K-Taint: An Executable Rewriting Logic Semantics for Taint Analysis in the K Framework

Md. Imran Alam[1], Raju Halder[1,2], Harshita Goswami[1] and Jorge Sousa Pinto[2]
[1]*Indian Institute of Technology Patna, India*
[2]*HASLab/INESC TEC & Universidade do Minho, Braga, Portugal*

Keywords: Taint Analysis, 𝕂 Framework, Information Flow, Security.

Abstract: The 𝕂 framework is a rewrite logic-based framework for defining programming language semantics suitable for formal reasoning about programs and programming languages. In this paper, we present K-Taint, a rewriting logic-based executable semantics in the 𝕂 framework for taint analysis of an imperative programming language. Our 𝕂 semantics can be seen as a sound approximation of programs semantics in the corresponding security type domain. More specifically, as a foundation to this objective, we extend to the case of taint analysis the semantically sound flow-sensitive security type system by Hunt and Sands's, considering a support to the interprocedural analysis as well. With respect to the existing methods, K-Taint supports context- and flow-sensitive analysis, reduces false alarms, and provides a scalable solution. Experimental evaluation on several benchmark codes demonstrates encouraging results as an improvement in the precision of the analysis.

1 INTRODUCTION

Taint analysis is a widely used program analysis technique that aims at averting malicious inputs from corrupting data values in critical computations of programs (Huang et al., 2014; Jovanovic et al., 2006; Tripp et al., 2009). Examples where taint attacks severely compromise security are SQL injection, cross-site scripting, buffer overflow, etc. (Jovanovic et al., 2006). The following code snippet in Figure 1 depicts one such taint attack where input supplied by a malicious source through the formal parameter '*src*' of the function '*foo*()' may affect neighboring cells of the character array '*buf*' in the memory.

```
void foo(char *src){
  char buf[20]; int i=0;
  while(i<= strlen(src)){
      buf[i] = src[i]; i++;}
  return ;}
```

Figure 1: An Example Taint Attack.

This way attackers may store some malicious data into the neighboring cells of '*buf*' which may be accessed by legitimate applications, causing unpredictable behavior.

Static taint analysis approaches, in principle, analyze the propagation of tainted values from untrusted sources to security-sensitive sinks along all possible program paths without actually executing the code (Cifuentes and Scholz, 2008; Huang et al., 2014; Jovanovic et al., 2006; Tripp et al., 2009). Of course, due to their sound and conservative nature, they often over-approximate the analysis results which, although may introduce false positives, however always establish a security guarantee: tainted data cannot be passed to security-sensitive operations.

In the context of software security, the integrity of software systems is treated as a dual of the confidentiality problem (Sabelfeld and Myers, 2006), both of which can be enforced by controlling information flows. Works in this direction have been starting with the pioneer work of Denning and Denning in (Denning and Denning, 1977) which enforces a restrictive information flow policy defined on a mathematical lattice-model of security classes partially ordered by sensitivity levels. Inspired from this, a wide range of language-based approaches are proposed in the literature, majority of which focuses on the confidentiality (Amtoft and Banerjee, 2004; Hunt and Sands, 2006; Sabelfeld and Myers, 2006; Volpano et al., 1996). Nevertheless, in the line of taint information flow addressing software integrity, the existing data-flow and point-to analysis-based approaches (Jovanovic et al., 2006; Noundou, 2015; Sridharan et al., 2011; Tripp

Alam, M., Halder, R., Goswami, H. and Sousa Pinto, J.
K-Taint: An Executable Rewriting Logic Semantics for Taint Analysis in the K Framework.
DOI: 10.5220/0006786603590366
In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018)*, pages 359-366
ISBN: 978-989-758-300-1

et al., 2009; Livshits and Lam, 2005) basically suffer from false alarms due to ignorance of the control-flow and the semantics of constant functions. Security type-system (Foster et al., 2002; Huang et al., 2014) has emerged independently as a probably most popular approach to static taint analysis in a competing manner.

In this paper, as a contribution to the same research line, we put forward a rewriting logic-based executable semantics for taint analysis in the $\mathbb{K}$ framework, considering an extension of Hunt and Sands's semantically sound flow-sensitive security type system as the basis. The $\mathbb{K}$ framework (Roşu and Şerbănută, 2010) is a rewrite logic-based formal framework for defining programming languages semantics. Such semantic definitions are directly executable in a rewriting logic language, e.g. Maude (Clavel and et al., 2007), thus support a development of verification and analysis tools at no cost.

To summarize, our main contributions are:

- We explore the power of $\mathbb{K}$ framework to define K-Taint.
- To this aim, we extend the flow-sensitive security type system proposed by Hunt and Sands's (Hunt and Sands, 2006) as the basis.
- We specify $\mathbb{K}$ rewrite rules which captures taint information propagation along all possible program paths.
- We enhance our proposed approach in terms of precision by handling pointer aliasing and constant functions.
- We present experimental evaluation results 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. Section 3 briefly introduces the $\mathbb{K}$ framework. In section 4, we extend to the case of taint analysis the Hunt and Sands's security type system. Sections 5 and 6 present the executable rewriting logic semantics in $\mathbb{K}$ designed for taint analysis. Section 7 defines the semantics rules to handle pointer aliases and constant functions. The experimental evaluation results are reported in section 8. Finally, section 9 concludes our work.

