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

Md. Imran Alam
Raju Halder
Harshita Goswami
Jorge Sousa Pinto

1Indian Institute of Technology Patna, India
2HASLab/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.

```
1. void foo(char *src)
2. char buf[20]; int i=0;
3. while(i<=strlen(src)) {
4.    buf[i] = src[i]; i++;
5. 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}
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 K framework, considering an extension of Hunt and Sands’s semantically sound flow-sensitive security type system as the basis. The K framework (Roșu and Șerbănuta, 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 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 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 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 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 adaptation without actually flowing the context information through the called function body, which may often result false alarms. Like CQual, the SFlow also forgets 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 K FRAMEWORK

The K framework provides a rewrite logic-based framework suitable for design and analysis of programming languages. Inspired by rewrite-logic semantics project (Meseguer and Roșu, 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 K framework mainly relies on configuration and 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).

<table>
<thead>
<tr>
<th>Language Supported</th>
<th>Imperative (including C-like syntax)</th>
<th>PHP</th>
<th>C</th>
<th>Java</th>
<th>C</th>
<th>C</th>
<th>Java</th>
<th>C</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported</td>
<td>K-Taint</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Flow-Sensitivity</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Constant Functions</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Implicit Flow</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Type System</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Semantics/Security</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>K-Taint</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**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 $(\tau id_1, \tau id_2, \ldots)$ and a sequence of arithmetic expressions $(E_1, E_2, \ldots)$ respectively.

**Table 2: Abstract Syntax of the Language.**

- $E ::= n | id | &id | (E) | if(B) \{E\} | \{if(B) then E else E\} | while(B) \{E\} | return E | return E | skip | D | A | \text{defun} id(D) | call id(E_1) | return | return | C_1 C_2 | if(B) \{C_1\} | \{if(B) then \{C_1\} else \{C_2\}\} | while(B) \{C\} |$

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: $\text{taint}$ and $\text{untaint}$. We will work with the flow semi-join lattice of the type domain as $SD = (S, \subseteq, \sqcup)$, where $S = \{\text{taint}, \text{untaint}\}$ and the partial order relation defined as $\text{untaint} \subseteq \text{taint}$.

The typing judgements are of the form $pc \vdash \Gamma \vdash \{C\} \Gamma'$, where $pc \in S$ represents the security context used to address implicit flow, $C$ is the program...
The typing rules ensure that for any given $C$, $\Gamma$, and $pc$ there is an environment $\Gamma'$ such that $pc \vdash \Gamma \{ C \} \Gamma'$ is derivable. We use the notation $\Gamma \vdash E : \tau$ to denote the sequence of type judgements $(\Gamma \vdash E_1 : \tau_1, \Gamma \vdash E_2 : \tau_2, \ldots)$. Similarly, $[\Gamma \vdash id \vdash \tau]$ denotes a sequence of type substitutions $(\Gamma \vdash \text{id}_1 \vdash \tau_1, \Gamma \vdash \text{id}_2 \vdash \tau_2, \ldots)$. 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.

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

5 \textbf{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 judgement $pc \vdash \Gamma \{ C \} \Gamma'$ which specifies that the security environment $\Gamma'$ is derived by executing the statement $C$ on the security environment $\Gamma$ under the program’s security context $pc$. To capture this, let us give algebraic representations of $\Gamma$, $\Gamma'$, $C$ 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: $\langle k | \langle \text{map} \rangle | \langle \text{env} \rangle | \langle \text{context} \rangle \rangle$.

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

$\langle \ldots | \langle \Gamma \rangle | \langle \text{env} \rangle | \langle \text{context} \rangle | \rangle$

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 $\Gamma$ in the env cell will be updated by the modified environment $\Gamma'$ (implicit) influenced by the current value (denoted by $_{\text{env}}$) of the security context $pc$ in the context cell.

