} \ \ldots\rangle_k \ when\ id_1 = zero \\ \qquad\qquad\qquad or\ id_2 = zero \\ \langle \frac{id_1 * id_2}{id_1 *_{Type} id_2} \ \ldots\rangle_k \ otherwise \end{cases}$$

$$\mathbf{R_{asg}}: \langle \frac{id\ :=\ \mathtt{T} : Type}{\cdot} \ \ldots\rangle_k \langle \ldots\ \rho[id \mapsto \frac{\_}{\mu(pc) \sqcup \mathtt{T} : Type}] \ \ldots\rangle_{env} \langle \mu \rangle_{context}$$

$$\mathbf{R_{if}}: \langle \frac{if(B : \mathtt{T})\ then\ \{C\}}{C \curvearrowright restore_c(\mu) \curvearrowright approx(\rho)} \ \ldots\rangle_k \langle \frac{\mu}{\mu[pc \leftarrow \mu(pc) \sqcup \mathtt{T}]} \rangle_{context} \langle \rho \rangle_{env}$$

$$\mathbf{R_{while}}: \langle \frac{while(B : \mathtt{T})\ do\ \{C\}}{C \curvearrowright restore_c(\mu) \curvearrowright approx(\rho) \curvearrowright fixpoint(B, C, \rho)} \ \ldots\rangle_k \langle \rho \rangle_{env} \langle \frac{\mu}{\mu[pc \leftarrow \mu(pc) \sqcup \mathtt{T}]} \rangle_{context}$$

$$\mathbf{R_{fun\text{-}decl}}: \langle \frac{defun\ Func\_name(Params)\{C\}}{\cdot} \ \ldots\rangle_k \langle \langle \frac{\psi}{\psi[Func\_name \leftarrow lambda(Params, C)]} \rangle_{\lambda\text{-}Def} \rangle_{control}$$

$$\mathbf{R_{fun\text{-}call}}: \langle \frac{lambda(Params, C)(Es : Ts) \curvearrowright K}{McDecls(Params, Ts) \curvearrowright C \curvearrowright return;} \ \ldots\rangle_k \langle \langle \frac{.List}{[ListItem(\rho, K, Ctr)]} \ \ldots\rangle_{fstack}\ Ctr \rangle_{control} \langle \rho \rangle_{env}$$

$$\mathbf{R_{aux\text{-}fun(a)}}: \langle \frac{makeAssign(id,\ E)}{id := E;} \ \ldots\rangle_K$$

$$\mathbf{R_{aux\text{-}fun(b)}}: \langle \frac{makeAssign((id,\ \vec{id}),\ (E,\ \vec{E}))}{id := E; \curvearrowright makeAssign(\vec{id},\ \vec{E})} \ \ldots\rangle_K$$

$$\mathbf{R_{aux\text{-}fun(c)}}: \langle \frac{makeupdAssign(id_1 = E_1, \ldots, id_n = E_n)}{id_1 = E_1; \curvearrowright \ \ldots\ \curvearrowright id_n = E_n;} \ \ldots\rangle_k$$

$$\mathbf{R_{sel}}: \langle \frac{\mathtt{SELECT}\ \vec{E}\ \mathtt{INTO}\ \vec{rs}\ \mathtt{FROM}\ id\ \mathtt{WHERE}\ B;}{\mathtt{if}(B)\{makeAssign(\vec{rs},\ \vec{E})\}} \ \ldots\rangle_k$$

$$\mathbf{R_{ins}}: \langle \frac{\mathtt{INSERT\ INTO}\ id(\vec{id})\ \mathtt{VALUES}\ (\vec{E});}{makeAssign(\vec{id},\ \vec{E})} \ \ldots\rangle_k$$

$$\mathbf{R_{upd}}: \langle \frac{\mathtt{UPDATE}\ id\ \mathtt{SET}\ \vec{id} = \vec{E}\ \mathtt{WHERE}\ B;}{\mathtt{if}(B)\{makeupdAssign(id_1 = E_1, \ldots, id_n = E_n)\}} \ \ldots\rangle_k$$

$$\mathbf{R_{del}}: \langle \frac{\mathtt{DELETE\ FROM}\ id\ \mathtt{WHERE}\ B;}{\cdot} \ \ldots\rangle_k$$

Figure 3: $\mathbb{K}$ rewrite rules for Database Applications.

