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
{imran.pcs16, harshita.mtcs14, halder}@iitp.ac.in, jsp@di.uminho.pt

Abstract. The $\mathbb{K}$ 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 $\mathbb{K}$ framework for taint analysis of an imperative programming language. Our $\mathbb{K}$ 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, considering a support to the interprocedural analysis as well.

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 [22, 24, 38]. Examples where taint attacks severely compromise security are SQL injection, cross-site scripting, buffer overflow, etc. [24]. 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. 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 [9, 22, 24, 27, 38]. 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 [35], 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 [13] which enforces a restrictive information flow policy defined on a mathematical lattice-model of security

```
1. void foo(char *src){
2. char buf[20];
3. int  i=0;
4. while(i<= strlen(src)){
5.       buf[i] = src[i];
6.       i = i + 1;}
7. return;}
```

Fig. 1: An Example of Taint Attack

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 [2, 23, 35, 40]. Nevertheless, in the line of taint information flow addressing software integrity, the existing data-flow and point-to analysis-based approaches [24, 31, 37, 38, 27] basically suffer from false alarms due to ignorance of the control-flow and the semantics of constant functions. Security type-system [18, 22] has emerged independently as a probably most popular approach to static taint analysis in a competing manner. A close observation by Mantel and Sudbrock [28] reveals a key insight that a security type system enriched with flow-sensitivity establishes an equivalent power as of the dependence graph-based analyses. Connecting with other formal techniques, Hunt and Sands [23] show that the type-based security system defined by them is in fact equivalent to the program logic defined in [2]. Interestingly, an attempt to formalize type-system as an abstract semantics approximating collecting semantics of imperative programs is presented in [12]. Thanks to continuous efforts enriching type-based security systems to be powerful enough by incorporating more language features including context-sensitivity, flow-sensitivity, etc.

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 [33] is a rewrite logic-based formal framework for defining programming languages semantics. Interestingly, the semantics expressed in $\mathbb{K}$ has a rigorous meaning as a term rewriting system, and is suitable for formal reasoning about programs and programming languages. Such semantic definitions are directly executable in a rewriting logic language, e.g. Maude [10], thus support a development of verification and analysis tools at no cost. $\mathbb{K}$ is inspired from the Rewriting Logic Semantics Project [30] aimed for the unification of algebraic denotational semantics and structural operational semantics. Many real world programming languages semantics are already defined in $\mathbb{K}$ such as C [14], PHP [17], JAVA [7], Python [20], K-Scheme [29], etc. Some notable formal analysis and verification approaches in $\mathbb{K}$ are introduced in [4–6].

Hunt and Sands's security type system was primarily designed for secure information flow for confidentiality policy [23]. We extend this to the case of

taint analysis by enriching it with a support to context-sensitivity in the case of interprocedural languages. Starting with this extension as a foundation, our $\mathbb{K}$ semantics for taint analysis can be seen as a sound approximation of the program semantics in the corresponding security type domain. The notable achievement of our approach is the support of key factors, such as flow-sensitivity, context-sensitivity, pointer aliases, constant functions semantics, etc. [21], improving the precision significantly.

To summarize, our main contributions in this paper are:

- We explore the power of $\mathbb{K}$ framework to define K-Taint – an executable rewriting logic semantics for static taint analysis of an imperative programming language.
- To this aim, we extend the flow-sensitive security type system proposed by Hunt and Sands [23] as the basis. In this setting, we provide a support to the context-sensitivity in case of interprocedural code.
- We specify $\mathbb{K}$ rewrite rules on a suitable configuration as formal taint semantics which captures taint information propagation along all possible program paths, respecting the security type system.
- We enhance our proposed approach in terms of precision by providing a solution that allows us to handle pointer aliasing and constant functions in many useful situations. To this end we introduce new semantics rules, taking advantage of the support for modularity in the $\mathbb{K}$ framework.
- We present experimental evaluation results to establish the effectiveness of our approach. Experiments demonstrate that our technique improves the precision w.r.t. existing works by reducing false alarms.

In general, we choose the $\mathbb{K}$ framework to define taint semantics for the following reasons: (*i*) The $\mathbb{K}$ framework supports the important aspect of *modularity* in programming language definition and design – one can add or remove programming language features without a need to modify any of the other, unrelated language features [33], which contributes to increased scalability; (*ii*) The $\mathbb{K}$ framework is built on top of the Maude rewrite tool and thus supports term rewriting. $\mathbb{K}$ rewrite rules, in particular, generalize conventional rewrite rules by making it explicit which parts of the term they read-only, write-only, or do not care about. (*iii*) Semantics in the $\mathbb{K}$ framework are directly executable, thus supports a test-driven development.

