A SOUND TYPE SYSTEM FOR SECURE FLOW ANALYSIS

Dennis Volpano
Computer Science Department
Naval Postgraduate School
Monterey, California 93943, U.S.A.

Geoffrey Smith
School of Computer Science
Florida International University
Miami, Florida 33199, U.S.A.

Cynthia Irvine
Computer Science Department
Naval Postgraduate School
Monterey, California 93943, U.S.A.

Abstract

Ensuring secure information flow within programs in the context of multiple sensitivity levels has been widely studied. Especially noteworthy is Denning's work in secure flow analysis and the lattice model [6][7]. Until now, however, the soundness of Denning's analysis has not been established satisfactorily. We formulate Denning's approach as a *type system* and present a notion of soundness for the system that can be viewed as a form of *noninterference*. Soundness is established by proving, with respect to a standard programming language semantics, that all well-typed programs have this noninterference property.

Keywords: type systems, program security, soundness proofs

1. Introduction

The problem of ensuring secure information flow within systems having multiple sensitivity levels has been studied extensively, beginning with the early work of Bell and LaPadula [3]. This was extended by the lattice-model work of Denning [5][6][7] who pioneered *program certification*, an efficient form of static analysis that could be easily incorporated into a compiler to verify secure information flow in programs. Denning's analysis has been characterized as an extension of an axiomatic logic for program correctness by Andrews and Reitman [1]. Other more recent efforts have been aimed at extending the analysis to properly handle language features like procedures [15][16] and nondeterminism [2], while others have focused on integrity analysis only [18][19].

So far there has not been a satisfactory treatment of the soundness of Denning's analysis. After all, we want to be assured that if the analysis succeeds for a given program on some inputs, then the program in some sense executes securely. Denning provides intuitive arguments only in [7][8]. Although a more rigorous account of information flow in terms of classical information theory is given in [8], no formal soundness proof is attempted. Andrews and Reitman [1] do not address the soundness of their flow logic at all. Soundness is considered in Ørbæk [18], but the treatment depends on an "instrumented semantics" where every value is tagged with a security class. These classes are updated for values at run time according to Denning's certification conditions. A similar approach is taken by Mizuno and Schmidt [17]. However, these approaches are unsatisfactory. By modifying the semantics in this way, there is no longer any basis for justifying the soundness of the analysis. Proving soundness in this framework essentially amounts to proving that the analysis is consistent with the instrumented semantics. But then it is fair to ask whether class tags are updated correctly in the instrumented semantics. There is no justification for tag manipulation in the semantics.

We take a type-based approach to the analysis. The certification conditions of Denning's analysis [7][8] are formulated as a simple type system for a deterministic language. A type system is basically a formal system of type inference rules for making judgments about programs. They are usually used to establish the type correctness of programs in a strongly-typed language, for example, Standard ML [20]. However, they are not limited to reasoning about traditional forms of type correctness. They can be regarded, in general, as logical systems in which to reason about a wide variety of program properties. In our case, the property of interest is secure information flow.

Characterizing the analysis as a type system has many advantages. It serves as a formal specification that cleanly separates the security policies from the algorithms for enforcing them in programs. The separation also admits a notion of soundness for the analysis that resembles traditional noninterference [9]. Intuitively, soundness states that variables in a well-typed program do not "interfere" with variables at lower security levels. This is formalized as a type soundness theorem and proved. It is interesting to point out that the soundness proof justifies a more flexible treatment of local variables—in some cases, there is an implicit flow to a local variable, but the flow is actually harmless, so it need not be rejected. The secure flow typing rules merge some traditional type correctness concerns with secure-flow enforcement. Upward information flows are easily accommodated through subtyping. And finally, though not addressed in this paper, the type system can be automated, using standard *type inference* techniques, to analyze programs for secure flows.

We begin with an overview of Denning's lattice model followed by an informal treatment of the type system. Examples are given to show how the typing rules are used. Then we turn our attention to a formal treatment of the type system and prove a soundness theorem with respect to a standard semantics for the language. Other soundness efforts will then be discussed along with language extensions and some directions for future research.

