On Syntactic Lattice-Based Secure Information Flow Analysis and its Soundness

Mohammad Ali ARABI[0000-0001-7088-417X]

Albert-Ludwigs-Universität Freiburg, Freiburg im Breisgau, Baden-Württemberg, DE
mohammad.ali.arabi@saturn.uni-freiburg.de

Abstract. Ensuring secure information flow within programs in the context of multiple sensitivity levels has been widely studied. Especially noteworthy is Dennings' work in secure flow analysis and the lattice model. We introduce Dennings' original approach, then present the work of Volpano *et. al.* that formulates Dennings' approach as a type system and present a notion of soundness for the system that can be viewed as a form of noninterference.

1 Introduction

Computer system security relies in part of *information flow control*, that is, guarding unauthorized information flow. This can be seen in particular as regulations to control data flow between different *security classes*. As an example, a flow from a high-security variable to a low-security variable must not be permitted.

Early works on enforcing flow policies concentrated on run-time mechanisms (cf. [1]). Dorothy and Peter Denning [2] introduced a syntactic method for secure information flow analysis, using a lattice model of security classes. As the work of Dennings lacked a precise discussion on the soundness of their method, Volpano *et. al.* [3] later introduced an equivalent analysis in terms of a type system, to prove the soundness of Dennings' method.

After a brief introduction to lattices, we discuss why it is convenient to use them in modeling security classes. Then we have an introduction to type systems. Provided with enough prerequisites, we state methods of Dennings and Volpano *et. al.*, with a brief discussion on their equivalence. Then we discuss the soundness of the methods.

2 Prerequisites

A very first instance of security classes may be the set $\{L, H\}$, denoting low- and high-security. Then the relation point $L \leq H$ together with the ones derived from reflexivity makes it a totally ordered set, *i. e.* $\leq = \{(L, H), (L, L), (H, H)\}$. Although, the order relation may be non-linear.

In information security, data integrity means maintaining and assuring the accuracy and completeness of data over its entire life cycle [4]. This means that data cannot be modified in an unauthorized or undetected manner. More generally, we can label variables with T and U, standing for trusted and untrusted, respectively, meaning whether the data is retrieved from a reliable source. One may want to combine the classes of integrity with classes of security, resulting $\{TL, TH, UL, UH\}$. Now the two classes TH and UL are not necessarily comparable. So, it will be handy for our approach to support a partially ordered set of security classes.

2.1 Lattice

Definition 1. *Let $(L, \leq)$ be a partially ordered set (poset), and let $S \subseteq L$. Then*

- *$u \in L$ is an **upper bound** of S iff $s \leq u$ for all $s \in S$,*
- *$l \in L$ is a **lower bound** of S iff $l \leq s$ for all $s \in S$,*
- *an upper bound u of S is said to be its **least upper bound**, or **join**, or **supremum**, if $u \leq x$ for each upper bound x of S,*
- *a lower bound l of S is said to be its **greatest lower bound**, or **meet**, or **infimum**, if $x \leq l$ for each lower bound x of S.*

For the next definition, note that join and meet are unique, upon existence. Let u and u' be two different joins of a set S, so we have $u \leq x$ and $u' \leq x$ for any upper bound x of S, hence $u \leq u'$ and $u' \leq u$. The same is the case for meet.

Definition 2. *A poset $(L, \leq)$ is called*

- ***join-semilattice** if every two-element subset $\{a, b\} \in L$ has a join $a \vee b$,*
- ***meet-semilattice** if every two-element subset $\{a, b\} \in L$ has a meet $a \wedge b$,*
- ***lattice** if it is both join-semilattice and meet-semilattice.*

The definition 2 makes $\vee$ and $\wedge$ associative commutative binary operations. Note that both operations are $\leq$-monotone, *i.e.* $a_1 \leq a_2$ and $b_1 \leq b_2$ implies $a_1 \vee b_1 \leq a_2 \vee b_2$ and $a_1 \wedge b_1 \leq a_2 \wedge b_2$. Also note that

$$x_i \leq x_1 \vee \cdots \vee x_n \tag{1}$$

and

$$x_1 \wedge \cdots \wedge x_n \leq x_i, \tag{2}$$

for all $i = 1, \ldots, n$. Lastly, if a poset $(L, \leq)$ is not a lattice, we can extend it into a lattice, by adding all the joins and meets, recursively. It is especially possible concretely if the initial set is finite.

