IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Volume-4, Issue-11 E-ISSN: 2347-2693

Evaluating the Overhead of Dynamic Information Flow Analysis Performed by Security Typed Programming Languages

Doaa Hassan

Computers and Systems Department, National Telecommunication Institute, Cairo, Egypt
e-mail: doaa@nti.sci.eg

Available online at: www.ijcseonline.org

Received: Oct/14/2016 Revised: Oct/26/2016 Accepted: Nov/20/2016 Published: Nov/30/2016

***Abstract*— Security-typed programming languages aim to track insecure information flows in application program. This is achieved by extending data types with security labels in order to identify the confidentiality and integrity policies for each data element. Such policies specify which principals or entities are allowed to read from or write to the value of data respectively. In this paper, we evaluate the run-time overhead of dynamic information flow (DIF) analysis in security typed programming languages. Such analysis is performed by including the security labeling in the dynamic operational semantics. Our evaluation mechanism relies on developing two different language implementations for a simple while programming language that has been considered as a case of study. The first one is a traditional interpreter that implements the ordinary operational semantics of the language without security labeling of data types and hence performs no information flow analysis. The second one is an interpreter that performs a dynamic information flow analysis by implementing the security labeling semantics (where language data types are augmented with security labels). Next, two execution times of a program executed using both interpreters are measured (i.e., one execution time for each interpreter). The resulting difference in execution time represents the absolute run-time overhead of dynamic information flow analysis. We have calculated the difference in execution time for some benchmark programs that are executed using both implementations.**

Keywords- Dynamic information flow; security labeling; run-time overhead; operational semantics

I. INTRODUCTION

Information flow control aims to protecting the confidentiality when dealing with sensitive information and dually the integrity when dealing with trustworthy information. Security typed programming languages have been a promising new approach for enforcing information flow policies [1]. Much of the prior work on those languages aims to track insecure information flows statically (i.e., before program execution) [2], [3]. Although such an approach offers no run-time cost, it lacks precision as it may reject safe programs [4]. On the other hand, there were few attempts that were presented to address the same issue by dynamically tracking insecure flows (i.e., during program execution). Such analysis is more convenient for enforcing information flow policies that varies dynamically. Also it offers precision (in comparison to static analysis) by rejecting only insecure executions. However such analysis has a run-time cost and is unable to detect implicit flows via branching on conditional or while-loop [5]. To this end, we explore in this paper how to evaluate the cost of dynamically tracking insecure information flow in the language runtime with the possibility to detect implicit flows. The particular language we consider is a simple while-language that incorporates a mechanism for providing dynamic analysis of information flow properties by adding security labeling to the language semantics. Our approach for measuring the run-time cost of dynamic tracking of insecure flows relies on measuring the execution time of reducing programs written in the language syntax to the outputs they generates. The programs are reduced to their outputs using two prototype interpreters which implements two different operational semantics for the presented language. The first one is a normal interpreter that implements the normal operational semantics of the language, while the other implements an operational semantics which uses straight forward security labeling representation, where every variable has an associated information flow label during its declaration to express its security level. However, security labeling offers overhead to allocate and dynamic track the labels attached to each variable. Thus our approach aims to evaluate this overhead by measuring the difference in execution time when reducing several benchmark programs using both interpreters.

The structure of this paper is organized as follows. The next section introduces the language that we use as the basis for our experiment. Section 3 presents the normal and secure labeling semantics. Section 4 describes the two language implementations, benchmarks, and experimental results. Section 5 discusses the related work, and Section 6 concludes.