2 RELATED WORKS

Although many language-based information flow approaches addressing confidentiality exist in the literature (Sabelfeld and Myers, 2006; Hunt and Sands, 2006; Volpano et al., 1996), this section restricts the discussions only to the static taint approaches in the same line. Works on taint analysis, as a dual of confidentiality, include security type systems (Foster et al., 2002; Huang et al., 2014), flow-analysis (Evans and Larochelle, 2002; Jovanovic et al., 2006; Noundou, 2015; Scholz et al., 2008; Tripp et al., 2009), point-to analysis (Livshits and Lam, 2005; Tripp et al., 2009), etc. The flow-sensitivity in `CQual` (Foster et al., 2002) is triggered by specifying manually a partial order configuration on security qualifiers. Unfortunately, `CQual` is unable to support implicit flow-sensitivity in presence of branches. On the other hand, `SFlow` (Huang et al., 2014), a type-based taint analyzer for Java Web applications, performs type judgement based on calling context viewpoint adaption without actually flowing the context information through the called function body, which may often result false alarms. Like `CQual`, the `SFlow` also forgoes the implicit flow. As alternative solutions, taint analysis attracts many proposals on data-flow analysis (Jovanovic et al., 2006; Noundou, 2015; Sridharan et al., 2011; Tripp et al., 2009) and point-to analysis (Livshits and Lam, 2005; Tripp et al., 2009). Unfortunately, given the ignorance of control dependencies, these techniques are unable to capture indirect influence of taint information on other variables due to implicit-flow. Although the authors in (Cifuentes and Scholz, 2008; Corin and Manzano, 2012; Evans and Larochelle, 2002; Scholz et al., 2008) have considered both data- and control-dependencies, these approaches fail to address false positives in presence of constant functions, such as $x := 0 \times x$, $x := y - y$, etc. A summary of the state-of-the-art tools and techniques in the line of static taint analysis only, as compared with K-Taint, is given in Table 1.

3 THE $\mathbb{K}$ FRAMEWORK

The $\mathbb{K}$ framework provides a rewrite logic-based framework suitable for design and analysis of programming languages. 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.

To define semantics of programming language constructs, the $\mathbb{K}$ framework mainly relies on *configuration* and $\mathbb{K}$ *rewrite rules*. *Configuration* specifies the structure of the abstract machine on which programs written in that language will run and this is represented as labeled nested cells (*i.e.*, `List`, `Map`, `Bag`, `Set`, etc.). For example, consider the following configuration with three cells:

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

	K-Taint	Pixy	Taintgrind	SAINT	TAJ	Splint	Parfait	SFlow	CQual	KLEE
Semantics/Security Type System	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓
Explicit Flow	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Implicit Flow	✓	✗	✗	✗	✗	✓	✓	✗	✗	✓
Constant Functions	☑	✗	✗	✗	✗	✗	✗	✗	✗	✗
Flow-Sensitivity	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
Context-Sensitivity	✓	✓	✗	✓	✓	✗	✓	✓	✗	✓
Language Supported	Imperative (including C-like syntax)	PHP	C	C	Java	C	C	Java	C	C

$$configuration \equiv \langle\langle K\rangle_k\ \langle \mathtt{Map}[Var \mapsto Loc]\rangle_{env}\ \langle \mathtt{Map}[Loc \mapsto Val]\rangle_{store}\ \rangle_T$$

The k cell holds a list of computational tasks, that is k : $\mathtt{List}\{K, \curvearrowright\}$ 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., $env : Var \mapsto Loc$) and the *store* cell maps locations to values (i.e., $store : Loc \mapsto Val$). These cells are covered by the top cell denoted by T. $\mathbb{K}$ *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 finding address of a variable:

$$\langle \frac{\&Y}{L} \ldots\rangle_k\ \langle \ldots Y \mapsto L \ldots\rangle_{env}$$

This specifies that the next task to evaluate is a reference operator (&) on the variable Y, which results the location L in the memory based on the match in the environment cell *env*.

In the $\mathbb{K}$ framework, a language syntax is given using conventional BNF notation annotated with semantics attributes which enforces the evaluation strategy of the construct. For example, consider the following definition for arithmetic expression:

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

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:

$$\langle \frac{E_1 + E_2}{E_1 \curvearrowright \Box + E_2} \ldots\rangle_k \;\mid\; \langle \frac{E_1 + E_2}{E_2 \curvearrowright \Box + E_1} \ldots\rangle_k$$
$$\langle \frac{V_1 \curvearrowright \Box + E_2}{V_1 + E_2} \ldots\rangle_k \;\mid\; \langle \frac{V_2 \curvearrowright E_1 + \Box}{E_1 + V_2} \ldots\rangle_k$$

Here, V_1 and V_2 are the evaluated results of the expressions E_1 and E_2 respectively. The construct $\Box$ (HOLE) is a place-holder that will be replaced by the result of the evaluated term or sub-term.

4 EXTENDING HUNT AND SANDS'S TYPE SYSTEM TO TAINT ANALYSIS

In this section, we define a type-based taint analysis for imperative programming language supporting functions, arrays, pointers, etc. Table 2 depicts the abstract syntax of the basic language under consideration, where D and E denote respectively a sequence of declarations $\langle \tau\ id_1,\ \tau\ id_2, \ldots\rangle$ and a sequence of arithmetic expressions $\langle E_1,\ E_2, \ldots\rangle$ respectively.