Similarly, the derivation rule $pc \vdash E : \tau$ specified as $\langle \ldots E \vdash \tau \ldots \rangle_{\text{env}}$ means that expression $E$ has the security type $\tau$ somewhere in the environment $\text{env}$. Each security type rule is written based on a number of premise judgements $\Gamma_i \vdash \tau_i$ above a horizontal line, with a single conclusion judgement $\Gamma \vdash \tau$ below the line. For example, given the type rule $\Gamma \vdash M : \text{Nat} \quad \Gamma \vdash N : \text{Nat}$, the corresponding K rule is defined as: $\langle M | N | \langle \ldots \rangle_{\text{env}} \rangle_{\text{context}}$ where $M : \text{Nat}$, $N : \text{Nat}$, and $\langle \ldots \rangle_{\text{env}} : \langle \ldots \rangle_{\text{context}}$.

6 \textbf{K REWRITING LOGIC SEMANTICS FOR TAIN ANALYSIS}

This section introduces an executable rewriting logic semantics in the 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 K modeling of the program configuration on which the semantics is defined:

$\text{configuration} \equiv \langle \langle \text{env} \rangle \rangle_{\text{context}} \langle \langle \text{map} \rangle \rangle_{\text{context}} \langle \langle \text{env} \rangle \rangle_{\text{context}} \langle \langle \text{context} \rangle \rangle_{\text{context}}$

As mentioned earlier, the special cell $\langle k \rangle_{k}$ contains the list of computation tasks of a special sort $K$ separated by the associative sequentialization operator $\wedge$. 

362
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 \( \lambda\)-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 \( \mathcal{K} \) definitional framework. We label the defined rules by \( \S \) for future reference. These rules capture 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 \((R_1)_{\text{decl}}\) deals with variables declarations and initialization of variables by their initial security types (\( untaint \) in our case) in the environment cell \( env \). Any unanitized input gets its type tainted in the rule \((R_\text{th})_{\text{read}}\). The lookup rule \((R_2)_{\text{lookup}}\) replaces the variable term appearing on top of \( k \) cell by its security type by looking into the environment cell. Note that the lookup rule for constant terms, although we do not mention here, always returns \( untaint \). As defined in rule \((R_3)_{\text{up-op}}\), the security types of expressions are sound approximated by least upper bound (defined in rule \((R_4)_{\text{join}}\)) of their component terms security types. Rule \((R_\text{th})_{\text{bag}}\) 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. \( \top \)) 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 \( \mu \) in the \( context \) cell based on the security type of \( B \) and the later is achieved by following \( restore, (\mu) \) and \( approx(p) \). These are depicted in rules \((R_4)_{\text{if}}, (R_6)_{\text{while}}, (R_9)_{\text{restore}}\) and \((R_9)_{\text{approx}}\).

\[
\begin{align*}
(R_{1a})_{\text{decl}}: & \quad \begin{cases}
\tau \text{id} & : \text{Type} \quad \mu
\end{cases} \quad \rho \text{id} \leftarrow \tau \quad \text{Type} \quad \mu
\text{env} \\
(R_{2})_{\text{lookup}}: & \quad \begin{cases}
\tau \text{id} & : \text{Type} \quad \mu
\end{cases} \quad \rho \text{id} \leftarrow \tau \quad \text{Type} \quad \mu
\text{env} \\
(R_{3a})_{\text{up-op}}: & \quad \begin{cases}
\text{id} & : \text{Type} \qquad \mu \\
\text{id} & : \text{Type} \oplus \text{Type} \quad \mu \\
\text{id} & : \text{Type} \quad \mu
\end{cases} \quad \rho \text{id} \leftarrow \text{id} \quad \text{Type} \quad \mu
\text{env} \\
(R_{4})_{\text{bag}}: & \quad \begin{cases}
\text{id} & : \text{Type} \quad \mu \\
\text{id} & : \text{Type} \quad \mu \\
\text{id} & : \text{Type} \quad \mu
\end{cases} \quad \rho \text{id} \leftarrow \text{id} \quad \text{Type} \quad \mu
\text{env} \\
(R_{4})_{\text{lookup}}: & \quad \begin{cases}
\text{id} & : \text{Type} \quad \mu \\
\text{id} & : \text{Type} \quad \mu
\end{cases} \quad \rho \text{id} \leftarrow \text{id} \quad \text{Type} \quad \mu
\text{env} \\
(R_{4})_{\text{join}}: & \quad \begin{cases}
\text{id} & : \text{Type} \quad \mu \\
\text{id} & : \text{Type} \quad \mu
\end{cases} \quad \mu \leftarrow \mu \\
(R_{4})_{\text{fixpoint}}: & \quad \begin{cases}
\text{id} & : \text{Type} \quad \mu
\end{cases} \quad \mu \leftarrow \mu \\
(R_{4})_{\text{restore}}: & \quad \begin{cases}
\text{id} & : \text{Type} \quad \mu
\end{cases} \quad \rho \text{id} \leftarrow \text{id} \quad \text{Type} \quad \mu
\text{env} \\
(R_{4})_{\text{approx}}: & \quad \begin{cases}
\text{id} & : \text{Type} \quad \mu
\end{cases} \quad \rho \text{id} \leftarrow \text{id} \quad \text{Type} \quad \mu
\text{env}
\end{align*}
\]

Figure 3: \( \mathcal{K} \) rewrite rules for executable semantics-based taint analysis.