2. The Lattice Model of Information Flow

The lattice model is an extension of the Bell and LaPadula model [3]. In this model, an information flow policy is defined by a lattice $(SC, \leq)$, where SC is a finite set of *security classes* partially ordered by $\leq$. SC may include secrecy classes, like low (L) and high (H), as well as integrity classes, like trusted (T) and untrusted (U), where $L \leq H$ and $T \leq U$. There may be combinations of them as well, like HT.

Every program variable x has a security class denoted by $\underline{x}$. It is assumed that $\underline{x}$ can be determined statically and that it does not vary at run time. If x and y are variables and there is a flow of information from x to y then it is a permissible flow iff $\underline{x} \leq \underline{y}$.

Every programming construct has a *certification condition*. It is a purely syntactic condition relating security classes. Some of these conditions control *explicit flows* while others control *implicit flows*. For example, the statement $y := x$ has the condition $\underline{x} \leq \underline{y}$, that is, the flow of information from the security class of x to that of y must be permitted by the flow policy. This is an example of a condition controlling an explicit flow. The conditions for other constructs, such as **if** statements and **while** loops, control implicit flows. For example, there is always an implicit flow from the guard of a conditional to its branches. For instance, in the statement

$$\textbf{if } x > y \textbf{ then } z := w \textbf{ else } i := i + 1$$

there is an implicit flow from x and y to z and i. So the statement has the certification condition $\underline{x} \oplus \underline{y} \leq \underline{z} \otimes \underline{i}$ where $\oplus$ and $\otimes$ denote least upper bound and greatest lower bound operators respectively. The lattice property makes it possible to enforce these conditions using a simple attribute grammar with synthesized attributes only.

3. An Informal Treatment of the Type System

A type system consists of a set of inference rules and axioms for deriving typing judgments. A typing judgment, for our purposes, has the form

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

This judgment asserts that program (or program phrase) p has type τ with respect to identifier typing γ. An identifier typing is a map from identifiers to types; it gives the types of any free identifiers of p. A judgment follows from the type system if it is the last in a sequence of judgments where each judgment in the sequence is an axiom or one that follows from preceding judgments by a type inference rule.

For example, consider a simple type system for integer-valued expressions. It might contain the following three rules: an axiom $\gamma \vdash i : int$, which asserts that every integer literal i has type int, an inference rule

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

giving us the type of any free identifier x, and the inference rule

$$\frac{\begin{array}{l}\gamma \vdash e : int, \\ \gamma \vdash e' : int\end{array}}{\gamma \vdash e + e' : int}$$

for deducing the types of expressions of the form $e + e'$. In inference rules, the judgments above the horizontal line are *hypotheses* and the judgment below the line is the *conclusion*. So if $\gamma(z) = int$, then

$$\gamma \vdash z + 1 : int$$

is a judgment that follows from the type system. We say $z + 1$ is *well typed* with respect to γ in this case and that it has type *int*. But if $\gamma(z) = bool$ then the judgment no longer follows from the system and we say $z+1$ is not well typed with respect to γ.

The preceding example illustrates a traditional type system. Our secure flow type system is also composed of types and type inference rules, but now the rules enforce secure flow as opposed to data type compatibility. The rules allow secure-flow judgments to be made for expressions and commands in a block-structured, deterministic language.

3.1. *Secure Flow Types*

The types of our system are stratified into two levels. At one level are the *data types*, denoted by τ, which are the security classes of SC. We assume that SC is partially ordered by $\leq$. At the other level are the *phrase types*, denoted by ρ. These include data types, which are the types given to expressions, variable types of the form $\tau\ var$, and command types of the form $\tau\ cmd$. As one would expect, a variable of type $\tau\ var$ stores information whose security class is τ or lower. More novelly, a command c has type $\tau\ cmd$ only if it is guaranteed that every assignment within c is made to a variable whose security class is τ or higher. This is a *confinement* property, needed to ensure secure implicit flows. We extend the partial order $\leq$ to a *subtype relation* which we denote $\subseteq$. The subtype relation is *antimonotonic* (or *contravariant*) in the types of commands, meaning that if $\tau \subseteq \tau'$ then $\tau'\ cmd \subseteq \tau\ cmd$. As usual, there is a type coercion rule that allows a phrase of type ρ to be assigned a type ρ' whenever $\rho \subseteq \rho'$.