Table 2: Abstract Syntax of the Language.

$$\begin{array}{l}
E ::= n \mid id \mid \&id \mid {}^*E \mid id[E] \mid E\ op\ E \mid (E), \qquad \text{where } op \in \{+,-,\times,/\}\\
B ::= true \mid false \mid E\ rel\ E \mid \neg B \mid B\ AND\ B \mid B\ OR\ B,\\
\qquad \text{where } rel \in \{\geqslant,\leqslant,<,>,==\}\\
\tau ::= int \mid float \mid char \mid bool \mid \tau[n] \mid \tau^*\\
D ::= \tau\ id\\
A ::= id := E \mid {}^*E := E \mid id[E] := E \mid id := read()\\
C ::= skip; \mid D; \mid A; \mid defun\ id(\vec{D})\{C\} \mid call\ id(\vec{E}); \mid return; \mid return\ E; \mid\\
\qquad C_1\ C_2 \mid if\ B\ then\ \{C\} \mid if(B)\ then\ \{C_1\}\ else\ \{C_2\} \mid while(B)\ do\ \{C\}
\end{array}$$

Our work mainly motivated by the security type system proposed in (Hunt and Sands, 2006), which is primarily proposed to detect possible leakage of sensitive information from programs. Unlike other similar type systems, (Hunt and Sands, 2006) is featured with flow-sensitivity. We extend this flow-sensitive type system for the purposes of our taint analysis with an additional support to the context-sensitivity in case of inter-procedural code, leading to a significant improvement in the precision. This is depicted in Figure 2. Although our proposal is scalable enough, we consider, for the sake of simplicity, a simple instance of the problem involving two security types: *taint* and *untaint*. We will work with the flow 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 $untaint \sqsubseteq taint$.

The typing judgements are of the form $pc \vdash \Gamma\ \{C\}\ \Gamma'$, where $pc \in \mathbb{S}$ represents the security context used to address implicit flow, C is the program

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

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

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

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

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

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

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

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

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

statements, and Γ, $\Gamma^{'}$: Variables $\rightarrow \mathbb{S}$ are environments. The security type $\mathtt{T}$ of expression E (denoted $\Gamma \vdash E : \mathtt{T}$) is defined simply by the least upper bound of the types of all free variables ($\mathtt{FV}$) in E, where $\sqcup$ represents the join operation in the security lattice *SD*. The typing rules ensure that for any given C, Γ, and pc there is an environment $\Gamma^{'}$ such that $pc \vdash \Gamma\ \{C\}\ \Gamma^{'}$ is derivable. We use the notation $\Gamma \vdash \vec{E} :\ \vec{\mathtt{T}}$ to denote the sequence of type judgements $\langle \Gamma \vdash E_1 :\ \mathtt{T}_1,\ \Gamma \vdash E_2 :\ \mathtt{T}_2, \dots \rangle$. Similarly, $\Gamma[\![\vec{id} \mapsto \vec{\mathtt{T}}]\!]$ denotes a sequence of type substitutions $\langle \Gamma[\![id_1 \mapsto \mathtt{T}_1]\!], \Gamma[\![id_2 \mapsto \mathtt{T}_2]\!], \dots \rangle$. Observe that reading inputs from unsanitized sources through *read*() always makes the corresponding variables tainted. The rule for function calls ensures the context-sensitivity in the system, where *getParam*() extracts formal parameters from the function definition. The analyzer associates security types with program constructs treating source variables as tainted, and then propagates their types along the program code to determine application's security. The flow sensitive typing rules in case of branching statements leverage the lattice-based operations on the security domain, resulting into conservative analysis results.

5 $\mathbb{K}$ SPECIFICATION OF SECURITY TYPE SYSTEM: A ROADMAP

This section provides a roadmap to specify $\mathbb{K}$ rewrite rules corresponding to the typing rules depicted in Figure 2. Let us consider the typing judgement $pc \vdash \Gamma\{C\}\Gamma^{'}$ which specifies that the security environment $\Gamma^{'}$ is derived by executing the statement C on the security environment Γ under the program's security context pc. To capture this, let us give algebraic representations of Γ, $\Gamma^{'}$, C and pc in $\mathbb{K}$ by defining a configuration consisting of three cells – k cell to contain program statements as a sequence of computations, *env* cell to hold the security levels of program variables and *context* cell to capture current program context pc in the security type domain – as follows: $\big\langle \langle K \rangle_k\ \langle \mathtt{Map} \rangle_{env}\ \langle \mathtt{Map} \rangle_{context} \big\rangle_T$.

The corresponding $\mathbb{K}$ rewrite rule capturing the type judgement $pc \vdash \Gamma\{C\}\Gamma^{'}$ is specified as:

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

The symbol "…" appearing in the k cell represents remaining computations. As a result of the execution of C which eventually be consumed (denoted by dot), 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.