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 Sand'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 approac-hes addressing confidentiality exist in the literature [2, 13, 23, 35, 40], this section restricts the discus-

	K-Taint	Pixy [24]	Taintgrind [25]	SAINT [31]	TAJ [38]	Splint [15]	Parfait [9]	SFlow [22]	CQual [18]	KLEE [11]
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

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

sions only to the static taint approaches in the same line. Works on taint analysis, as a dual of confidentiality, include security type systems [18, 22], flow-analysis [15, 24, 25, 31, 36, 38], point-to analysis [27, 38], etc. The flow-sensitivity in `CQual` [18] 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` [22], a type-based taint analyzer for Java Web applications, performs type judgment 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 [24, 25, 31, 37, 38] and point-to analysis [27, 38]. 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 [9, 11, 15, 36] 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. In the context of modern component-based systems, the semantics-based information flow approach in [32] ensures that all codes responsible for security sensitive actions are sufficiently authorized in order to respect the integrity policy which is extracted from a given access control policy. Some recently proposed taint analysis of Android applications among many others can be found in [3, 19, 26].

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 : Preliminaries

The $\mathbb{K}$ framework provides a rewrite logic-based framework suitable for design and analysis of programming languages. Inspired by rewrite-logic semantics project [30], this framework unifies algebraic denotational semantics and op-

erational 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:

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

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., $env : Var \mapsto Loc$) 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*. This simple example illustrates various important features of the $\mathbb{K}$ framework. First, $\mathbb{K}$ rules mention only those cells which are relevant, whereas unspecified cells infrastructures can automatically be inferred, making the rule more robust and modular. Second, in order to omit remaining part of a cell one may write "...". For example, in the k cell we are only interested in the current $\&Y$, but not the rest of the context. Finally, we evaluate only part of the task, but neither the context nor the environment change. Observe that the traditional rewrite rule equivalent to the above rule is specified as:

$$\langle \&Y \curvearrowright K \rangle_k \langle \rho_1, Y \mapsto L, \rho_2 \rangle_{env} \Rightarrow \langle L \curvearrowright K \rangle_k \, \langle \rho_1, Y \mapsto L, \rho_2 \rangle_{env}$$

Here one can observe that nearly the complete rule is copied on the right hand side.

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:

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

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:

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

Here, V_1 and V_2 are the evaluated results of the expressions E_1 and E_2 respectively. The first two rules are `heating` rules which push any of these operands to top of the stack. The construct $\square$ (HOLE) is a place-holder that will be replaced by the result of the evaluated term or sub-term. Putting the result of the evaluated sub-term back into the term is called `cooling`, depicted by the last two `cooling` rules.

4 Extending Hunt and Sands's Security 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 $\vec{D}$ and $\vec{E}$ 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.

Our work mainly motivated by the security type system proposed in [23], which is primarily proposed to detect possible leakage of sensitive information from programs. Unlike other similar type systems, [23] 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 interprocedural 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 judgments 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 statements, and Γ, Γ': 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 (`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 Γ' 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 judgments $\langle \Gamma \vdash E_1 : \mathtt{T}_1,\ \Gamma \vdash E_2 : \mathtt{T}_2, \ldots \rangle$. Similarly, $\Gamma[\![\vec{id} \mapsto \vec{\mathtt{T}}]\!]$ denotes a sequence of type substitutions $\langle \Gamma[\![id_1 \mapsto \mathtt{T}_1]\!], \Gamma[\![id_2 \mapsto \mathtt{T}_2]\!], \ldots \rangle$. 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

Syntactic Elements
$n \in \mathbb{Z}$: Numerical Values
$id \in Var$: Variables
$E \in \mathbb{E}$: Arithmetic Expressions
$B \in \mathbb{B}$: Boolean Expressions
$C \in \mathbb{C}$: Program Statements

Arithmetic Expressions
$E ::= n$
| id
| $\&id$
| $^*E$
| $id[E]$
| $E\ op\ E$
| (E)
where $op \in \{+, -, \times, /\}$

Boolean Expressions
$B ::= true$
| $false$
| $E\ rel\ E$
| $\neg B$
| $B\ AND\ B$
| $B\ OR\ B$
where $rel \in \{\geqslant, \leqslant, <, >, ==\}$

$\tau ::= int \mid float \mid char \mid bool \mid \tau[n] \mid \tau^*$
$D ::= \tau\ id$
$A ::= id := E \mid {}^*E := E \mid id[E] := E \mid id := read()$
$C ::= skip; \mid D; \mid A; \mid defun\ id(\vec{D})\{C\} \mid call\ id(\vec{E}); \mid return; \mid return\ E; \mid C_1\ C_2 \mid if\ B\ then\ \{C\}$
| $if(B)\ then\ \{C_1\}\ else\ \{C_2\} \mid while(B)\ do\ \{C\}$

Table 2: Abstract Syntax of the Language

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 𝕂 Specification of Security Type System: A Roadmap

This section provides a roadmap to specify 𝕂 rewrite rules corresponding to the typing rules depicted in Figure 2. Let us consider the typing judgment $pc \vdash \Gamma\{C\}\Gamma'$ 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 capture this, let us give algebraic representations of Γ, Γ', C and pc in 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: $\left\langle \langle K \rangle_k \langle \texttt{Map} \rangle_{env} \langle \texttt{Map} \rangle_{context} \right\rangle_T$.