Definition 3. *A **bounded lattice** is a lattice $(L, \leq)$ that has a greatest element $\top \in L$ (also called **maximum** or **top**) and a least element $\bot \in L$ (also called **minimum** or **bottom**), satisfying*

$$\bot \leq x \leq \top, \text{ for all } x \in L. \tag{3}$$

Every non-empty finite lattice is bounded, by letting $\top = \bigvee L$ and $\bot = \bigwedge L$.

Example 1. The set L of all logical formulae generated by a set of atoms A is a lattice with $\rightarrow$ being its order relation, $\vee$ being logical or, and $\wedge$ being logical and. We can also extend L to a bounded lattice, by adding two atomic formulae $\top$ and $\bot$, denoting tautology and contradiction, *i.e.* constants always evaluating to true and false, respectively.

2.2 Type Systems

A type system is a tractable syntactic method to derive a type judgement for various constructs of a computer program, such as variables, expressions, functions or modules [5]. Such a judgement, for our purpose, has the form

$$\gamma \vdash p : \tau,$$

asserting that the program phrase p has the type τ, with respect to identifier typing γ, which is a map from identifiers to types, giving types to all the free identifier in p. Very similar to proof calculi in logic, we can express our type system as a set of axioms and inference rules. A typing judgement is implied by a type system if we can derive it by using inference rules, starting from axioms.

Definition 4. *A* ***typing rule*** *(with n assumptions) is an inference rule of the form*

$$\frac{\gamma_1 \vdash p_1 : \tau_1 \quad \cdots \quad \gamma_n \vdash p_n : \tau_n}{\gamma \vdash p : \tau}$$

which establishes that the typing judgement $\gamma \vdash p : \tau$ holds, if all the assumptions hold. Furthermore, a ***typing axiom*** *is a typing rule with 0 assumptions. A* ***type system*** *is then a set of typing rules.*

Note that the definition is too general, also naming an empty set a type system. Although it is not our concern here, it is handy to restrict the definition a but further, *e.g.* by requiring to have at least one judgement rule for each kind of syntax.

Example 2. A simple example would be a type system for integer-valued expressions [3]. This type system has 3 rules, 2 of which are axioms:

$$\gamma \vdash i : \tau,$$

with i denoting integer literals,

$$\gamma \vdash x : \tau, \text{ if } \gamma(x) = \tau,$$

and the non-axiom rule

$$\frac{\gamma \vdash e : \text{int} \quad \gamma \vdash e' : \text{int}}{\gamma \vdash e + e' : \text{int}}\ .$$

3 Lattice Model of Information Flow and Dennings' Certification Mechanism

The lattice model of information flow was first introduced in Dorothy E. Denning's 1976 work [1], and was used the next year in the joint work of Dorothy E. and Peter J. Denning [2] for the static syntactic analysis of information flow.

3.1 Policy Description and Flow

A flow policy can be represented by a finite lattice $(S, \rightarrow)$, where S is a given set of security classes and $\rightarrow$ a relation specifying permissible flows between pairs of classes. Every storage object x is statically assigned to a security class denoted by underbar $\underline{x}$. Hence $\underline{x} \rightarrow \underline{y}$ means that information flow from x to y is permissible. Every unnamed constant (*i.e.* literal) has the security class $\perp$.