Similarly, the derivation rule $\Gamma \vdash E : \mathtt{T}$ specified as $\langle \dots\ E \mapsto \mathtt{T}\ \dots \rangle_{env}$ means that expression E has the security type $\mathtt{T}$ somewhere in the environment *env*. Each security type rule is written based on a number of *premise* judgements $\Gamma_i \vdash \zeta_i$ above a horizontal line, with a single *conclusion* judgement $\Gamma \vdash \zeta$ below the line. For example, given the type rule $\frac{\Gamma \vdash M : Nat \quad \Gamma \vdash N : Nat}{M + N : Nat}$, the corresponding $\mathbb{K}$ rule is defined as: $\langle \frac{M+N}{M +_{Nat} N}\ \dots \rangle_k\ \langle \dots\ M \mapsto Nat, N \mapsto Nat\ \dots \rangle_{env}$ where M: Nat, N: Nat, and $+_{Nat}$: $Nat \times Nat \mapsto Nat$. Having this setting as foundation, in the next section we specify $\mathbb{K}$ rewrite rules for static taint analysis of imperative language in the abstract security type domain $\mathbb{S}$.

6 $\mathbb{K}$ REWRITING LOGIC SEMANTICS FOR TAINT ANALYSIS

This section introduces an executable rewriting logic semantics in the $\mathbb{K}$ framework for taint analysis of our language under consideration. As mentioned earlier, our semantics can be seen as a sound semantics approximation in the security type domain.

To this aim, we consider the following $\mathbb{K}$ modeling of the program configuration on which the semantics is defined:

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

As mentioned earlier, the special cell $\langle\rangle_k$ contains the list of computation tasks of a special sort K separated by the associative sequentialization operator $\curvearrowright$.

The environment cell *env* maps variables (including pointers variables) to their security types. The current program context *pc* over the security domain is captured in *context* cell. The λ-*Def* cell supports interprocedural feature holding the bindings of function names (when defined) to their lambda abstraction. All the function calls are controlled by *control* cell maintaining a stack-based context switching using *fstack* cell. The cells *in* and *out* are used to perform standard input-output operations. To avoid false negatives in the analysis-results, we consider *ptr-alias* cell which maintains pointer aliasing information in *alias* cell. The *ptr* cell is aimed to separate pointer variables from other variables to assist the alias analysis.

Figure 3 depicts the semantics rewrite rules for taint analysis in the $\mathbb{K}$ definitional framework. We label the defined rules by $\mathtt{R}_{\_}$ for future reference. These rules captures both the explicit and implicit flow sensitivity, the context-sensitivity in presence of function calls, the semantics of constant functions, pointer aliases, etc. Let us explain these rules in detail.

Declaration, Input, Lookup and Assignment: The first rule $\mathbf{(R_{1a})_{decl}}$ 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 $\mathbf{(R_{1b})_{read}}$. The lookup rule $\mathbf{(R_2)_{lookup}}$ replaces the variable term appearing on top of k cell by its security type by looking into the environment cell. Note that the look up rule for constant terms, although we do not mention here, always returns *untaint*. As defined in rule $\mathbf{(R_{3a})_{ar\text{-}op}}$, the security types of expressions are sound approximated by least upper bound (defined in rule $\mathbf{(R_8)_{join}}$) of their component-terms security types. Rule $\mathbf{(R_{3b})_{asg}}$ 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. $\mathtt{T}$) and program's current security context *pc* in the *context* cell. The assignment is then replaced by an empty computation.

Conditional or Iteration: The presence of condition B in simple *if*- or *while*-statement gives rise to the following two: (*i*) implicit flow of taint information based on the security type of B, and (*ii*) multiple execution paths with the possibility of entering into the *if*- or *while*-block. The former is achieved by updating the security context μ in the *context* cell based on the security type of B and the later is achieved by following $restore_c(\mu)$ and $approx(\rho)$. These are depicted in rules $\mathbf{(R_4)_{if}}$, $\mathbf{(R_6)_{while}}$, $\mathbf{(R_{9a})_{restore}}$ and $\mathbf{(R_{9b})_{approx}}$.

$\mathbf{(R_{1a})_{decl}}:\ \langle \frac{\tau\ id}{.}\ \ldots\rangle_k \langle \frac{\rho}{\rho[id \leftarrow \mathtt{T}:Type]}\rangle_{env}$ $\quad \mathbf{(R_{1b})_{read}}:\ \langle \frac{read(\,)}{taint}\ \ldots\rangle_k$

$\mathbf{(R_2)_{lookup}}:\ \langle \frac{id}{\mathtt{T}:Type}\ \ldots\rangle_k \langle \ldots\ id \mapsto \mathtt{T}:Type\ \ldots\rangle_{env}$

$\mathbf{(R_{3a})_{ar\text{-}op}}:\ \langle \frac{\mathtt{T}_1:Type\ op\ \mathtt{T}_2:Type}{\mathtt{T}_1:Type \sqcup \mathtt{T}_2:Type}\ \ldots\rangle_k$

$\mathbf{(R_{3b})_{asg}}:\ \langle \frac{id\ :=\ \mathtt{T}:Type}{.}\ \ldots\rangle_k \langle \ldots\ \rho[id \mapsto \frac{\_}{\mu(pc)\sqcup\mathtt{T}:Type}]\ \ldots\rangle_{env}\ \langle\mu\rangle_{context}$