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

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

The symbol "..." appearing in the k cell represents some unspecified computations after the top computation C currently under consideration. 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 $\Gamma \vdash E : \texttt{T}$ specified as $\langle \ldots\ E \mapsto \texttt{T}\ \ldots \rangle_{env}$ means that expression E has the security type $\texttt{T}$ somewhere in the environment *env*. The security type rule asserts the validity of certain judgment on the basis of other judgments that are already known to be valid. Each security type rule is written based on a number of *premise* judgments $\Gamma_i \vdash \zeta_i$ above a horizontal line, with

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

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

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

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

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

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

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

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

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

a single *conclusion* judgment $\Gamma \vdash \zeta$ 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: $\langle \frac{M + N}{M +_{Nat} N} \ \ldots \rangle_k$ $\langle \ldots \ M \mapsto Nat, N \mapsto Nat \ \ldots \rangle_{env}$ 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:

$$\begin{aligned} configuration \equiv \Big\langle \langle K \rangle_k \ \langle \texttt{Map} \rangle_{env} \ \langle \texttt{Map} \rangle_{context} \ \langle \ \langle \texttt{Map} \rangle_{\lambda\text{-}Def} \ \langle \texttt{List} \\ \rangle_{fstack} \ \rangle_{control} \ \langle \texttt{List} \rangle_{in} \ \langle \texttt{List} \rangle_{out} \ \langle \ \langle \texttt{Map} \rangle_{alias} \ \langle \texttt{Set} \rangle_{ptr} \ \rangle_{ptr\text{-}alias} \ \Big\rangle_T \end{aligned}$$

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. These rules, conveniently represented in bi-dimensional form, generalize the conventional rewrite rules by manipulating parts of rewrite terms in different ways: read, write, and don't care. In this form, the left and right hand sides of the rewrite rules are placed above and below respectively of the horizontal line. We label the defined rules by `R_` for future reference. This is worthwhile to mention that our semantics rules fulfill the prime objective to improve the precision of the analysis taking into account 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 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. `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 $\mathbf{(R_{9b})_{approx}}$.

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

$$\textbf{(R}_{\textbf{2}}\textbf{)}_{\textbf{lookup}}: \ \langle \frac{id}{\mathtt{T} : Type} \ \ldots\rangle_k \langle \ldots\ id \mapsto \mathtt{T} : Type \ \ldots\rangle_{env}$$

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

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

$$\textbf{(R}_{\textbf{4}}\textbf{)}_{\textbf{if}}: \ \langle \frac{if(B : \mathtt{T})\ then\ \{C\}}{C \curvearrowright restore_c(\mu) \curvearrowright approx(\rho)} \ \ldots\rangle_k \langle \frac{\mu}{\mu[pc \leftarrow \mu(pc) \sqcup \mathtt{T}]}\rangle_{context} \langle\rho\rangle_{env}$$

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

$$\textbf{(R}_{\textbf{5b}}\textbf{)}_{\textbf{exit-if}}: \ \langle \frac{exitIf()}{save(\rho)} \ \ldots\rangle_k\langle\rho\rangle_{env} \qquad \textbf{(R}_{\textbf{5c}}\textbf{)}_{\textbf{exit-else}}: \langle \frac{exitElse()}{approx(save(\rho))} \ \ldots\rangle_k$$

$$\textbf{(R}_{\textbf{6}}\textbf{)}_{\textbf{while}}:$$
$$\langle \frac{while(B : \mathtt{T})\ do\ \{C\}}{C \curvearrowright restore_c(\mu) \curvearrowright approx(\rho) \curvearrowright fixpoint(B, C, \rho)} \ \ldots\rangle_k \langle\rho\rangle_{env}$$
$$\langle \frac{\mu}{\mu[pc \leftarrow \mu(pc) \sqcup \mathtt{T}]}\rangle_{context}$$

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

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

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

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

$$\textbf{(R}_{\textbf{8}}\textbf{)}_{\textbf{join}}:$$
$$\langle \mathtt{T}_1 : Type\ \sqcup\ \mathtt{T}_2 : Type\ \ldots\rangle_k = \begin{cases} \langle \frac{\mathtt{T}_1 : Type\ \sqcup\ \mathtt{T}_2 : Type}{untaint} \ \ldots\rangle_k, & if\ \mathtt{T}_1 = \mathtt{T}_2 = untaint \\ \langle \frac{\mathtt{T}_1 : Type\ \sqcup\ \mathtt{T}_2 : Type}{taint} \ \ldots\rangle_k, & otherwise \end{cases}$$

$$\textbf{(R}_{\textbf{9a}}\textbf{)}_{\textbf{restore}}: \ \langle \frac{restore_c(\mu)}{.} \ \ldots\rangle_k \langle \frac{\_}{\mu}\rangle_{context} \qquad \textbf{(R}_{\textbf{9b}}\textbf{)}_{\textbf{approx}}: \ \langle \frac{approx(\rho)}{.} \ \ldots\rangle_k \langle \frac{\rho_c}{\rho \sqcup \rho_c}\rangle_{env}$$

Fig. 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. In order to restore the current program context on returning from a called function, the rule enforces to save current program context which includes the current environment ρ, the remaining tasks K and the control Ctr into $\langle\ \rangle_{fstack}$ cell. 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. This returns the security type of the resultant expression and restores the previous context to start execution of the 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 [4] 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.