Definition 5. *Information flow from object x to object y, denoted by $x \Rightarrow y$, is the usage of information stored in x to derive information transferred to y.*

A program p is secure if and only if no execution of p results in a flow $x \Rightarrow y$ unless $\underline{x} \rightarrow \underline{y}$. Unfortunately, this definition of security is generally undecidable, so we work instead with the alternative syntactic definition:

Definition 6. *A program p is* ***syntactically secure*** *if for all storage objects x and y,*

$$x \Rightarrow y \text{ is syntactically specified by } p \text{ only if } \underline{x} \rightarrow \underline{y}. \tag{4}$$

An example can demonstrate the difference. Listing 1 shows a Python code, with an assignment `b = c`. Although this part of the code is unreachable, it is not clear before you compile the whole code. A flow from variable `c` to variable `b` is specified in this code, syntactically. So, the code is not syntactically secure, unless such a flow is permitted.

```python
a = 3 + 2 ** 5
if a % 2 == 0:
    b = c
```

Listing 1: Syntactic specification of data flow

3.2 Certification Mechanism

The secure information flow certificate mechanism is introduced in Dennings' work of 1977 [2]. The mechanism is very similar to semantic evaluation, and is done in the compile time. The simple idea is that a flow from objects x_i to objects y_j is permissible iff

$$\underline{x_1} \vee \cdots \vee \underline{x_m} \to \underline{y_1} \wedge \cdots \wedge \underline{y_n}. \tag{5}$$

As an example, let p be "$y := f(x_1, \ldots, x_m)$". So, p indicates information flows $x_i \Rightarrow y$. Hence, this is permissible if $\underline{x_1} \vee \cdots \vee \underline{x_m} \to \underline{y}$. Whenever such a constraint dissatisfied, we will flag the whole program as *uncertified*.

Another example of implicit information flow would be the if statement. So, let p be

$$\texttt{if } x_1 > x_2 \texttt{ then } y_1 := w \texttt{ else } y_2 := y_2 + 1.$$

Here, the flows $x_i \Rightarrow y_i$ are authorized if $\underline{x_1} \vee \underline{x_2} \to \underline{y_1} \wedge \underline{y_2}$. Now let c_1 and c_2 be program parts, *e.g.* assignments. Replacing one of the statements under the if with "$c_1; c_2$", *e.g.*

$$\texttt{if } x_1 > x_2 \texttt{ then } c_1; c_2 \texttt{ else } y_2 := y_2 + 1.$$

we encounter $\underline{x_1} \vee \underline{x_2} \to \underline{c_1; c_2} \wedge \underline{y_2}$. So, we will then need the semantic rule

$$\underline{c_1; c_2} := \underline{c_1} \wedge \underline{c_2}$$

to infer the security classes of the branches, before we can do the security check.

4 Type System for Secure Flow Analysis

Let $(S, \to)$ be a finite lattice of security classes. Let $T = \{\tau \text{ exp}, \tau \text{ var}, \tau \text{ com} \mid \tau \in S\}$ be the set of security types, τ exp denoting the type of an expression with security class τ, τ var denoting variables, and τ com denoting commands. The order relation $\leq$ on T is then defined by:

- τ_1 exp $\leq \tau_2$ exp if $\tau_1 \to \tau_2$,
- τ_1 var $\leq \tau_2$ var if $\tau_1 = \tau_2$,
- τ_1 com $\leq \tau_2$ com if $\tau_2 \to \tau_1$.

In other words, exp, var, and com can be seen as functions from S to T, exp being covariant, and com being contravariant, and var being invariant.

4.1 The Type System

The type system is defined for a block-structured language described below. It consists of phrases denoted by p that are either an expression e or a command c:

$p ::= e \mid c$

$e ::= x \mid l \mid n \mid e + e' \mid e - e' \mid e = e' \mid e < e'$

$c ::= e := e' \mid c; c' \mid \texttt{if } e \texttt{ then } c \texttt{ else } c' \mid \texttt{while } e \texttt{ do } c \mid \texttt{letvar } x := e \texttt{ in } c$

Here, x denotes variables, l locations, and n integer literals. The typing judgements are in the form of

$$\lambda; \gamma \vdash p : \rho,$$

with λ being a location typing (having the type information for locations), γ being an identifier typing, and $\rho \in T$.