$\mathbf{(R_4)_{if}}:\ \langle \frac{if(B:\mathtt{T})\ then\ \{C\}}{C \curvearrowright restore_c(\mu) \curvearrowright approx(\rho)}\ \ldots\rangle_k \langle \frac{\mu}{\mu[pc \leftarrow \mu(pc)\sqcup\mathtt{T}]}\rangle_{context}\ \langle\rho\rangle_{env}$

$\mathbf{(R_{5a})_{if\text{-}else}}:$
$\langle \frac{if(B:\mathtt{T})\ then\ \{C_1\}\ else\ \{C_2\}}{C_1 \curvearrowright exitIf() \curvearrowright restore_{env}(\rho) \curvearrowright C_2 \curvearrowright exitElse() \curvearrowright restore_c(\mu)}\ \ldots\rangle_k\ \langle\rho\rangle_{env}$ $\langle \frac{\mu}{\mu[pc \leftarrow \mu(pc)\sqcup\mathtt{T}]}\rangle_{context}$

$\mathbf{(R_{5b})_{exit\text{-}if}}:\ \langle \frac{exitIf()}{save(\rho)}\ \ldots\rangle_k\langle\rho\rangle_{env}$ $\quad \mathbf{(R_{5c})_{exit\text{-}else}}:\ \langle \frac{exitElse()}{approx(save(\rho))}\ \ldots\rangle_k$

$\mathbf{(R_6)_{while}}:$
$\langle \frac{while(B:\mathtt{T})\ do\ \{C\}}{C \curvearrowright restore_c(\mu) \curvearrowright approx(\rho) \curvearrowright fixpoint(B,C,\rho)}\ \ldots\rangle_k\ \langle\rho\rangle_{env}$ $\langle \frac{\mu}{\mu[pc \leftarrow \mu(pc)\sqcup\mathtt{T}]}\rangle_{context}$

$\mathbf{(R_{7a})_{fun\text{-}decl}}:$
$\langle \frac{defun\ Func\_name(Params)\{C\}}{.}\ \ldots\rangle_k\langle\langle \frac{\psi}{\psi[Func\_name \leftarrow lambda(Params,C)]}\rangle_{\lambda\text{-}Def}\rangle_{control}$

$\mathbf{(R_{7b})_{fun\text{-}lookup}}:$
$\langle \frac{call\ Func\_name(Es:Ts)}{lambda(Params,C)(Es:Ts)}\ \ldots\rangle_k\langle\langle \ldots\ Func\_name \mapsto lambda(Params, C)\ \ldots\rangle_{\lambda\text{-}Def}\rangle_{control}$

$\mathbf{(R_{7c})_{fun\text{-}call}}:$
$\langle \frac{lambda(Params,C)(Es:Ts)\ \curvearrowright K}{McDecls(Params,Ts) \curvearrowright C \curvearrowright return;}\ \ldots\rangle_k\langle\langle \frac{.List}{[ListItem(\rho,K,Ctr)]}\ \ldots\rangle_{fstack}\ Ctr\rangle_{control}\langle\rho\rangle_{env}$

$\mathbf{(R_{7d})_{fun\text{-}ret}}:$
$\langle \frac{return(\mathtt{T}:Type); \curvearrowright\ \_}{\mathtt{T}:Type\ \curvearrowright\ K}\ \ldots\rangle_k\ \langle\langle \frac{[ListItem(\rho,K,Ctr)]}{.List}\ \ldots\rangle_{fstack}\ (\frac{\_}{Ctr})\rangle_{control}\ \langle \frac{\_}{\rho}\rangle_{env}$

$\mathbf{(R_8)_{join}}:$

$$\langle \mathtt{T}_1:Type \sqcup \mathtt{T}_2:Type\ \ldots\rangle_k = \begin{cases} \langle \frac{\mathtt{T}_1:Type\sqcup\mathtt{T}_2:Type}{untaint}\ \ldots\rangle_k, & if\ \mathtt{T}_1 = \mathtt{T}_2 = untaint \\ \langle \frac{\mathtt{T}_1:Type\ \sqcup\ \mathtt{T}_2:Type}{taint}\ \ldots\rangle_k, & otherwise \end{cases}$$

$\mathbf{(R_{9a})_{restore}}:\ \langle \frac{restore_c(\mu)}{.}\ \ldots\rangle_k\ \langle \frac{\_}{\mu}\rangle_{context}$

$\mathbf{(R_{9b})_{approx}}:\ \langle \frac{approx(\rho)}{.}\ \ldots\rangle_k\ \langle \frac{\rho_c}{\rho\sqcup\rho_c}\rangle_{env}$

Figure 3: $\mathbb{K}$ rewrite rules for executable semantics-based taint analysis.

Specifically, $restore_c(\mu)$ 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. Observe that the least fixed point solution in case of "*while*" is achieved by defining an auxiliary function *fixpoint*() as follows: either (1) $\langle \frac{fixpoint(B,C,\rho_i)}{.}\ \ldots\rangle_k\langle\rho'_i\rangle_{env}$ when $\rho_i = \rho'_i$, or (2) $\langle \frac{fixpoint(B,C,\rho_i)}{while(B)\ do\ \{C\}}\ \ldots\rangle_k\langle\rho'_i\rangle_{env}$ when $\rho_i \neq \rho'_i$. Note that the first case indicates that the computation reaches the fix-point and therefore the computation is consumed. If not, then the iteration continues as shown in the second case.