Apart from this, capturing the semantics of constant functions has a significant impact on the precision of taint analysis. As outputs of constant functions are always independent of their inputs, any taint effect propagated from untrusted sources will get nullified by them. 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. Although our attempt solves this problem partially, however to obtain a complete solution a rich analysis in $\mathbb{K}$ is required, possibly by combining with it a copy or constant propagation analysis in the concrete domain considering additional cells in the configuration. We consider this scope of improvement as our future challenge. 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 $y := read()$; $x := y$; $v := x$ *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. An SQL injection attack is one such example, where injection of a malicious input as a control part may severely compromise the security.

We end this section stating the fundamental results on K-Taint.

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.*

Proof. We prove this theorem in the line of Abstract Interpretation, designing security type systems as a sound semantics approximation [41]. Let $\mathbb{L}$ be the set of available memory locations. Given a set of n security classes $\mathbb{S} = \{s_1, \ldots, s_n\}$,

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

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

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

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

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

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

Fig. 4: Taint semantics rules for pointer aliasing and constant functions

each security class $s_i \in \mathbb{S}$ corresponds to a set of locations $L_i \subseteq \mathbb{L}$ with accessibility level respecting the sensitivity level intended for s_i. Considering $\mathbb{S}$ as variables abstract value domain representing security properties, the following Galois Connection is established: $((\wp(\mathbb{L}), \leqslant), \alpha_v, \gamma_v, (\mathbb{S}, \sqsubseteq))$ where $\leqslant$ and $\sqsubseteq$ denote memory accessibility level and value security level relations respectively. The concretization function $\gamma_v : \mathbb{S} \mapsto \wp(\mathbb{L})$ is defined as:

$$\gamma_v(s_i) = L_i = \{l \mid l \ has\ access\ level\ s_i\}$$

where concretizations of different security classes are disjoint.

Now moving towards expressions, we can similarly define the Galois connection $((\wp(\mathbb{L}), \leqslant), \alpha_e, \gamma_e, (\mathbb{S}, \sqsubseteq))$ where γ_e is defined by the upper approximation of the locations applying γ_v on the security levels of variables in the expressions.

Given an environment $\sigma \in \Sigma = Var \mapsto Val$ and a type environment $\rho \in \Psi = Var \mapsto \mathbb{S}$, let us define the following mapping where $loc : Var \mapsto \mathbb{L}$:

$$\gamma(\rho) = \{\sigma \mid \forall x \in Var.\ \sigma(x) \neq \bot \implies loc(x) \in \gamma_v(\rho(x))\}$$

A semantic property refers to a set of semantics. The standard collecting semantics is the function that maps every object in its stronger semantic property, precisely in the set having as unique element the standard semantics of that object [41]. Given the expression semantic domain and the program semantic domain $\mathbb{R}^e : \Sigma \mapsto Val$ and $\mathbb{R} : \Sigma \mapsto \Sigma$ respectively, the standard collecting semantic property for them are defined as $\mathbb{CS}^e = \wp(\mathbb{R}^e)$ and $\mathbb{CS} = \wp(\mathbb{R})$. The type systems are considered as abstract compositional semantics approximating the standard collecting semantics of imperative languages, and is formally proved by building a Galois-connection between security types and semantic properties.

The Galois connection that approximates semantic properties of expressions, $\mathbb{CS}^e$, with security typings for expressions, $\mathbb{T}^e : \Psi \mapsto \mathbb{S}$, is defined as $(L_c, \alpha_t, \gamma_t, L_a)$ where $L_c = (\mathbb{CS}^e, \subseteq_c, \emptyset, \mathbb{R}^e, \cap_c, \cup_c)$ and $L_a = (\mathbb{T}^e, \sqsubseteq_a, \bot_a, \top_a, \sqcap_a, \sqcup_a)$ and $\alpha_t : \mathbb{CS}^e \mapsto \mathbb{T}^e$ defined as

$$\mathcal{P} \mapsto \lambda\rho \in \Psi.\ \alpha_e(\{loc(x) \mid \exists \mathsf{p} \in \mathcal{P} : \forall\sigma \in \gamma(\rho).\sigma(x) \neq \bot \implies \mathsf{p}\sigma \neq \bot\})$$

Given $\mathcal{E}^e[\![.]\!] : \mathbb{E} \mapsto \mathbb{CS}^e$ and $\mathcal{T}^e[\![.]\!] : \mathbb{E} \mapsto \mathbb{T}^e$, the security typing of expressions is sound, i.e. for every expression e: $\alpha_t(\mathcal{E}^e[\![e]\!]) \sqsubseteq_a \mathcal{T}^e[\![e]\!]$, as the following semantics are defined in the rules $\mathbf{(R_{1a})_{decl}}$, $\mathbf{(R_{1b})_{read}}$, and $\mathbf{(R_{3a})_{ar\text{-}op}}$:

$$\begin{aligned} n &\mapsto \lambda\ \rho \in \Psi.\ untaint \\ id &\mapsto \lambda\ \rho \in \Psi.\ \rho(id) \\ read() &\mapsto \lambda\ \rho \in \Psi.\ taint \\ E_1\ op\ E_2 &\mapsto \lambda\ \rho \in \Psi.\ \mathcal{T}^e[\![E_1]\!] \sqcup \mathcal{T}^e[\![E_2]\!] \end{aligned}$$