(2) $\langle \frac{fixpoint(B, C, \rho_i)}{while(B)\ do\ \{C\}} \ \ldots\rangle_k \langle \rho_i' \rangle_{env}$ when $\rho_i \neq \rho_i'$.

Where case (1) represents that the computation reaches the fix-point and therefore the computation is consumed. If not, then the iteration continues as shown in case (2). The context-sensitivity in presence of function calls is captured by the rule $\mathbf{R_{func\text{-}call}}$.

SQL Statements: Before we define the semantics of SQL statements, let us first define the semantics of auxiliary function *makeAssign()* which takes two parameters and generate an assignment statement as follows:

$$makeAssign(id,\ E) => id := E;$$

where *id* represents an identifier and E denotes an expression. On passing a list as parameters to the function *makeAssign()*, it generates a set of assignment statements as follows:

$$makeAssign((id, \vec{id}),\ (E,\ \vec{E})) => id := E; \curvearrowright makeAssign(\vec{id},\ \vec{E})$$

where $\vec{id}$ denotes a list of identifier and $\vec{E}$ denotes a list of expression. Observe that, in case of empty parameter passed to *makeAssign()*, it generates an empty computation "." (*i.e.*, *makeAssign(.id, .E)* => .). Similarly, the function *makeupdAssign*() accepts term of the form $\langle id_1 = E_1,\ \ldots,\ id_n = E_n \rangle$, and returns a set of assignment statements as follows:

$$makeupdAssign(id_1 = E_1,\ \ldots,\ id_n = E_n) => id_1 = E_1; \curvearrowright \ \ldots\ \curvearrowright id_n = E_n;$$

Let us now define the semantics of the subset of SQL statements supported by the language under consideration.

The `SELECT` statement Q_{sel} retrieves values of those attributes that are present in the expression $\vec{E}$. It includes a `WHERE` clause to select number of rows to be retrieved. The presence of condition B in the `WHERE` clause act as an implicit flow of taint information. In order to capture such implicit flow, we translate Q_{sel} to an equivalent `if` statement as depicted in rule $\mathbf{R_{sel}}$. Note that, the auxiliary function $makeAssign(\vec{rs},\ \vec{E})$ in $\mathbf{R_{sel}}$ captures flow of tainted data which are retrieved from the database table using Q_{sel}.

The `UPDATE` statement Q_{upd} updates the values of the attributes according to the condition in `WHERE` clause. Like `SELECT` statement, condition in the `WHERE` clause of `UPDATE` statement also acts as implicit flow of taint information. Therefore, rule $\mathbf{R_{upd}}$ translates Q_{upd} into an equivalent `if` statement to capture implicit taint flow. The translation of $\vec{id} = \vec{E}$ into an auxiliary function $makeupdAssign(id_1 = E_1,\ \ldots,\ id_n = E_n)$ captures direct taint flow due to assignment of tainted data through the expression E.

Table 3: Taint Analysis on Benchmark Programs Set (PL/SQL, 2021) (✓: Passed, ✗$_+$: False Positives, ✗$_-$: False negatives).

Progs.	Descriptions	K-DBTaint	WAP-TA	Pixy	Su et al.	Cao et al.
Prog1	Balance_Transfer.sql (explicit-flow)	✓	✗$_-$	✗$_-$	✗$_-$	✗$_-$
Prog2	Update_Inventory.sql (explicit-flow)	✓	✗$_-$	✗$_-$	✗$_-$	✗$_-$
Prog3	Budget.sql (SQL injection)	✓	✗$_-$	✗$_-$	✗$_-$	✗$_-$
Prog4	Proc_Inventory.sql (malware attack)	✓	✗$_+$	✗$_+$	✗$_-$	✗$_-$
Prog5	Update_Quantity.sql (SQL injection)	✓	✗$_-$	✗$_+$	✗$_-$	✗$_-$
Prog6	Populate_Products.sql(SQL injection, constant function)	✓	✗$_-$	✗$_-$	✗$_-$	✗$_-$
Prog7	delete_Client.sql (SQL injection)	✓	✗$_+$	✗$_+$	✗$_+$	✗$_+$
Prog8	Update_Account.sql (SQL injection, implicit flow)	✓	✗$_+$	✗$_-$	✗$_-$	✗$_+$
Prog9	Record_NewSale.sql (explicit-flow)	✓	✗$_-$	✗$_-$	✗$_-$	✗$_-$
Prog10	Get_Country_Id.sql (explicit-flow, constant function)	✓	✗$_+$	✗$_+$	✗$_+$	✗$_+$
Prog11	Add_Car.sql (SQL injection, explicit-flow, constant function)	✓	✗$_+$	✗$_+$	✗$_+$	✗$_+$
Prog12	Award_Bonus.sql (SQL injection, malware attack)	✓	✗$_-$	✗$_-$	✗$_-$	✗$_-$
Prog13	Resrvation_Proc.sql (SQL injection, XSS attacks)	✓	✗$_-$	✗$_-$	✗$_+$	✗$_+$
Prog14	Credit_Account.sql (SQL injection, implicit flow)	✓	✗$_+$	✗$_-$	✗$_-$	✗$_+$
Prog15	Debit_Account.sql (SQL injection, implicit flow)	✓	✗$_+$	✗$_-$	✗$_-$	✗$_+$