The soundness of the analysis in presence of *if*-

else is guaranteed by approximating the analysis-results from both the branches C_1 and C_2 (a *may*-analysis), as depicted in rule $\mathbf{(R_{5a})_{if\text{-}else}}$, $\mathbf{(R_{5b})_{exit\text{-}if}}$ and $\mathbf{(R_{5c})_{exit\text{-}else}}$. Observe that both the branches are executed over the same environment (using $restore_{env}(\rho)$ which restores environment and is defined similar to the rule $\mathbf{(R_{9a})_{restore}}$) which occurs at the entry point of *if-else*.

Dealing with Functions: We specify the rules $\mathbf{(R_{7a})_{fun\text{-}decl}}$, $\mathbf{(R_{7b})_{fun\text{-}lookup}}$, $\mathbf{(R_{7c})_{fun\text{-}call}}$, and $\mathbf{(R_{7d})_{fun\text{-}ret}}$ to handle interprocedural feature in our analysis. For each function definition, the rule $\mathbf{(R_{7a})_{fun\text{-}decl}}$ creates a *lambda* abstraction binding it to the function name in the $\langle\ \rangle_{\lambda\text{-}Def}$ cell. Coming across a function call, the rule $\mathbf{(R_{7b})_{fun\text{-}lookup}}$ replaces this function call by its *lambda* abstraction. We use a helper function *McDecls*() which recursively extracts the formal parameters in the called function and assigns to them the security types of the actual parameters in the calling function, as shown below:

$$\langle \frac{McDecls((param,\ params),(Type,\ Types))}{param := \ Type; \curvearrowright \ McDecls(params,\ Types)} \ \ldots\rangle_k$$

Note that the function *McDecls*() enforces the context sensitivity by treating same function call with different parameters differently. As usual, *McDecls*() is followed by a sequence of computations C in the function body and then by a *return* statement. When a function returns the result by explicitly mentioning it as "*return E*" statement, the rule $\mathbf{(R_{7d})_{fun\text{-}ret}}$ is applied which returns the security type of the resultant expression and restores the previous context to start the execution of remaining tasks specified as $\langle \frac{[ListItem(\rho,K,Ctr)]}{.List} \ \ldots\rangle_{fstack}$.

7 DEALING WITH POINTERS ALIASING AND CONSTANT FUNCTIONS

The rules defined for implicit flow in Figure 3 are unsound in presence of pointers. More precisely, given an assignment computation $id := E$, the correctness of the analysis is established by ensuring the update of the security type not only for *id* but also for all of its aliases by the security type of E. To handle this scenario, the nested cells *alias* and *ptr* are designed to store the alias information and the set of pointer variables. The semantics rules are depicted in Figure 4. In case of a simple assignment $id := E$ when *id* is not a pointer variable, the rule $\mathbf{(R_{10a})_{alias}}$ triggers the update of the security type of *id* and its direct pointers identified in the *alias* cell by the security type of E. As a consequence of it, the rule $\mathbf{(R_{10b})_{alias}}$ then performs the same update action to all of its indirect pointers as well. The reason behind this is to ensure that all pointers which are pointing, directly or indirectly, to a taint value must be tainted, leading to a sound analysis. Similarly, rules $\mathbf{(R_{10c})_{alias}}$, $\mathbf{(R_{10d})_{alias}}$, and $\mathbf{(R_{10e})_{alias}}$ refer to the assignment of security types to pointer variables and the creation of new alias information in the *alias* cell. This is to note that the author in (Asăvoae, 2014) integrated the alias analysis in $\mathbb{K}$ as an instantiation of the collecting semantics where alias information can be extracted from the *alias* cell on demand-driven way. Our approach follows the same line, but in a much simpler way without considering an exhaustive execution in worst case scenario.

$$\mathbf{(R_{10a})_{alias}}\text{:}\ \langle \frac{id := E : T}{id := T \curvearrowright \ P := T} \ \ldots\rangle_k \ \langle\langle \ldots \ P \mapsto PointsTo(id) \ \ldots\rangle_{alias} \ \langle\eta\rangle_{ptr}\rangle_{ptr\text{-}alias} \ \langle\rho\rangle_{env} \ \text{ when } P \in \eta$$

$$\mathbf{(R_{10b})_{alias}}\text{:}\ \langle \frac{P := T}{R := T} \ \ldots\rangle_k \ \langle\langle \ldots \ R \mapsto PointsTo(P) \ \ldots\rangle_{alias} \langle\eta\rangle_{ptr}\rangle_{ptr\text{-}alias} \ \langle \ldots \ P \mapsto \frac{-}{T} \ \ldots\rangle_{env} \text{ when } P \in \eta$$

$$\mathbf{(R_{10c})_{alias}}\text{:}\ \langle \frac{P := \&Q : T}{P := T} \ \ldots\rangle_k \ \langle\langle \ \xi[P \mapsto \frac{-}{PointsTo(Q)}] \ \rangle_{alias} \ \langle\eta\rangle_{ptr} \ \rangle_{ptr\text{-}alias} \text{ when } P \in \eta$$