Similarly, the soundness of programs semantic properties, i.e. for every command c: $\alpha_t(C[\![c]\!]) \sqsubseteq_a \mathcal{T}[\![c]\!]$ where $\mathbb{T} : \Psi \mapsto \Psi, \mathcal{T}[\![.]\!] : \mathbb{C} \mapsto \mathbb{T}$ and $C[\![.]\!] : \mathbb{C} \mapsto \mathbb{CS}$, can be proved based on the use of *approx*() (rule $\mathbf{(R_{9b})_{approx}}$) in the rules $\mathbf{(R_{3b})_{asg}}$, $\mathbf{(R_4)_{if}}$, $\mathbf{(R_{5a})_{if\text{-}else}}$, $\mathbf{(R_6)_{while}}$ and $\mathbf{(R_7)_{fun}}$.

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

Proof. Let h be the height of the security lattice built over the finite set $\mathbb{S}$ of n security classes. Let X be the set of variables in program P, where $|X| = m$. The set of type environment for P is defined as $\Psi = X \mapsto \mathbb{S}$. Considering the ordering $\sqsubseteq$, join $\sqcup$ and meet $\sqcap$ defined in $\mathbb{S}$ component-wise, we can build a lattice over the type environments as $L_t = (\Psi, \sqsubseteq_t, \bot_t, \top_t, \cap_t, \cup_t)$ whose height is mh. Consider the auxiliary function *fixpoint*() which follow either (1) $\langle \frac{fixpoint(B, C, \rho_i)}{\cdot} \ldots \rangle_k \langle \rho_i' \rangle_{env}$ 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 $\rho_i \neq \rho_i'$. As the join operation $\cup_t$ in L_t is monotonic (hence the function *approx*()) and the height of L_t is finite, therefore the function *fixpoint*(), in presence of iterations, takes in the worst case mh number of iterations to reach fix-point solution $\top_t$. This establishes a guarantee for any execution to always converge after a finite number of iterations.

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 $\langle\rho_1'\rangle_{env} \approx_{s_t} \langle\rho_2'\rangle_{env}$.*

Proof. The proof is established by induction considering all different base cases on assignment, conditional, iteration, etc., and then by induction step on execution paths, as follows: *(1) Expressions:* Given an expression E. Let, as per the rules $\mathbf{(R_{1b})_{read}}$, $\mathbf{(R_2)_{lookup}}$, and $\mathbf{(R_{8b})_{ar\text{-}op}}$, we get $\langle \frac{E}{s_1} \dots\rangle_k\langle\rho_1\rangle_{env}$ and $\langle \frac{E}{s_2} \dots\rangle_k\langle\rho_2\rangle_{env}$. As $\langle\rho_1\rangle_{env} \approx_{s_t} \langle\rho_2\rangle_{env}$, by following the set theoretic law on the least upper bound of each variable's security level s_v in E satisfying $s_v \sqsubseteq s_t$, we get $s_1 \sqsubseteq s_t \wedge s_2 \sqsubseteq s_t$. *(2) Assignment:* Given an assignment $x := E$, after execution of the assignment statement on ρ_1 and ρ_2, we have $\langle \dots \rho_1[x \mapsto \mu(pc) \sqcup s_1] \dots\rangle$ and $\langle \dots \rho_2[x \mapsto \mu(pc) \sqcup s_2] \dots\rangle$ respectively. Assuming $\mu(pc) \sqsubseteq s_t$ and as $s_1, s_2 \sqsubseteq s_t$, we get $(\mu(pc) \sqcup s_1) \sqsubseteq s_t$ and $(\mu(pc) \sqcup s_2) \sqsubseteq s_t$. Therefore, $\rho_1[x \mapsto \mu(pc) \sqcup s_1] \approx_{s_t} \rho_2[x \mapsto \mu(pc) \sqcup s_1]$. *(3) Conditional:* In case of conditional statement "$if(B)$ *then* $\{C_1\}$ *else* $\{C_2\}$", application of $approx(save(\rho))$ in rule $\mathbf{(R_{5c})_{exit\text{-}else}}$ performs the join of two type-environments obtained by executing two branches. Let $\langle \frac{C_1}{\cdot} \dots\rangle_k\langle\frac{\rho_1}{\rho'}\rangle_{env}$ and $\langle \frac{C_2}{\cdot} \dots\rangle_k\langle\frac{\rho_1}{\rho''}\rangle_{env}$. As $\forall x \in X.\ \rho'(x) \sqsubseteq s_t \wedge \rho''(x) \sqsubseteq s_t$, then $\rho'(x) \sqcup \rho''(x) = \rho_1'(x) \sqsubseteq s_t$. This is also true with respect to ρ_2, i.e. $\rho_2'(x) \sqsubseteq s_t$ as well. Therefore, given $\langle\rho_1\rangle_{env} \approx_{s_t} \langle\rho_2\rangle_{env}$, we get $\forall x \in X.\ \rho_1'(x) \sqsubseteq s_t \wedge \rho_2'(x) \sqsubseteq s_t$ which implies $\langle\rho_1'\rangle_{env} \approx_{s_t} \langle\rho_2'\rangle_{env}$. *(4) Iteration:* In case of while statement "$while(B)$ *do* $\{C\}$", the $approx(\rho)$ function until least fix point solution performs join operation $\sqcup$ on the type-environments after each iteration. Therefore, this can also be proved similarly that, starting with ρ_1 and ρ_2, after each iteration we get $\langle\rho_1^i\rangle_{env} \approx_{s_t} \langle\rho_2^i\rangle_{env}$ and on reaching fix-point solution $\langle\rho_1'\rangle_{env} \approx_{s_t} \langle\rho_2'\rangle_{env}$. *(5) Sequence:* Finally, in case of $C_1\ C_2$, given $\langle \frac{C_1}{\cdot} \dots\rangle_k\langle\frac{\rho_1}{\rho'}\rangle_{env}$, $\langle \frac{C_1}{\cdot} \dots\rangle_k\langle\frac{\rho_2}{\rho''}\rangle_{env}$, $\langle \frac{C_2}{\cdot} \dots\rangle_k\langle\frac{\rho'}{\rho_1'}\rangle_{env}$, and $\langle \frac{C_2}{\cdot} \dots\rangle_k\langle\frac{\rho''}{\rho_2'}\rangle_{env}$, we get $\langle \frac{C_1\ C_2}{\cdot} \dots\rangle_k\langle\frac{\rho_1}{\rho_1'}\rangle_{env}$ and $\langle \frac{C_1\ C_2}{\cdot} \dots\rangle_k\langle\frac{\rho_2}{\rho_2'}\rangle_{env}$. Using the above base cases, we can prove $\langle\rho_1'\rangle_{env} \approx_{s_t} \langle\rho_2'\rangle_{env}$.

