Typesystems for Information Flow Security

René R. Hansen

March 11, 2005

1 Introduction

In this paper we shall see how the concept of a *typesystem* can be used to guarantee that a program does not leak secret information. Using a typesystem has the advantage that it can be done *statically*, i.e., at compile time, whereas security measures like *reference monitors* work at runtime.

First we must define our security model. Here we follow the traditional *information flow* model and define two security levels: high-security (H) and low-security (L). A *leak* is then defined to be any information flow from a high-security variable to a low-security variable. If no such leaks occur in a given program we say that the program is *secure*. This model is formalised as a *lattice*:

$$\mathsf{Level} = (\{\mathsf{H}, \mathsf{L}\}, \sqsubseteq)$$

The chosen security policy (that information can flow from low-security to high-security) is then formalised as the requirement that: $\mathsf{L} \sqsubseteq \mathsf{H}$.

In the remainder of this paper we show how the above notion of information flow security can be incorporated into the While language. This is done by classifying the *variables* of the program into either high-security or low-security and then require that for a program to be secure no information must be leaked from a high-security variable into a low-security variable. Furthermore we develop a special typesystem that can be used to prove (at compile time) that a given program is indeed secure.

2 The While Language

In this section we define the syntax and formal semantics for the While language under investigation. We start with the syntax. The language consists of *arithmetic expressions* (AExp), *boolean expressions* (BExp), and *statements*

$$\langle n, s\rangle \to n \qquad \langle x, s\rangle \to s(x) \quad \text{if } x \in \mathrm{dom}(s)$$

$$\frac{\langle a_1, s\rangle \to n_1 \qquad \langle a_2, s\rangle \to n_2}{\langle a_1 + a_2, s\rangle \to n_1 + n_2} \qquad \frac{\langle a_1, s\rangle \to n_1 \qquad \langle a_2, s\rangle \to n_2}{\langle a_1 * a_2, s\rangle \to n_1 \cdot n_2}$$

$$\frac{\langle a_1, s\rangle \to n_1 \qquad \langle a_2, s\rangle \to n_2}{\langle a_1 - a_2, s\rangle \to n_1 - n_2}$$

Figure 1: Natural semantics for arithmetic expressions

$$\langle \texttt{true}, s\rangle \to tt \qquad \langle \texttt{false}, s\rangle \to ff$$

$$\frac{\langle a_1, s\rangle \to n_1 \qquad \langle a_2, s\rangle \to n_2}{\langle a_1 = a_2, s\rangle \to (n_1 = n_2)} \qquad \frac{\langle a_1, s\rangle \to n_1 \qquad \langle a_2, s\rangle \to n_2}{\langle a_1 \leq a_2, s\rangle \to (n_1 \leq n_2)}$$

$$\frac{\langle b_1, s\rangle \to t_1 \qquad \langle b_2, s\rangle \to t_2}{\langle b_1 \wedge b_2, s\rangle \to (t_1 \wedge t_2)} \qquad \frac{\langle b, s\rangle \to t}{\langle \neg b, s\rangle \to \neg t}$$

Figure 2: Natural semantics for boolean expressions

(Stmt):

$$\begin{array}{lcl} \mathsf{AExp} & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \\ \mathsf{BExp} & ::= & \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid b_1 \wedge b_2 \mid \neg b \\ \mathsf{Stmt} & ::= & \texttt{skip} \mid x \ \texttt{:=} \ a \mid S_1 \texttt{;} S_2 \mid \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \mid \\ & & \texttt{while } b \texttt{ do } S \end{array}$$

Figures 1, 2, and 3 define a natural semantics for arithmetic expressions, boolean expressions, and statements respectively.