Specifically, \( restore, (\mu) \) restores the previous context on exiting a block guarded by \( B \) and \( approx(p) \) 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 \( \text{while}\) is achieved by defining an auxiliary function \( \text{fixpoint}() \) as follows: either (1) \( \text{fixpoint}(B, C, p_1) \rightarrow \langle \mu \rangle_{\text{un}} \) when \( p_1 = p' \), or (2) \( \text{fixpoint}(B, C, p) \rightarrow \langle \mu \rangle_{\text{un}} \) when \( p_1 \neq p'_1 \). 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 \((R_{5a})_{test-if}\), \((R_{5b})_{exit-if}\) and \((R_{5c})_{exit-else}\). Observe that both the branches are executed over the same environment (using \( \text{restore}_\text{env}(p) \)) which restores environment and is defined similar to the rule \((R_9)_{restore}\) which occurs at the entry point of if-else.

### Dealing with Functions:
We specify the rules \((R_7a)_{fun-decl}\), \((R_7b)_{fun-call}\), \((R_7c)_{fun-call}\) and \((R_7d)_{fun-ret}\) to handle interprocedural feature in our analysis. For each function definition, the rule \((R_7a)_{fun-decl}\) creates a lambda abstraction binding it to the function name in the \((\lambda)_{\text{Def}}\) cell. Coming across a function call, the rule \((R_7b)_{fun-call}\) replaces this function call by its lambda abstraction. We use a helper function \( \text{McDecl}\)() 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:

\[
\begin{align*}
\text{McDecl}(\text{param}, \text{params}, (\text{Type}, \text{Types})) & : \text{fun-call}(	ext{fun-decl})
\end{align*}
\]

Note that the function \( \text{McDecl}\)() enforces the context sensitivity by treating same function call with different parameters differently. As usual, \( \text{McDecl}\)() 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 \((R_7d)_{fun-ret}\) 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 \( [\text{ListTerm}(p,K,Cry) . . . ]_{\text{func}} \).

### 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 \((R_{10a})_{alias}\) 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 \((R_{10b})_{alias}\) 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 tainted value must be tainted, leading to a sound analysis. Similarly, rules \((R_{10c})_{alias}\), \((R_{10d})_{alias}\), and \((R_{10e})_{alias}\) 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 (Asävão, 2014) integrated the alias analysis in \( K \) 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.

![Figure 4: K rules for pointer aliasing and constant functions](image)

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 \( x - x, x \lor x, x \times 0 \), etc. We mention one of such rules in \((R_{11})_{con-func}\).

In this context, as a notable observation, we consider the following scenario: given the code fragment...
8 EXPERIMENTAL ANALYSIS

We have implemented the full set of semantics rules (more than 200 rules) in the 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) (relevant results 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 ‘✓’ indicate failures due to false positives and false negatives, whereas ‘✓’ indicates a successful detection of taint vulnerabilities. Observation 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 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 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


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