3.2. *Secure Flow Typing Rules*

The typing rules guarantee secure explicit and implicit flows as do certification rules in the lattice model. Consider, for example, the typing rule for assignment:

$$\frac{\begin{array}{l}\gamma \vdash e : \tau\ var, \\ \gamma \vdash e' : \tau\end{array}}{\gamma \vdash e := e' : \tau\ cmd}$$

This rule essentially says that in order to ensure that the explicit flow from e' to e is secure, e' and e must agree on their security levels, which is conveyed by τ appearing in both hypotheses of the rule. Note, however, that an upward flow from e' to e is still allowed; if $e : H\ var$ and $e' : L$, then with subtyping, the type of e' can be coerced up to H and the rule applied with $\tau = H$.[1]

[1] Keep in mind that secrecy and integrity are treated uniformly in our type system [4][11], as they are in the lattice model. Examples throughout the paper will be given for secrecy only, but they could alternatively be stated for integrity.

Notice that in the preceding typing rule, the entire assignment is given type $\tau\ cmd$. The reason for this is to control implicit flows. Here is a simple example. Suppose x is either 0 or 1 and consider

$$\mathbf{if}\ x = 1\ \mathbf{then}\ y := 1\ \mathbf{else}\ y := 0$$

Although there is no explicit flow from x to y, there is an implicit flow because x is indirectly copied to y. To ensure that such implicit flows are secure, we use the following typing rule for conditionals:

$$\frac{\begin{array}{l}\gamma \vdash e : \tau,\\ \gamma \vdash c : \tau\ cmd,\\ \gamma \vdash c' : \tau\ cmd\end{array}}{\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' : \tau\ cmd}$$

The intuition behind the rule is that c and c' are executed in a context where information of level τ is implicitly known. For this reason, c and c' may only assign to variables of level τ or higher. Although the rule requires the guard e and branches c and c' to have the same security level, namely τ, it does not prevent an implicit upward flow from e to branches c and c'. Again subtyping can be used to establish agreement, but unlike the case with assignment statements, there are now two ways to get it. The type of e can be coerced to a higher level, or the types of the branches can be coerced to lower levels using the antimonotonicity of command types. In some situations both kinds of coercions are necessary. Observe that no coercions will lead to agreement if there is downward flow from e. The typing rule must reject the conditional in this case.

For example, suppose $\gamma(x) = \gamma(y) = H\ var$. By the preceding typing rule for assignment, we have $\gamma \vdash y := 1 : H\ cmd$ and $\gamma \vdash y := 0 : H\ cmd$. This means that each statement can be placed in a context where high information is implicitly known through the guard of a conditional statement. An example is $\mathbf{if}\ x = 1\ \mathbf{then}\ y := 1\ \mathbf{else}\ y := 0$. With $\tau = H$, the secure flow typing rule for conditionals gives

$$\gamma \vdash \mathbf{if}\ x = 1\ \mathbf{then}\ y := 1\ \mathbf{else}\ y := 0 : H\ cmd$$

So the statement is well typed, as is expected, knowing that since x and y are high variables, the implicit flow from x to y is secure. The resulting type $H\ cmd$ assures us that no low variable is updated in either branch (no write down). This would permit the entire statement to be used where high information again is implicitly known. Now if $\gamma(x) = L\ var$, then the implicit flow is still secure, but establishing this fact within the type system now requires subtyping. One option is to use the antimonotonic subtyping of command types where $H\ cmd \subseteq L\ cmd$ since $L \leq H$. Each branch then is coerced from type $H\ cmd$ to $L\ cmd$ so that we can let $\tau = L$ and get

$$\gamma \vdash \mathbf{if}\ x = 1\ \mathbf{then}\ y := 1\ \mathbf{else}\ y := 0 : L\ cmd$$

On the other hand, we might coerce the type of x upward from L to H and let $\tau = H$ instead. Then once again the conditional has type $H\ cmd$. This would be our only choice if we had to successfully type the conditional, say, as the branch of yet another conditional whose guard is high. And finally, if $\gamma(x) = H\ var$ and $\gamma(y) = L\ var$, then the conditional is not well typed, which is what we would expect since now the implicit flow is downward.

$$\begin{array}{l} \textbf{if } x = 1 \textbf{ then} \\ \quad \textbf{letvar } y := 1 \textbf{ in } c \\ \textbf{else} \\ \quad \textbf{letvar } y := 0 \textbf{ in } c' \end{array}$$

Figure 1. An implicit flow from x to y

3.3. *Local Variable Declarations*

Our core language includes a construct for declaring local variables. A local variable, say x, in our language is declared as

$$\textbf{letvar } x := e \textbf{ in } c$$

It creates x initialized with the value of expression e. The scope and lifetime of x is command c. The initialization can cause an implicit flow, but it is always harmless.