Progs.	Descriptions	K-Taint	Splint [15]	Pixy [24]	SFlow [22]	CQual [18]
Prog1	Explicit Flow [1]	✓	✓	✓	✓	✓
Prog2	Implicit Flow [1]	✓	$✗_-$	$✗_-$	$✗_-$	$✗_-$
Prog3	Malware Attack [8]	✓	$✗_-$	$✗_-$	$✗_-$	$✗_-$
Prog4	XSS Attack [39]	✓	$✗_-$	$✗_-$	$✗_-$	$✗_-$
Prog5	Buffer Overflow [16]	$✗_+$	✓	✓	$✗_+$,$✗_-$	$✗_-$
Prog6	Presence of 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 [34]	✓	$✗_-$	✓	$✗_-$	$✗_-$
Prog13	Swapping of two numbers using pointers	✓	✓	✓	✓	$✗_-$

Table 3: Taint Analysis on Benchmark Programs Set [1, 8, 16, 34, 39] (✓: `Passed`, $✗_+$: `False Positives`, $✗_-$: `False negatives`)

8 Experimental Analysis

We have implemented the full set of semantics rules (more than 200 rules) in the $\mathbb{K}$ tool (version 4.0)[3] for our imperative language under consideration and performed experiments on a set of benchmark codes collected from [1, 8, 16, 34, 39] and on some well-known programs [4]. 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` [15], `Pixy` [24], `SFlow` [22], and `CQual` [18], 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 [8], [34] and [39] highlighted some special cases where their approaches fail. We consider those special cases (shown as `Prog3`, `Prog4` and `Prog12` in Table 3), and we show in Table 4 how K-Taint successfully captures those taint flows by showing their corresponding execution steps.

[3] Online $\mathbb{K}$ tool is available at *http://www.kframework.org/tool/run/*.

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

Prog. Pts	Prog Stmts	Security Types	Rules
1	`int` w,x,y,z;	w|->U x|->U y|->U z|->U	$(R_{1a})_{\texttt{Decl}}$
2	y = `read();`	w|->U x|->U y|->T z|->U	$(R_{1b})_{\texttt{read}}$,$(R_{3b})_{\texttt{asg}}$
3	x=0; z = 0;	w|->U x|->Z y|->T z|->Z	$(R_{3b})_{\texttt{asg}}$
4	`if`(y == 1){	w|->U x|->Z y|->T z|->Z	$(R_{4c})_{\texttt{if-else}}$
5	x = 1;}	w|->U x|->T y|->T z|->Z	$(R_{3b})_{\texttt{asg}}$
6	`else`{	w|->U x|->T y|->T z|->Z	$(R_{4d})_{\texttt{if-else}}$
7	z = 1;}	w|->U x|->T y|->T z|->T	$(R_{3b})_{\texttt{asg}}$
8	`if`(x==0){	w|->U x|->T y|->T z|->T	$(R_{4a})_{\texttt{if-else}}$
9	w = 0; }	w|->T x|->T y|->T z|->T	$(R_{3b})_{\texttt{asg}}$
10	if(z==0) {	w|->T x|->T y|->T z|->T	$(R_{4a})_{\texttt{if-else}}$
11	w = 1; }	w|->T x|->T y|->T z|->T	$(R_{3b})_{\texttt{asg}}$
`Output by K-Taint`		w|->T x|->T y|->T z|->T	