II. A SECURITY TYPED WHILE LANGUAGE

We formalize the dynamic information flow tracking in the variables of a simple security typed while language. We call

it secure-while language which is a variant of the standard while language with a mechanism for augmenting declared variables with information flow labels. The syntax of secure-while is shown in Figure 1. Expression (e) includes constants (c), variables (v) and arithmetic and Boolean operations on expressions. A program (p) is a sequential piece of code, whose body consists of a number of variable declarations (vd) and statements (stmt). A statement (stmt) can be an assignment, a conditional or a loop statement. We assume that the information flow labels that augment variables are elements of a simple security lattice L_{LH} with two elements L and H representing low and high confidentiality levels, respectively [6], with the ordering operation L ⊑ H, which reads "less restrictive than" [1]. Hence, one can take the join ⊔ and meet ⊓ of two labels to obtain respectively, more and less restrictive labels (the least upper bound and the greatest lower bound of the two labels, to be precise) than the composed labels. The join operation is useful for computing an upper bound on the security level of an expression that combines sub-expressions at different security levels.

```
e       ::= c | v| e op e
p       ::= (vd | stmt)*
vd      ::= type"{" label "}" ":" v ":="c";"
type    ::= "bool" | "integer"| "string"
label   ::= "L" | "H"
stmt    ::= v ":=" e ";" | "if" e "then" p1 "else" p2 |
            "while" e "do" p
```

Figure 1. Syntax of secure-while language.

Let's now consider some examples written in the secure-While syntax.

A. Example 1

```
1 int {H} :x=5;
2 int {H} :y=6;
3 int {L}:z=0;
3 z=x+y;
```

B. Example 2

```
 int {H} :a=1;
2 int {L} :b=6;
3 int {L} :c=4;
4 int {L} :z=0;
5 if(a==0) then
6 z=b+c;
7 else
8 z=b-c;
```

The first Example is illegal as it permits explicit flow by assigning high level security information to lower level one. The second example is also illegal as it permits an implicit flow by deciding which branch to be taken in if-then-else statement based on the high level security expression of if-then-else (i.e., a in this case) and each branch make an assignment to a lower level security variable.

III. NORMAL AND SECURITY LABELING SEMANTICS

The normal semantics of while language is defined using the big-step operational semantics. This semantics does not perform dynamic information flow analysis, as the information flow labels are excluded from type of each variable (i.e., there is no security labeling). Each big-step evaluation in the normal operational semantics takes the form $(p, v) \rightarrow v'$ where:

- p is a program written in the syntax given in Figure 1 after excluding the security labels from variable declaration.
- v and v' are valuations of variables, which denote, respectively, the store (memory) before and after the evaluation, and represented mathematically by partial functions from variable names to values.

The security labeling semantics for secure-while language is supposed to control explicit and implicit end-to-end information flow to prevent information leakage. Each big-step evaluation in this semantics takes the form $PC \vdash (p, v, s) \rightarrow (v', s')$, where:

- p, v and v' have the same notion as in the normal operational semantics, but we include again the security labels in variable declaration.
- s and s' are security labeling of variables before and after the evaluation, represented by mappings from variable names to their security labels.
- PC is the program counter and is represented initially by the lower security level element (i.e., L security label) in the security lattice LLH. The program counter is used to detect implicit flows via branching on conditional or while loop, following the approach of [7].

Figure 2 and 3 respectively shows the normal and security labeling operational semantics of while and secure-while languages respectively.

1A more complex security lattice with more than two elements might be considered in practice [21].

$$\frac{x \notin \mathrm{dom}(v) \quad t \in \{bool, intger, string\} \quad v' = v \oplus \{x \mapsto val\} \quad \langle p, v' \rangle \to \langle v'' \rangle}{PC \vdash \langle t : x := val; p \rangle \to \langle v'' \rangle}$$