$$\mathbf{(R_{10d})_{alias}}\text{:}\ \langle \frac{P := Q : T}{P := T} \ \ldots\rangle_k \ \langle\langle \ldots \ Q \mapsto PointsTo(S) \ \ldots \ P \mapsto \frac{-}{PointsTo(S)} \ \ldots\rangle_{alias} \langle\eta\rangle_{ptr}\rangle_{ptr\text{-}alias} \ \text{ when } P \in \eta$$

$$\mathbf{(R_{10e})_{alias}}\text{:}\ \langle \frac{P :=^{*} Q : T}{P := T} \ \ldots\rangle_k \ \langle\langle \ldots \ Q \mapsto PointsTo(S) \ \ldots \ S \mapsto PointsTo(M) \ \ldots \ P \mapsto \frac{-}{PointsTo(M)} \ \ldots\rangle_{alias} \langle\eta\rangle_{ptr}\rangle_{ptr\text{-}alias} \text{ when } P \in \eta$$

$$\mathbf{(R_{11})_{con\text{-}func}} : \ \langle id_1 * id_2 \ \ldots\rangle_k = \begin{cases} \langle \frac{id_1 * id_2}{untaint} \ \ldots\rangle_k \ when \ id_1 = zero \\ \qquad\qquad\qquad or \ id_2 = zero \\ \langle \frac{id_1 * id_2}{id_1 *_{Type} id_2} \ \ldots\rangle_k \ otherwise \end{cases}$$

Figure 4: $\mathbb{K}$ rules for pointer aliasing and constant functions.

Apart from this, capturing the semantics of constant functions has a significant impact on the precision of taint analysis. For example, consider the statement $v := x \times 0 + 4$, where x is a tainted variable. It is worthwhile to observe that, although the syntax-based taint flow makes the variable v tainted, the semantics of the constant function "$x \times 0 + 4$" that always results 4 irrespective of the value of x makes v 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 $\mathbf{(R_{11})_{con\text{-}func}}$. In this context, as a notable observation, we consider the following scenario: given the code fragment

Table 3: Taint Analysis on Benchmark Programs Set (SecuriBench, 2006; Cavallaro et al., 2008; Vogt et al., 2007; Evans et al., 2003; Russo and Sabelfeld, 2010) (✓: `Passed`, ✗$_{+}$: `False Positives`, ✗$_{-}$: `False negatives`).

Progs.	**Descriptions**	**K-Taint**	**Splint (Evans and Larochelle, 2002)**	**Pixy (Jovanovic et al., 2006)**	**SFlow (Huang et al., 2014)**	**CQual (Foster et al., 2002)**
Prog1	Explicit Flow	✓	✓	✓	✓	✓
Prog2	Implicit Flow	✓	✗$_{-}$	✗$_{-}$	✗$_{-}$	✗$_{-}$
Prog3	Malware Attack	✓	✗$_{-}$	✗$_{-}$	✗$_{-}$	✗$_{-}$
Prog4	XSS Attack	✓	✗$_{-}$	✗$_{-}$	✗$_{-}$	✗$_{-}$
Prog5	Buffer Overflow	✗$_{+}$	✓	✓	✗$_{+}$,✗$_{-}$	✗$_{-}$
Prog6	Constant Function "`subtraction`"	✓	✗$_{+}$	✗$_{+}$	✗$_{+}$	✗$_{+}$
Prog7	Program consists of multiple functions	✓	✗$_{-}$, ✗$_{+}$	✗$_{-}$	✓	✗$_{-}$
Prog8	Program with context-sensitivity	✓	✗$_{-}$, ✗$_{+}$	✓	✓	✗$_{+}$
Prog9	Factorial Program	✓	✗$_{-}$	✗$_{-}$	✗$_{-}$	✗$_{-}$
Prog10	Binary Search	✗$_{+}$	✗$_{-}$	✗$_{-}$	✗$_{-}$	✗$_{-}$
Prog11	Merge Sort	✗$_{+}$	✗$_{-}$	✗$_{-}$	✗$_{-}$	✗$_{-}$
Prog12	Program with flow-sensitivity	✓	✗$_{-}$	✓	✗$_{-}$	✗$_{-}$
Prog13	Swapping of two numbers using pointers	✓	✓	✓	✓	✗$_{-}$

$y := read();\ x := y;\ v := x\ xor\ y$, 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.

We end this section stating the fundamental results on K-Taint. We skip the proofs for brevity.

Theorem 1 (Soundness). *The semantics defined in the K-Taint is a sound approximation of the concrete collecting semantics with respect to variables security properties.*

Theorem 2 (Termination). *Any execution in the K-Taint is always finite.*

Consider the security type domain $\mathbb{S}$ of n security levels with order relation $\sqsubseteq$. Given $s_i, s_j \in \mathbb{S}$, $s_i \sqsubseteq s_j$ denotes that s_i is more trusted than s_j. For example, $untaint \sqsubseteq taint$.