Consider, for instance, the program fragment in Figure 1, for some commands c and c'. If x is high and each instance of y is low, then it might appear as though the program should be rejected because there is a downward implicit flow from x to y. But if c and c' do not update any low variables, that is, each can be typed as high commands, then the program is actually secure, despite the downward flow. The contents of x cannot be "laundered" via y. To see this, suppose x is high. Then the rule for typing conditionals given above forces c and c' to be typed as high commands. By the confinement property, then, neither c nor c' has any assignments to low variables and thus y cannot be assigned to any low variables.

3.4. *Type Soundness*

We prove two interesting security lemmas for our type system, namely Simple Security and Confinement. Simple Security applies to expressions and Confinement to commands. If an expression e can be given type τ in our system, then Simple Security says, for secrecy, that only variables at level τ or lower in e will have their contents read when e is evaluated (no read up). For integrity, it says that every variable in e stores information at integrity level τ. On the other hand, if a command c can be given type $\tau\ cmd$, then Confinement says, for secrecy, that no variable below level τ is updated in c (no write down). For integrity, it states that every variable assigned to in c can indeed be updated by information at integrity level τ.

These two lemmas are used to prove the type system is sound. Soundness is formulated as a kind of noninterference property. Intuitively, it says that variables in a well-typed program do not interfere with variables at lower security levels. That is, if a variable v has security level τ, then one can change the initial values of any variables whose security levels are not dominated by τ, execute the program, and the final value of v will be the same, provided the program terminates successfully.

3.5. *Type Inference*

It is possible to check automatically whether a program is well typed by using standard techniques of *type inference*. While a detailed discussion of type inference is beyond the scope of this paper, the basic idea is to use type variables to represent unknown types and to collect constraints (in the form of type inequalities) that the type variables must satisfy for the program to be well typed. In this way, one can construct a *principal type* for the program that represents all possible types that the program can be given.

4. A Formal Treatment of the Type System

We consider a core block-structured language described below. It consists of phrases, which are either expressions e or commands c:

$$
\begin{array}{llcl}
(\mathit{phrases}) & p & ::= & e \mid c \\
(\mathit{expressions}) & e & ::= & x \mid l \mid n \mid e + e' \mid e - e' \mid e = e' \mid e < e' \\
(\mathit{commands}) & c & ::= & e := e' \mid c; c' \mid \textbf{if } e \textbf{ then } c \textbf{ else } c' \mid \\
 & & & \textbf{while } e \textbf{ do } c \mid \textbf{letvar } x := e \textbf{ in } c
\end{array}
$$

Metavariable x ranges over identifiers, l over *locations* (addresses), and n over integer literals. Integers are the only values. We use 0 for false and 1 for true, and assume that locations are well ordered.

There are no I/O primitives in the language. All I/O is done through free locations in a program. That is, if a program needs to "read input" then it does so by dereferencing an explicit location in the program. Likewise, a program that needs to "write output" does so by an assignment to an explicit location. Locations may also be created during program execution due to local variable declarations. So a partially-evaluated program may contain newly-generated locations as well as those used for I/O.

The types of the core language are stratified as follows.

$$
\begin{array}{llcl}
(\mathit{data\ types}) & \tau & ::= & s \\
(\mathit{phrase\ types}) & \rho & ::= & \tau \mid \tau\ \mathit{var} \mid \tau\ \mathit{cmd}
\end{array}
$$

Metavariable s ranges over the set SC of security classes, which is assumed to be partially ordered by $\leq$. Type $\tau\ \mathit{var}$ is the type of a variable and $\tau\ \mathit{cmd}$ is the type of a command.