$$\frac{x \in \mathrm{dom}(v) \quad vars(e) \subseteq \mathrm{dom}(v) \quad v' = v \oplus \{x \mapsto v(e)\} \quad \langle p, v \rangle \to \langle v' \rangle}{PC \vdash \langle x := e; p, v \rangle \to \langle v'' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(b) = true \quad \langle p_1, v \rangle \to \langle v' \rangle \quad \langle p_3, v' \rangle \to \langle v'' \rangle}{\langle \text{if } e \text{ then } p_1 \text{ else } p_2; p_3, v \rangle \to \langle v'' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(e) = false \quad \langle p_2, v \rangle \to \langle v' \rangle \quad \langle p_3, v' \rangle \to \langle v'' \rangle}{\langle \text{if } e \text{ then } p_1 \text{ else } p_2; p_3, v \rangle \to \langle v'' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(e) = true \quad \langle p; \text{ while } e \text{ do } p \text{ od}; p', v \rangle \to \langle v' \rangle}{\langle \text{while } e \text{ do } p; p', v \rangle \to \langle v' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(e) = false \quad \langle p', v \rangle \to \langle v' \rangle}{\langle \text{while } e \text{ do } p; p', v \rangle \to \langle v' \rangle}$$

Figure 2. Normal-Operational semantics of While Language without security labeling

IV. IMPLEMENTATION AND EXPERIMENTAL RESULTS

The evaluation of the run-time overhead of dynamic information flow analysis is done by measuring the execution-time in two cases: first when excluding the security labeling of data types in the operational semantics rules of the secure-while language (i.e., by implementing the normal operational semantics presented in figure 2 in a traditional interpreter) and the second case is when including this labeling in the semantics (i.e., by implementing the security labeling operational semantics presented in figure 3 in another interpreter that tracks insecure information flows at run-time). We compared both implementations on the following benchmark programs:

- **Sum**: adds the values of two variables and assign the result to a third one.
- **Implicit-if-true**: Implements an implicit information flow leakage, where the if-then-else conditional expression e evaluates to true.
- **Implicit-if-false**: Implements an implicit information flow leakage, where the if-then-else conditional expression e evaluates to false.

$$\frac{x \notin \mathrm{dom}(v) \quad t \in \{bool, intger, string\} \quad PC = L \quad v' = v \oplus \{x \mapsto val\} \quad s' = s \oplus \{x \mapsto l\} \quad \langle p, v', s' \rangle \to \langle v'', s'' \rangle}{PC \vdash \langle t\ \{l\} : x := val; S, v, s \rangle \to \langle v'', s'' \rangle}$$

$$\frac{x \in \mathrm{dom}(v) \quad vars(e) \subseteq \mathrm{dom}(v) \quad s(e) \sqcup PC \sqsubseteq s(x) \quad v' = v \oplus \{x \mapsto v(e)\} \quad \langle p, v', s \rangle \to \langle v'', s' \rangle}{PC \vdash \langle x := e; p, v, s \rangle \to \langle v'', s' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(e) = true \quad PC \sqcup s(e) \vdash \langle p_1, v, s \rangle \to \langle v', s' \rangle \quad PC \vdash \langle p_3, v', s' \rangle \to \langle v'', s'' \rangle}{PC \vdash \langle \text{if } e \text{ then } p_1 \text{ else } p_2; p_3, v, s \rangle \to \langle v'', s'' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(e) = false \quad PC \sqcup s(e) \vdash \langle p_2, v, s \rangle \to \langle v', s' \rangle \quad PC \vdash \langle p_3, v', s' \rangle \to \langle v'', s'' \rangle}{PC \vdash \langle \text{if } e \text{ then } p_1 \text{ else } p_2; p_3, v, s \rangle \to \langle v'', s'' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(e) = true \quad PC \sqcup s(e) \vdash \langle p; \text{ while } e \text{ do } p \text{ od}; p', v, s \rangle \to \langle v', s' \rangle}{PC \vdash \langle \text{while } e \text{ do } p; p', v, s \rangle \to \langle v', s' \rangle}$$

$$\frac{vars(e) \subseteq \mathrm{dom}(v) \quad \overline{v}(e) = false \quad PC \sqcup s(e) \vdash \langle p', v, s \rangle \to \langle v', s' \rangle}{PC \vdash \langle \text{while } e \text{ do } p; p', v, s \rangle \to \langle v', s' \rangle}$$

Figure 3. Security labeling- operational semantics of secure-While Language.