3 Typesystem for Information Flow Security

Many modern programming languages incorporate a *typesystem* that guarantees the *type safety* of a program. Typically type safety in such languages amounts to ensuring that arithmetic operations are only performed on values of the right *type*; in this case numbers, e.g., integers. In Java and other object-oriented languages the class hierarchy also induces an extended type structure in a program.

$$\langle \texttt{skip}, s\rangle \to s \qquad \frac{\langle a, s\rangle \to n}{\langle x \ \texttt{:=} \ a, s\rangle \to s[x \mapsto n]}$$

$$\frac{\langle b, s\rangle \to tt \qquad \langle S_1, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'} \qquad \frac{\langle b, s\rangle \to ff \qquad \langle S_2, s\rangle \to s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2, s\rangle \to s'}$$

$$\frac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1\,\texttt{;}\,S_2, s\rangle \to s''} \qquad \frac{\langle b, s\rangle \to ff}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s}$$

$$\frac{\langle b, s\rangle \to tt \qquad \langle S, s\rangle \to s' \qquad \langle \texttt{while}\ b\ \texttt{do}\ S, s'\rangle \to s''}{\langle \texttt{while}\ b\ \texttt{do}\ S, s\rangle \to s''}$$

Figure 3: Natural semantics for statements

In this case type safety also includes checking that methods are only invoked on objects that actually define them.

In the following we shall develop a special type system with the intended goal that type safe programs fulfil the conditions for secure information flow. In other words: well-typed programs do not leak high-security information. We start by defining the base types. Rather than the traditional, e.g., `int` or `bool`, we simply choose security levels as base types:

$$\tau \in \mathsf{Type} = \mathsf{Level} = (\{\mathsf{H}, \mathsf{L}\}, \sqsubseteq)$$

with $\mathsf{L} \sqsubseteq \mathsf{H}$, indicating that information is allowed to flow from L to H but not the other way.

Using the *reference monitor* implementation of the *high watermark* model as inspiration we want our typesystem to track two things:

1. The highest level of any variable *read*
2. The lowest level of any variable *written*

To achieve this, we need two kinds of *typing judgements*: one to be used for expressions and one for statements. For expressions we use the following:

$$\gamma \vdash a : \tau \qquad \text{and} \qquad \gamma \vdash b : \tau$$

Intuitively this means that in the expression a (or b) no variable of a level higher than τ was *read*. This is complemented by the typing judgement for statements:

$$\gamma \vdash S : \tau \text{ cmd}$$

which is then taken to mean the no variable with a level lower than τ was *written*, i.e., assigned to, in the statement S. Note that we use τ and τ cmd to

distinguish the two uses of the type. Such types are called *phrase types* and are formally defined as follows:

$$\rho \in \mathsf{PhraseType} = \tau \mid \tau \text{ cmd}$$

Assigning security level to the variables of a program can then be done through a *type environment* that simply maps variables to their security level, i.e., to their type:

$$\gamma \in \mathsf{TypeEnv} = \mathsf{Var} \to \mathsf{Type}$$

As an example, let us consider the typing rule for the arithmetic expression implementing addition: $\gamma \vdash a_1 + a_2 : \tau$. This means that during execution of the arithmetic expression $a_1 + a_2$ no variable with a security level higher than τ is read. Clearly this implies that no variable with a level higher than τ can be read in either a_1 or a_2 and therefore we must require that $\gamma \vdash a_1 : \tau$ and $\gamma \vdash a_2 : \tau$. This then gives rise to the following typing rule for addition:

$$\frac{\gamma \vdash a_1 : \tau \qquad \gamma \vdash a_2 : \tau}{\gamma \vdash a_1 + a_2 : \tau}$$

However, this rule actually requires that both the constituent arithmetic expressions, a_1 and a_2, have the *same type*. Thus if, for instance, $\gamma \vdash a_1 : \mathsf{L}$ and $\gamma \vdash a_2 : \mathsf{H}$ then it would not be possible to use the above rule to give a type to the expression $a_1 + a_2$. To overcome this we need an extra rule that encodes the intuition that if $\gamma \vdash a_1 : \mathsf{L}$ then it is also the case that $\gamma \vdash a_1 : \mathsf{H}$ because if no variables with a level greater than L are read during evaluation of a_1 then certainly no variables of level greater than H are read. This is formalised as the following rule for arithmetic expressions:

$$\frac{\gamma \vdash a : \tau \qquad \tau \sqsubseteq \tau'}{\gamma \vdash a : \tau'}$$

Using the above rule we can then infer the type for $a_1 + a_2$:

$$\frac{\dfrac{\gamma \vdash a_1 : \mathsf{L} \quad \mathsf{L} \sqsubseteq \mathsf{H}}{\gamma \vdash a_1 : \mathsf{H}} \quad \gamma \vdash a_2 : \mathsf{H}}{\gamma \vdash a_1 + a_2 : \mathsf{H}}$$

The typing rules for arithmetic expressions are shown in Figure 4. The typing rules for Boolean expressions, shown in Figure 5, follow the same pattern as for arithmetic expressions.