The INSERT statement appends a tuple to the database table. We specify attributes names $\vec{id}$ and list of values $\vec{E}$ in INSERT statement in the same order as specified in the table definition. Therefore, any tainted expression $E \in \vec{E}$ directly stores tainted data in the database table that may be accessed and used in the future computations. Hence, the rule $\mathbf{R_{ins}}$ translates Q_{ins} into sequence of assignment statements to capture such direct flow of taint data due to assignment of E to id.

The DELETE statement deletes tuples from a table, we translate it by an empty computation leaving the current execution environment unchanged. This is depicted in rule $\mathbf{R_{del}}$.

Constant Functions Apart from the semantics of SQL statements, defining the semantics of constant functions greatly improve the precision of taint analysis. For example, consider the statement $y := z \times 0 + 4$, where z is a tainted variable. Note that, although the syntax-based taint flow makes the variable y tainted, the semantics of the constant function "$z \times 0 + 4$" that always results 4 irrespective of the value of z makes y untainted. The semantics approximation in the security domain, due to the abstraction, leads to a challenge in dealing with constant functions. As a partial solution, we specify rules for some simple cases of constant functions such as x - x, x *xor* x, $x \times 0$, etc. We mention one of such rules in $\mathbf{R_{con\text{-}func}}$. In this context, as a notable observation, we consider the following scenario: given the code fragment $y := read()$; $x := y$; $v := x$ *xor* y, the analysis successfully marks the variable v as tainted. Indeed, attackers may inject some malicious input containing a vulnerable control part for which the *xor* operation fails to nullify the effect, affecting the subsequent critical computation involving v.

6 EXPERIMENTAL RESULTS

This section presents experimental results on a set of benchmark codes collected from (PL/SQL, 2021). These codes represent a wide range of PL/SQL codes including explicit-flow (Prg1, Prg2, Prg09, Prg10, Prg11), implicit-flow (Prg8, Prg14, Prg15), XSS attacks (Prg13), malware attacks (Prg4, Prg12), SQL injection attacks (Prg3, Prg5, Prg6, Prg7, Prg8, Prg11, Prg12, Prg13, Prg14, Prg15), constant functions (Prg6, Prg10, Prg11), etc. All experiments are conducted using a computer system equipped with core i3, 2.60 GHz CPU, 3GB memory, and Ubuntu 20.04 operating system.

Table 4: Detail Execution Steps in K-DBTaint of Code Snippet Depicted in Figure 1 (U: untaint, T: taint).

Prog. Points	Security Types	Rules
1	*prod*|->U	R_{decl}
2	*prod*|->U, z|->U	R_{decl}
3	*prod*|->T, z|->U	R_{read}
4	*prod*|->T, z|->U	R_{if}
5	*prod*|->T, z|->T, *PName*|->T	R_{sel}

We have implemented a prototype K-DBTaint in the $\mathbb{K}$ tool (version 5.0)[2]. We have defined a full set of semantics rules (more than 430 rules) for our database language under consideration. The tool K-DBTaint accepts PL/SQL code as input from the console using $\mathbb{K}$ Framework-specific commands. Table 3 depicts the evaluation results of the benchmark codes.