Implicit-do-while-true: Implements an implicit information flow leakage example, where the while-do conditional expression e evaluates to true.
Implicit-do-while-false: Implements an implicit information flow leakage example, where the while-do conditional expression e evaluates to false.

The measurements were performed on Intel Pentium (R) Dual-Core CPU (2.00GHz) with 3GB of memory running Lunix. Both language implementations were interpreters developed using the language specification formalism ASF+SDF [8]. The choice of ASF+SDF was because it combines two formalisms: Algebraic Specification Formalism (ASF) [9] and Syntax Definition Formalism (SDF) [10]. The later allows the definition of concrete (lexical and context free) and abstract syntax of the domain specific language (DSL), while the former allows a simultaneous definition of a set of conditional equations that defines its semantics. Both formalisms allow the ASF+SDF to integrate the syntax and semantics definition of the DSL in a modular specification, where each module has a set of lexical and context-free grammar rules specified in its SDF part and a set of conditional equations specified in its ASF


```
module whileL
imports expressions
imports values
imports basic/Whitespace
imports basic/Comments

exports

sorts
 Type VarDecl Statement Body  Program

context-free syntax

"Integer"-> Type
"String"-> Type
"Boolean"-> Type

%% Definition of VarDecl
Type ":" Var "=" Value ";" -> VarDecl

Body -> Program

(VarDecl | Statement)*  -> Body

%% Definition of Statement
Var ":=" EXP ";"-> Statement
"if" "(" EXP ")" "then" "{" Body "}" "else" "{" Body"}"
-> Statement
"while" "(" EXP ")" "do"  "{" Body"}" -> Statement

hiddens
context-free start-symbols
Program
```

Figure 4. An implementation of the While language syntax in SDF.

part. Each module can import one or more external modules as well as library modules [11] that are required according to the specification. Figures 4 and 5 show the implementation of the while syntax and its assignment rule as one of its normal operational semantics (presented in Figure 2) in SDF and ASF respectively.

Another important reason for implementing the syntax and semantics of the while language in the ASF+SDF formalism is that it has an interactive development environment called the ASF+SDF Meta-Environment [12], [13] which can be integrated with Eclipse platform [14] that provides functionality for on-line help and error reporting and documentation [15]. Moreover, the ASF+SDF Meta-Environment allows to construct the domain specific languages (DSLs) definitions given their formal specifications in ASF+SDF formalism, edit, check and compile those definitions just like programs. We refer to [12] for more information about implementing domain specific languages in ASF+SDF.

Table I summarizes the results that estimate the execution time of all bench mark programs. As the table shows, the execution time with interpreter A shown in column 2 performs the fastest as it implements the normal operational semantics without no security labeling and hence allowing either explicit or implicit insecure information flows. On the other hand, column three shows that interpreter B is slower than interpreter A. This is because of adding the security labeling to the language semantics (in order to prevent explicit and implicit insecure information flows at run-time) increases the execution time by the amount required for doing information flow analysis dynamically. Column four shows the absolute run-time overhead of dynamic information flow analysis (DIFA), represented by the difference in execution times when using both interpreters for evaluating each benchmark program.

While these experimental results are for a prototype preliminary interpreters for a simple imperative language, these results do suggest that our approach for evaluating the absolute run-time overhead of dynamic flow analysis can be applied to a highly-optimized language implementation.

```
[Assignment]
1 $Varset1:=SetDeclaredVariables($v),
2 $Varset2:=SetExpressionVariables($exp,{}),
3 subset($Varset2,$Varset1)==true,
4 elem($var,$Varset1)==true,
5 $value:=ev-exp-value($exp,$v),
6  $v':=store($v,$var,$value),
7 ====>
8 ev-body($var:=$exp;$body,$v)=
9  ev-body($body,$v')
```

Figure 5. An implementation of the assignment operational semantics rule for While language in ASF.