(a) Execution Steps of Prog3 [8]

Prog. Pts	Prog Stmts	Security Types	Rules
1	`int` x,y,z,a;	x|->U y|->U z|->U a|->U	$(R_{1a})_{\texttt{Decl}}$
2	x = 0;y = 0;	x|->Z y|->Z z|->U a|->U	$(R_{3b})_{\texttt{asg}}$
3	z = `read();`	x|->Z y|->Z z|->T a|->U	$(R_{3b})_{\texttt{asg}}$
4	`if` ($z \leq a$){	x|->Z y|->Z z|->T a|->U	$(R_{4c})_{\texttt{if-else}}$
5	x=1;}	x|->T y|->Z z|->T a|->U	$(R_{3b})_{\texttt{asg}}$
6	`else`{	x|->T y|->Z z|->T a|->U	$(R_{4d})_{\texttt{if-else}}$
7	y=1;}	x|->T y|->T z|->T a|->U	$(R_{3b})_{\texttt{asg}}$
8	`if`(x==0){	x|->T y|->T z|->T a|->U	$(R_4)_{\texttt{if}}$
9	x = a;}	x|->T y|->T z|->T a|->U	$(R_{3b})_{\texttt{asg}}$
10	if(y==0){	x|->T y|->T z|->T a|->U	$(R_4)_{\texttt{if}}$
11	y = a;}	x|->T y|->T z|->T a|->U	$(R_{3b})_{\texttt{asg}}$
`Output by K-Taint`		x|->T y|->T z|->T a|->U	

(b) Execution Steps of Prog4 [39]

Prog. Pts	Prog Stmts	Security Types	Rules
1	`int` *pub*,*temp*,*secret*,*input*;	*input*|->U *pub*|->U *temp*|->U *secret*|->U	$(R_{1a})_{\texttt{Decl}}$
2	*input* = `read();`	*input*|->T *pub*|->U *temp*|->U *secret*|->U	$(R_{1b})_{\texttt{read}}$, $(R_{3b})_{\texttt{asg}}$
3	*pub*=1;	*input*|->T *pub*|->U *temp*|->U *secret*|->U	$(R_{3b})_{\texttt{asg}}$
4	*temp*=0;	*input*|->T *pub*|->U *temp*|->Z *secret*|->U	$(R_{3b})_{\texttt{asg}}$
5	`if`(*secret*≤*input*){	*input*|->T *pub*|->U *temp*|->Z *secret*|->U	$(R_4)_{\texttt{if}}$
6	*temp*=1;}	*input*|->T *pub*|->U *temp*|->T *secret*|->U	$(R_{3b})_{\texttt{asg}}$
7	`if`(*temp*≤0){	*input*|->T *pub*|->U *temp*|->T *secret*|->U	$(R_4)_{\texttt{if}}$
8	*pub*=0;}	*input*|->T *pub*|->T *temp*|->T *secret*|->U	$(R_{3b})_{\texttt{asg}}$
	`Output by K-Taint`	*input*|->T *pub*|->T *temp*|->T *secret*|->U	

(c) Execution Steps of Prog12 [34]

Table 4: Detail Execution Steps in K-Taint (`U: untaint`, `T: taint`, `Z: zero`)

Observe that, at program points 5 and 7 in tables 4(a), K-Taint makes the variables x and z tainted, which causes the variable w to become tainted at program points 9 and 11, thus reducing false negatives. Similarly in table 4(b), at program points 5 and 7, K-Taint makes the variables x and y tainted preventing them to disclose private information to attackers. In table 4(c) at program point 6, K-Taint makes the variable `temp` tainted which causes the variable `pub` to become tainted at program point 8, reducing false negatives. The last row of every table is the result of the analysis in K-Taint representing the final security types of program variables.

9 Conclusions

This paper proposed 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 endeavour 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.

References

1. Stanford securibench micro., `http://suif.stanford.edu/~livshits/work/securibench-micro/index.html`, [Online; accessed 29-April-2016]
2. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: SAS. vol. 3148, pp. 100–115. Springer (2004)
3. Arzt, S., et al.: Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. ACM Sigplan Notices 49(6), 259–269 (2014)
4. Asăvoae, I.M.: Abstract semantics for alias analysis in k. Electronic Notes in Theoretical Computer Science 304, 97–110 (2014)
5. Asavoae, I.M., Asavoae, M.: Collecting semantics under predicate abstraction in the k framework. In: WRLA. vol. 6381, pp. 123–139. Springer (2010)
6. Asavoae, I.M., Asavoae, M., Lucanu, D.: Path directed symbolic execution in the k framework. In: 12th Int. Symp. on SYNASC. pp. 133–141. IEEE (2010)
7. Bogdanas, D., Roşu, G.: K-java: a complete semantics of java. ACM SIGPLAN Notices 50(1), 445–456 (2015)
8. Cavallaro, L., Saxena, P., Sekar, R.: 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. pp. 143–163. Springer (2008)
9. Cifuentes, C., Scholz, B.: Parfait: designing a scalable bug checker. In: Proc. of the 2008 workshop on Static analysis. pp. 4–11. ACM (2008)
10. Clavel, M., et al.: All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic. Springer-Verlag (2007)