Definition 1 (s_t-indistinguishability). *Let X be the set of program variables participating in critical computations of a program P. Let $s_t \in \mathbb{S}$ be the permissible security level for critical computations in P, meaning that any variable in X with security level $s \sqsubseteq s_t$ can securely participate in the critical computations. Given two type environments ρ_1 and ρ_2, we say that they are s_t-indistinguishable (denoted $\langle\rho_1\rangle_{env} \approx_{s_t} \langle\rho_2\rangle_{env}$) iff $\forall x \in X.\ \rho_1(x) \sqsubseteq s_t \wedge \rho_2(x) \sqsubseteq s_t$, meaning that they agree on the sensitivity levels for security-sensitive variables.*

Theorem 3 (Non-interference). *Given any two type environments ρ_1 and ρ_2 such that $\langle\rho_1\rangle_{env} \approx_{s_t} \langle\rho_2\rangle_{env}$. A program P is secure iff $\mathbb{K}$-executions of P on the above two environments result into the environments $\langle\rho'_1\rangle_{env}$ and $\langle\rho'_2\rangle_{env}$ respectively satisfying $\langle\rho'_1\rangle_{env} \approx_{s_t} \langle\rho'_2\rangle_{env}$.*

8 EXPERIMENTAL ANALYSIS

We have implemented the full set of semantics rules (more than 200 rules) in the $\mathbb{K}$ tool (version 4.0) for our imperative language under consideration and performed experiments on a set of benchmark codes collected from (SecuriBench, 2006; Cavallaro et al., 2008; Evans et al., 2003; Russo and Sabelfeld, 2010; Vogt et al., 2007) and on some well-known programs[1,2]. A wide range of representative programs are considered, including explicit flow, implicit flow due to conditional or iteration, XSS attacks, malware attacks, merge sort, binary search, factorial, constant functions, etc. Since K-Taint supports C-like language, it accepts the benchmark C-codes files as input from the console using $\mathbb{K}$ Framework-specific commands. The evaluation results are shown in Table 3. The results of K-Taint are compared with the results obtained from some of the available static taint analysis tools, such as `Splint` (Evans and Larochelle, 2002), `Pixy` (Jovanovic et al., 2006), `SFlow` (Huang et al., 2014), and `CQual` (Foster et al., 2002), 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 and the enhancement to deal with constant functions, K-Taint significantly reduces the occurrences of false alarms. The authors in (Cavallaro et al., 2008), (Russo and Sabelfeld, 2010) and (Vogt et al., 2007) highlighted some special cases where their approaches fail. We consider those special cases (shown as `Prog3`, `Prog4` and `Prog12` in Table 3), and observed that K-Taint successfully captures those taint flows.

[1] The online version of the $\mathbb{K}$ tool is available at *http://www.kframework.org/tool/run/*

[2] The full set of semantics rules in K-taint and the evaluation results on the test codes are available for download at *www.iitp.ac.in/∼halder/ktaint*

9 CONCLUSION

This paper presented an executable rewriting logic semantics for static taint analysis of an imperative programming language in the $\mathbb{K}$ framework. The proposed approach has improved precision with respect to the existing techniques, as shown by our experimental evaluation on a set of well-known benchmark programs. We made the full set of semantics rules and the experimental data available for download. We are currently investigating how to integrate in the proposed analyzer a preprocessing phase which allows to address specific cases where exact variables values may improve the precision. We consider in our future endeavor more semantic rules to cover more language features as an extension to the current imperative language and we also address more semantics-based non-dependencies.

ACKNOWLEDGEMENT

This work is partially supported by the research grant (SB/FTP/ETA-315/2013) from the Science and Engineering Research Board (SERB), Department of Science and Technology, Government of India.

REFERENCES

Amtoft, T. and Banerjee, A. (2004). Information flow analysis in logical form. In *SAS*, volume 3148, pages 100–115. Springer.

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

Cavallaro, L., Saxena, P., and Sekar, R. (2008). On the limits of information flow techniques for malware analysis and containment. In *Proc. of Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer.

Cifuentes, C. and Scholz, B. (2008). Parfait: designing a scalable bug checker. In *Proc. of the 2008 workshop on Static analysis*, pages 4–11. ACM.

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

Corin, R. and Manzano, F. A. (2012). Taint analysis of security code in the klee symbolic execution engine. In *ICICS*, pages 264–275. Springer.

Denning, D. E. and Denning, P. J. (1977). Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513.

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

Evans, D., Larochelle, D., and Evans, D. (2003). Splint manual: Version 3.1.1-1. http://lclint.cs.virginia.edu/manual/manual.html.

Foster, J. S. et al. (2002). Cqual user's guide. *University of California, Berkeley, version 0.9 edition*.

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.

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

Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium*, volume 14, pages 18–18.

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

Noundou, X. N. (2015). Saint: Simple static taint analysis tool users manual. https://archive.org/details/saint_201507.

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.

Russo, A. and Sabelfeld, A. (2010). Dynamic vs. static flow-sensitive security analysis. In *23rd IEEE Computer Security Foundations Symposium*, pages 186–199. IEEE.

Sabelfeld, A. and Myers, A. C. (2006). Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19.

Scholz, B., Zhang, C., and Cifuentes, C. (2008). User-input dependence analysis via graph reachability. Technical Report SMLI TR-2008-171, Mountain View, CA, USA.

SecuriBench (2006). Stanford securibench micro. http://suif.stanford.edu/∼livshits/work/securibench-micro/.

Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., and Berg, R. (2011). F4f: taint analysis of framework-based web applications. *ACM SIGPLAN Notices*, 46(10):1053–1068.

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.

Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. (2007). Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, volume 2007, page 12.

Volpano, D., Irvine, C., and Smith, G. (1996). A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187.