V. RELATED WORK

Shroff et al. [5], [16] introduced λ^{deps}, a higher-order language with mutable state, to dynamically track the dependencies between program points at run-time and at the same time use the collected set of dependencies to detect indirect information flows. However, λ^{deps} might leak indirect information in the initial run(s) before capturing the appropriate dependencies. As an improvement to λ^{deps}, in the same work, they introduced λ^{deps+} which is initialized with a

TABLE I. BENCHMARK EXPERIMENTAL RESULTS

Benchmark program	Interpreter A	Interpreter B	Run-time overhead of DIFA
Assignment	0.142s	0.211s	0.069s
if-true	0.161s	0.310s	0.149s
if-false	0.172s	0.322s	0.15s
do-while-true	0.234s	0.432s	0.198s
do-while-false	0.251s	0.417s	0.166s
Average	0.19s	0.34s	0.15s

statically generated complete set of dependencies for a given program to detect all indirect information flows at run-time.

Our approach is similar to their approach in that we also track the implicit flows due to conditional and while-loop in our dynamic analysis of information flow using the label of the program counter. However, we are not aware of any implementation for λ^{deps} to our knowledge. Thus the runtime overhead of dynamic information flow analysis in λ^{deps} was not evaluated.

Gurvan Le Guerntic presented a development of a monitor for concurrent programs including synchronization commands [17]. This monitor combines the dynamic and static information flow analyses in order to check the absence of insecure information flow in concurrent programs. Gurvan also investigated the monitor precision as well as its soundness regarding enforcing non-interference property. However the focus of the work presented in [17] was not on the evaluation of run-time overhead by the monitor's dynamic analysis of information flow.

Our work is inspired by that of Austin and Flanagan [18, 19]. They used dynamic information flow analysis in JavaScript to prevent the leak of confidential information caused by malicious JavaScript code. They presented two semantics for λ^{info} [18] a variant of λ-calculus from which the evaluation rules for Featherweight JavaScript (FWJS), a subset of JavaScript can be derived [20]. The first semantics uses universal labeling scheme, where every value has an associated information flow label to track information flow in a straightforward manner. The second one uses Sparse Labeling representation that leaves labels implicit and uses explicit labels when values migrate between different domains. This work is related to ours in that we also use two different types of semantics; the normal and security labeling operational semantics. However, they used Sparse Labeling when migrating between different domains, which is not the focus of our approach.

VI. CONCLUSIONS

There is a growing need for dynamic information analysis in the application programs. This is due its ability to enforce dynamic information flow policies that vary at run-time and its precision in detecting insecure flows. In this paper, we have shown that by using the security labeling in the dynamic operational semantics of a simple imperative language, it is possible to track information flow dynamically with acceptable overhead. By comparing this overhead to the one in case of excluding such type of labeling from the semantics and get the difference, we have been able to extract the absolute run-time cost of dynamic information flow analysis.

As a future work, we are planning to apply our approach to measure the run-time overhead caused by dynamic analysis of secure information flow in more complicated security-typed programming languages such as concurrent ones.

REFERENCES

[1] Sabelfeld, A., and Myers, A. C.: Language-Based information-Flow Security. IEEE J. on Sel. Areas in Comm., 21(1):5– 19, 2003.

[2] Myers, A. C.: Flow: Practical Mostly-Static information Flow Control. In Proceeding of POPL'99, pp. 228–241, ACM, 1999.

[3] Simonet, v., and Rocquencourt, I.: Flow Caml in a Nutshell. In Proc. Of APPSEM-II, pp. 152–165, 2003.

[4] Alejandro, R. and Andrei, S. Dynamic vs. Static Flow-Sensitive Security Analysis. In Proceedings of 23rd IEEE Computer Security Foundations Symposium (CSF'10), pp.186 - 199, 2010.