11. Corin, R., Manzano, F.A.: Taint analysis of security code in the klee symbolic execution engine. In: International Conference on Information and Communications Security. pp. 264–275. Springer (2012)
12. Cousot, P.: Types as abstract interpretations. In: Proc. of the 24th ACM SIGPLAN-SIGACT symposium on POPL. pp. 316–331. ACM (1997)
13. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
14. Ellison, C., Rosu, G.: An executable formal semantics of c with applications. In: ACM SIGPLAN Notices. vol. 47, pp. 533–544. ACM (2012)
15. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE software 19(1), 42–51 (2002)
16. Evans, D., Larochelle, D., Evans, D.: Splint manual: Version 3.1.1-1, 2003, `http://www.splint.org/manual`, [Online; accessed 15-June-2016]
17. Filaretti, D., Maffeis, S.: An executable formal semantics of php. In: ECOOP. pp. 567–592. Springer (2014)
18. Foster, J.S., et al.: Cqual user's guide. University of California, Berkeley, version 0.9 edition (2002)
19. Fritz, C., Arzt, S., Rasthofer, S., Bodden, E., Bartel, A., Klein, J., L. Traon, Y., Octeau, D., McDaniel, P.: Highly precise taint analysis for android applications. EC SPRIDE, TU Darmstadt, Tech. Rep (2013)
20. Guth, D.: A formal semantics of python 3.3 (2013), `https://www.ideals.illinois.edu/handle/2142/45275`
21. Hammer, C., Snelting, G.: Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. Int. Journal of Information Security 8(6), 399–422 (2009)
22. Huang, W., Dong, Y., Milanova, A.: Type-based taint analysis for java web applications. In: In Proc. of Int. Conf. on Fundamental Approaches to Software Engineering. pp. 140–154. Springer (2014)
23. Hunt, S., Sands, D.: On flow-sensitive security types. In: Conf. Record of the 33rd ACM SIGPLAN-SIGACT Sym. on POPL. pp. 79–90. ACM, S. California (2006)
24. Jovanovic, N., Kruegel, C., Kirda, E.: Pixy: A static analysis tool for detecting web application vulnerabilities. In: IEEE Symposium on Security and Privacy (S&P'06). pp. pp. 258–263. IEEE. IEEE (2006)
25. Khoo, W.M.: Taintgrind: A valgrind taint analysis tool, `https://github.com/wmkhoo/taintgrind`, [Online; accessed 18-May-2016]
26. Li, L., Bartel, A., Klein, J., Traon, Y.L., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., Mcdaniel, P.: I know what leaked in your pocket: uncovering privacy leaks on android apps with static taint analysis. arXiv preprint arXiv:1404.7431 (2014)
27. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX Security Symposium. vol. 14, pp. 18–18 (2005)
28. Mantel, H., Sudbrock, H.: Types vs. pdgs in information flow analysis. In: LOPSTR. vol. 7844, pp. 106–121 (2012)
29. Meredith, P., Rosu, M.H.G.: An executable rewriting logic semantics of k-scheme. In: Workshop on Scheme and Functional Programming. vol. 1, p. 10 (2007)
30. Meseguer, J., Roşu, G.: The rewriting logic semantics project. Theoretical Computer Science 373(3), 213–237 (2007)
31. Noundou, X.N.: Saint taint analyzer, `https://github.com/xaviernoumbis/saint`, [Online; accessed 15-August-2016]
32. Pistoia, M., Banerjee, A., Naumann, D.A.: Beyond stack inspection: A unified access-control and information-flow security model. In: Proc. of the IEEE Symposium on Security and Privacy. pp. 149–163. IEEE, USA (2007)

33. Roşu, G., Şerbănuţă, T.F.: An overview of the k semantic framework. The Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
34. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: 23rd IEEE Computer Security Foundations Symposium. pp. 186–199. IEEE (2010)
35. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications 21(1), 5–19 (2006)
36. Scholz, B., Zhang, C., Cifuentes, C.: User-input dependence analysis via graph reachability. Tech. Rep. SMLI TR-2008-171, Mountain View, CA, USA (2008)
37. Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., Berg, R.: F4f: taint analysis of framework-based web applications. ACM SIGPLAN Notices 46(10), 1053–1068 (2011)
38. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. In: ACM Sigplan Notices. vol. 44, pp. 87–97. ACM (2009)
39. Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross site scripting prevention with dynamic data tainting and static analysis. In: NDSS. vol. 2007, p. 12 (2007)
40. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. 4(2-3), 167–187 (1996)
41. Zanotti, M.: Security typings by abstract interpretation. Static Analysis pp. 267–297 (2002)