The typing rules are listed in figure 4.1 [3].

$$\lambda, \gamma \vdash n : \tau \text{ exp}$$

$$\lambda, \gamma \vdash x : \tau \text{ var} \qquad \text{if } \gamma(x) = \tau \text{ var}$$

$$\lambda, \gamma \vdash l : \tau \text{ var} \qquad \text{if } \lambda(l) = \tau \text{ exp}$$

$$\frac{\lambda, \gamma \vdash e : \tau \text{ exp} \quad \lambda, \gamma \vdash e' : \tau \text{ exp}}{\lambda, \gamma \vdash e + e' : \tau \text{ exp}}$$

$$\frac{\lambda, \gamma \vdash e : \tau \text{ var}}{\lambda, \gamma \vdash e : \tau \text{ exp}}$$

$$\frac{\lambda, \gamma \vdash e : \tau \text{ var} \quad \lambda, \gamma \vdash e' : \tau \text{ exp}}{\lambda, \gamma \vdash e := e' : \tau \text{ cmd}}$$

$$\frac{\lambda, \gamma \vdash c : \tau \text{ cmd} \quad \lambda, \gamma \vdash c' : \tau \text{ cmd}}{\lambda, \gamma \vdash c; c' : \tau \text{ cmd}}$$

$$\frac{\lambda, \gamma \vdash e : \tau \text{ exp} \quad \lambda, \gamma \vdash c : \tau \text{ cmd} \quad \lambda, \gamma \vdash c' : \tau \text{ cmd}}{\lambda, \gamma \vdash \texttt{if } e \texttt{ then } c \texttt{ else } c' : \tau \text{ cmd}}$$

$$\frac{\lambda, \gamma \vdash e : \tau \text{ exp} \quad \lambda, \gamma \vdash c : \tau \text{ cmd}}{\lambda, \gamma \vdash \texttt{while } e \texttt{ do } c : \tau \text{ cmd}}$$

$$\frac{\lambda, \gamma \vdash e : \tau \text{ exp} \quad \lambda, \gamma[x : \tau \text{ var}] \vdash c : \tau' \text{ cmd}}{\lambda, \gamma \vdash \texttt{letvar } x := e \texttt{ in } c : \tau' \text{ cmd}}$$

Fig. 1. Typing rules for secure information flow

Typing is possible through the rule named the *subtyping rule*, stating that

$$\frac{\lambda, \gamma \vdash p : \rho \quad \rho \leq \rho'}{\lambda, \gamma \vdash p : \rho'} \; .$$

Combining the the original rules with the subtyping rule, we can derive more general typing rules, called *syntax-derived typing rules*. The example would be:

$$\frac{\lambda, \gamma \vdash e : \tau \text{ exp} \quad \lambda, \gamma \vdash c : \tau \text{ cmd} \quad \lambda, \gamma \vdash c' : \tau \text{ cmd} \quad \tau' \to \tau}{\lambda, \gamma \vdash \texttt{if } e \texttt{ then } c \texttt{ else } c' : \tau' \text{ cmd}} \; .$$

4.2 Soundness

The soundness of this type system is stated as a *non-interference* property, *i.e.* changing the values of locations with security classes properly greater than τ, would not affect the final value of a location with security class τ.

Having introduced semantic evaluation relations

$$\mu \vdash e \Rightarrow n$$

and

$$\mu \vdash c \Rightarrow \mu',$$

for expressions and commands, respectively, we can formulate our soundness result as the following theorem [3]:

Theorem 1 (Soundness). *Suppose*

- $\lambda \vdash c : \rho$,
- $\mu \vdash c \Rightarrow \mu'$,
- $\nu \vdash c \Rightarrow \nu'$,
- $\mathrm{dom}(\lambda) = \mathrm{dom}(\mu) = \mathrm{dom}(\nu)$, *and*
- $\mu(l) = \nu(l)$ *for all* l *such that* $\lambda(l) \leq \tau$ exp.

Then $\mu'(l) = \nu'(l)$ *for all* l *such that* $\lambda(l) \leq \tau$ exp.

References

1. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5) (1976) 236–243
2. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7) (1977) 504–513
3. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2/3) (1996) 167–188
4. Boritz, J.E.: Is practitioners' views on core concepts of information integrity. International Journal of Accounting Information Systems **6**(4) (2005) 260 – 279
5. Pierce, B.C.: Types and programming languages. MIT Press (2002)