[5] Shroff, P., Smith, S., and Thober, M.: Securing information Flow via Dynamic Capture of Dependencies. In Journal of Computer Security, 16:673–688, 2008.

[6] Denning, D. E.: A Lattice Model of Secure Information Flow. journal of Commun. ACM, 19(5):236 – 243, 1976.

[7] Molnar. D., Piotrowski, M. , Schultz,D., and Wagner, D: The program counter security model: Automatic detection and removal of control flow side channel attacks. In Proc. of ICISC'05, vol. 3935 of LNCS, pp.156168. Springer, 2005.

[8] Brand, M. G. J. V. D., Deursen, A. V., Heering, J., Jonge, J. M., Kuipers, T., Klint, P. , Moonen, L., Olivier, P., Scheerder, J., Vinju, J.J. , Visser,E., and Visser, J.: The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. In Proc. Of CC'01, vol. 2027 of LNCS, pp. 365–370, Springer, 2001.

[9] Deursen, A. V., Heering, J., and Klint, P.: Language Prototyping: An Algebraic Specification Approach: Vol. V. AMAST Series in Computing, World Scientific, 1996.

[10] Heering, J., Hendriks, P., Klint, P., and Rekers.J.: The Syntax Definition Formalism SDF - Reference Manual, 1989.

[11] Brand, M. G. J. V. D., and Klint, P. Asf+sdf Meta-Environment User Manual Revision 1.134. Technical report, CWI Centrum voor Wiskunde en Informatica, Amsterdam, 2003. Available at: http://www. cwi.nl/projects/MetaEnv/meta.

[12] Brand, M. G. J. V. D., Klint, P., and Vinju, J.J. The Language Specification Formalism ASF+SDF, 2008.

[13] Klint, P.: A Meta-Environment for Generating Programming Environments. In ACM TOSEM, 2(2):176–201, 1993.

[14] Eclipse Platform technical overview. Object Technology International, Inc., 2003.

[15] Brand, M. G. J. V. D., Jong, H. A., Klint, P. and Kooiker, A. T. A language development environment for Eclipse. In Proceedings of OOPSLA Workshop on Eclipse Technology eXchange., 2003.

[16] Shroff, P., Smith, S., and Thober, M.: Dynamic Dependency Monitoring to Secure Information Flow. In Proc. of CSF'07, pp. 203–217, IEEE, 2007.

[17] Le Guernic, G. Confidentiality Enforcement Using Dynamic Information Flow Analyses. PhD thesis, Kansas State University, 2007.

[18] Austin, T. H. and Flanagan, C. Efficient Purely-Dynamic Information Flow Analysis. In Proc. of PLAS 2009, pp. 113–124, ACM, 2009.

[19] Austin, T. H. Dynamic Information Flow Analysis for JavaScript in a Web Browser. PhD thesis, University of California, Santa Cruz, March, 2013.

[20] Austin, T. H., Disney, T., Flanagan, C and Jeffrey, A. Dynamic Information Flow Analysis for Featherweight JavaScript. Technical Report #UCSC-SOE-11-19, University California, Santa Cruz, 2011.

[21] Myers, A. C. and Liskov, B. Protecting Privacy Using the Decentralized Label Model. ACM TOSEM, 9:410 – 442, 2000.

Authors Profile

Doaa Hassan earned her Ph.D. in January, 2012 from Computer and Systems Engineering Department at Faculty of Engineering at Zagazig University - Egypt. As a part of her PhD, she also spent one year and half as Ph.D. candidate at Computer Science Department at Eindhoven University of Technology in Netherlands. Currently, she is affiliated as an Assistant Professor at Computers and Systems Department at National Telecommunication Institute in Cairo- Egypt. She is also a visiting research associate at School of Informatics and Computing at Indiana University - Bloomington. Her research interest focuses on enforcement of information flow policies and using machine learning and data mining techniques for automatic detection of network intrusions and applications malware.