For statements we must consider the dual situation: the typesystem should track the level of variables that are written rather than read. Taking assignment as an example the typing $\gamma \vdash x \texttt{ := } a : \tau$ cmd should guarantee that no variables of a level *lower* than τ is written in the assignment. For this to be true it must be the case that the variable, x, has a level of τ or higher; in other words: $\gamma \vdash x : \tau$.

Since information leaks can only occur at assignments, this is the place where we must be careful to define the typing rule in such a way that it prevents high

$$\gamma \vdash n : \tau \qquad\qquad \gamma \vdash x : \tau \quad \text{if } \gamma(x) = \tau$$

$$\frac{\gamma \vdash a_1 : \tau \qquad \gamma \vdash a_2 : \tau}{\gamma \vdash a_1 + a_2 : \tau} \qquad\qquad \frac{\gamma \vdash a_1 : \tau \qquad \gamma \vdash a_2 : \tau}{\gamma \vdash a_1 * a_2 : \tau}$$

$$\frac{\gamma \vdash a_1 : \tau \qquad \gamma \vdash a_2 : \tau}{\gamma \vdash a_1 - a_2 : \tau} \qquad\qquad \frac{\gamma \vdash a : \tau \qquad \tau \sqsubseteq \tau'}{\gamma \vdash a : \tau'}$$

Figure 4: Typesystem for arithmetic expressions

$$\frac{\gamma \vdash b : \tau}{\gamma \vdash \neg b : \tau} \qquad\qquad \frac{\gamma \vdash b_1 : \tau \qquad \gamma \vdash b_2 : \tau}{\gamma \vdash b_1 \wedge b_2 : \tau}$$

$$\frac{\gamma \vdash a_1 : \tau \qquad \gamma \vdash a_2 : \tau}{\gamma \vdash a_1 = a_2 : \tau} \qquad\qquad \frac{\gamma \vdash a_1 : \tau \qquad \gamma \vdash a_2 : \tau}{\gamma \vdash a_1 \leq a_2 : \tau}$$

Figure 5: Typesystem for Boolean expressions

security information to flow into a low security variable. Therefore we must require that the information read/contained in a has level no higher than τ, i.e., the level of the variable. Using the type rules above this requirement can be expressed as: $\gamma \vdash a : \tau$. Putting this together we arrive at the following typing rule for assignment:

$$\frac{\gamma \vdash x : \tau \qquad \gamma \vdash a : \tau}{\gamma \vdash x \quad \texttt{:=} \quad a : \tau \text{ cmd}}$$

As was the case for arithmetic expressions this definition is too strict and will give rise to problems when composing statements of different types. To overcome this we notice that if no variables of a level lower than τ are written ($\gamma \vdash S : \tau$ cmd) then for $\tau' \sqsubseteq \tau$ clearly no variables with level lower than τ' are written either ($\gamma \vdash S : \tau'$ cmd). For arithmetic expressions this idea was formalised as a separate rule; for statements we simply build it into all the rules. Thus the complete rule for typing assignments looks like this:

$$\frac{\gamma \vdash x : \tau \qquad \gamma \vdash a : \tau \qquad \tau \sqsupseteq \tau'}{\gamma \vdash x \quad \texttt{:=} \quad a : \tau' \text{ cmd}}$$

Figure 6 displays the typing rules for statements. In the next section the soundness of the typesystem is stated and discussed.