[2]https://github.com/kframework/k/releases

The results of the K-DBTaint are compared with the results obtained from some of the available static taint analysis tools, such as WAP-TA (Evans and Larochelle, 2002), Pixy (Jovanovic et al., 2006), (Wassermann and Su, 2008), and (Cao et al., 2017), are reported in columns 3-7. The notations '✗$_+$' and '✗$_-$' indicate failures due to false positives and false negatives respectively, whereas '✓' indicates a successful detection of taint vulnerabilities. Observe that, due to the flow-sensitivity, context-sensitivity, the enhancement to deal with constant functions and semantics of SQL statements, K-DBTaint significantly reduces the occurrences of false alarms. In Table 4, we show how K-DBTaint successfully captures taint flows of the motivating example in Figure 1 by showing its corresponding execution steps.

7 CONCLUSIONS

In this paper, we proposed an executable rewriting logic semantics for static taint analysis of a database language in the $\mathbb{K}$ framework. The proposed analysis addressed the semantics of both the database statements and the imperative program statements together. We develop a prototype K-DBTaint in $\mathbb{K}$ based on the theoretical foundation, which allows the user to analyse PL/SQL code for integrity issues. As compared to existing works, the proposed approach has improved precision, as shown by our experimental evaluation on a set of benchmark programs. In future, we aim to add more semantic rules to cover more language features such as aggregate functions, nested queries, set operations, etc., as an extension to the current database language and we also address more semantics-based non-dependencies.

REFERENCES

Alam, M., Halder, R., Goswami, H., Pinto, J. S., et al. (2018). K-taint: an executable rewriting logic semantics for taint analysis in the k framework. In *Proc of the 13th Int. Conf. on ENASE*, pages 359–366.

Alam, M. I. and Halder, R. (2021). Formal verification of database applications using predicate abstraction. *SN Computer Science*, 2(3):1–24.

Alam, M. I., Halder, R., and Pinto, J. S. (2021). A deductive reasoning approach for database applications using verification conditions. *Journal of Systems and Software*, 175:110903.

Asăvoae, I. M. (2014). Abstract semantics for alias analysis in k. *Electronic Notes in Theoretical Computer Science*, 304:97–110.

Cao, K., He, J., Fan, W., Huang, W., Chen, L., and Pan, Y. (2017). Php vulnerability detection based on taint analysis. In *2017 6th ICRITO*, pages 436–439. IEEE.

Clavel, M. and et al. (2007). *All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic*, volume 4350. Springer-Verlag.

Evans, D. and Larochelle, D. (2002). Improving security using extensible lightweight static analysis. *IEEE software*, 19(1):42–51.

Halim, V. H. and Asnar, Y. D. W. (2019). Static code analyzer for detecting web application vulnerability using control flow graphs. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. IEEE.

Huang, W., Dong, Y., and Milanova, A. (2014). Type-based taint analysis for java web applications. In *In Proc. of Int. Conf. on Fundamental Approaches to Software Engineering*, pages 140–154. Springer.

Hunt, S. and Sands, D. (2006). On flow-sensitive security types. In *Conf. Record of the 33rd ACM SIGPLAN-SIGACT Sym. on POPL*, pages 79–90, S. California. ACM.

Jana, A., Alam, M. I., and Halder, R. (2018). A symbolic model checker for database programs. In *ICSOFT*, pages 381–388.

Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE, S&P'06*, pages pp. 258–263.

Maskur, A. F. and Asnar, Y. D. W. (2019). Static code analysis tools with the taint analysis method for detecting web application vulnerability. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6. IEEE.

Medeiros, I., Neves, N., and Correia, M. (2015). Detecting and removing web application vulnerabilities with static analysis and data mining. *IEEE Transactions on Reliability*, 65(1):54–69.

Meseguer, J. and Roşu, G. (2007). The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237.

PL/SQL(2021). Github pl/sql project. https://github. com/-topics/plsql. [Online accessed March-2021].

Roşu, G. and Şerbănută, T. F. (2010). An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434.

Su, G., Wang, F., and Li, Q. (2018). Research on sql injection vulnerability attack model. In *2018 5th IEEE Int. Conf. on CCIS*, pages 217–221. IEEE.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. (2009). Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM.

Vijayalakshmi, K. and Syed Mohamed, E. (2021). Case study: Extenuation of xss attacks through various detecting and defending techniques. *Journal of Applied Security Research*, 16(1):91–126.

Wassermann, G. and Su, Z. (2008). Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 171–180. IEEE.