$$\gamma \vdash \texttt{skip} : \tau \text{ cmd}$$

$$\frac{\gamma \vdash x : \tau \qquad \gamma \vdash a : \tau \qquad \tau \sqsupseteq \tau'}{\gamma \vdash x \;\; \texttt{:=} \;\; a : \tau' \text{ cmd}}$$

$$\frac{\gamma \vdash S_1 : \tau \text{ cmd} \qquad \gamma \vdash S_2 : \tau \text{ cmd}}{\gamma \vdash S_1 \texttt{;} S_2 : \tau \text{ cmd}}$$

$$\frac{\gamma \vdash b : \tau \qquad \gamma \vdash S : \tau \text{ cmd} \qquad \tau \sqsupseteq \tau'}{\gamma \vdash \texttt{while } b \texttt{ do } S : \tau' \text{ cmd}}$$

$$\frac{\gamma \vdash b : \tau \qquad \gamma \vdash S_1 : \tau \text{ cmd} \qquad \gamma \vdash S_2 : \tau \text{ cmd} \qquad \tau \sqsupseteq \tau'}{\gamma \vdash \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 : \tau' \text{ cmd}}$$

Figure 6: Typesystem for statements

4 Soundness of the Typesystem

First we state two simple security properties of the typesystem. These are very similar to the *simple security* and the $*$-property of the Bell/LaPadula (multilevel security) model. First the simple security property that formalises the intuitive meaning of the typing rules for expressions:

Lemma 1 (Simple Security). *Let* $e \in (\mathsf{AExp} \cup \mathsf{BExp})$. *If* $\gamma \vdash e : \tau$ *then for every variable* x *in* e: $\gamma(x) \leq \tau$.

The second property, containment, is the dual formalisation of the intended meaning of the typing rules for statements:

Lemma 2 (Containment). *Let* $S \in \mathsf{Stmt}$. *If* $\gamma \vdash S : \tau$ *cmd then for every variable* x *assigned to in* S: $\gamma(x) \geq \tau$.

While the above properties are significant and show that the typesystem captures the intended information it is less obvious that the typesystem actually guarantees that no information can be leaked from high variables to low variables. In order to formalise and prove this we shall first need the following definition:

Definition 3 (Low-equivalence). *Let* $s_1, s_2 \in \mathsf{State}$ *then* s_1 *and* s_2 *are* low-equivalent, *written* $s_1 \approx_{\mathsf{L}} s_2$, *if and only if* $\mathrm{dom}(s_1) = \mathrm{dom}(s_2)$ *and*

$$\forall x \in \mathrm{dom}(s_1)\text{: } \gamma(x) \leq \tau \;\; \Rightarrow \;\; s_1(x) = s_2(x)$$

Essentially two states are low-equivalent whenever they agree on all the low-variables defined by the state(s). In particular this means that two states that are low-equivalent can differ in any high-variables. Using low-equivalence it is now possible to formulate the soundness of the typesystem:

Theorem 4 (Soundness). *If* $\gamma \vdash S : \tau$ *cmd,* $\langle S, s_1 \rangle \rightarrow s_1'$ *and* $\langle S, s_2 \rangle \rightarrow s_2'$ *then*

$$s_1 \approx_{\mathsf{L}} s_2 \quad \Rightarrow \quad s_1' \approx_{\mathsf{L}} s_2'$$

The above theorem states that if the same statement S is executed in two different states s_1 and s_2 that need only agree on the low-variables, then the two (different) final states s_1' and s_2' will also agree on the low-variables. This shows that even if s_1 and s_2 differ wildly on the high-variables this does not in any way, shape or form influence the final values of any low-variables in the respective final states and therefore: no information could have been leaked from any high-variable to any low-variable. In other words: a well-typed program does not leak information.

This security property is often called a *non-interference* property because it proves that no high-variable can possibly interfere with any low-variable.

5 Exercises

Exercise 1. *Try to infer a type for the following* WHILE *programs the type environment* $\gamma = [\mathtt{l} \mapsto \mathsf{L}, \mathtt{h} \mapsto \mathsf{H}]$*:*

```
h := 42;          l := 42;        h := 17;
l := h            h := l          if (h = 42) then
                                    l := 1
                                  else
                                    l := 0
```

□

Exercise 2. *Can you find a program that is rejected by the* high watermark *model but accepted by the typesystem? And vice versa?* □

Exercise 3. *Implement the type checker in SML.* □

Exercise 4. *Consider the following program:*

```
l := 0;
while (h = 42) do skip;
l := 1
```

1. *Can you infer a type for it?*
2. *Do you think the program is secure? Argue why/why not.*

□

Exercise 5. *What are the advantages/disadvantages of using a typesystem over a reference monitor?* □