The typing rules for the core language are given in Figure 2. We omit typing rules for some of the expressions since they are similar to rule (ARITH). Typing judgments have the form

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

where λ is a *location typing* and γ is an *identifier typing*. The judgment means that phrase p has type ρ, assuming λ prescribes types for locations in p and γ prescribes types for any free identifiers in p. An identifier typing is a finite function mapping identifiers to ρ types; $\gamma(x)$ is the ρ type assigned to x by γ. Also, $\gamma[x : \rho]$ is a modified identifier typing that assigns type ρ to x and assigns type $\gamma(x')$ to any identifier x' other than x. A location typing is a finite function mapping locations to τ types. The notational conventions for location typings are similar to those for identifier typings.

The remaining rules of the type system constitute the subtyping logic and are given in Figure 3. Properties of the logic are established by the following lemmas.

$$\text{(INT)} \quad \lambda;\gamma \vdash n : \tau$$

$$\text{(VAR)} \quad \lambda;\gamma \vdash x : \tau\ var \quad \text{if } \gamma(x) = \tau\ var$$

$$\text{(VARLOC)} \quad \lambda;\gamma \vdash l : \tau\ var \quad \text{if } \lambda(l) = \tau$$

$$\text{(ARITH)} \quad \frac{\lambda;\gamma \vdash e : \tau, \quad \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e + e' : \tau}$$

$$\text{(R-VAL)} \quad \frac{\lambda;\gamma \vdash e : \tau\ var}{\lambda;\gamma \vdash e : \tau}$$

$$\text{(ASSIGN)} \quad \frac{\lambda;\gamma \vdash e : \tau\ var, \quad \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e := e' : \tau\ cmd}$$

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

$$\text{(IF)} \quad \frac{\lambda;\gamma \vdash e : \tau, \quad \lambda;\gamma \vdash c : \tau\ cmd, \quad \lambda;\gamma \vdash c' : \tau\ cmd}{\lambda;\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' : \tau\ cmd}$$

$$\text{(WHILE)} \quad \frac{\lambda;\gamma \vdash e : \tau, \quad \lambda;\gamma \vdash c : \tau\ cmd}{\lambda;\gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ c : \tau\ cmd}$$

$$\text{(LETVAR)} \quad \frac{\lambda;\gamma \vdash e : \tau, \quad \lambda;\gamma[x : \tau\ var] \vdash c : \tau'\ cmd}{\lambda;\gamma \vdash \mathbf{letvar}\ x := e\ \mathbf{in}\ c : \tau'\ cmd}$$

Figure 2. Typing rules for secure information flow

Lemma 4.1 *(Structural Subtyping) If $\vdash \rho \subseteq \rho'$, then either*
(a) ρ is of the form τ, ρ' is of the form τ', and $\tau \le \tau'$,
(b) ρ is of the form $\tau\ var$ and $\rho' = \rho$, or
(c) ρ is of the form $\tau\ cmd$, ρ' is of the form $\tau'\ cmd$, and $\tau' \le \tau$.

Proof. By induction on the height of the derivation of $\vdash \rho \subseteq \rho'$. If the derivation ends with rule (BASE) then (a) is true by the hypothesis of the rule. If it ends with (REFLEX), then $\rho = \rho'$. So if ρ is of the form τ, then (a) holds since $\le$ is reflexive. And if ρ is of the form $\tau\ var$ or $\tau\ cmd$, then (b) or (c) hold, respectively.

Now suppose the derivation ends with rule (TRANS). Then there is a ρ'' such that $\vdash \rho \subseteq \rho''$ and $\vdash \rho'' \subseteq \rho'$ by the hypotheses of the rule. There are three cases:

1. If ρ is of the form τ, then by induction ρ'' is of the form τ'' and $\tau \le \tau''$. So by

$$\text{(BASE)} \quad \frac{\tau \leq \tau'}{\vdash \tau \subseteq \tau'}$$

$$\text{(REFLEX)} \quad \vdash \rho \subseteq \rho$$

$$\text{(TRANS)} \quad \frac{\vdash \rho \subseteq \rho', \ \ \vdash \rho' \subseteq \rho''}{\vdash \rho \subseteq \rho''}$$

$$\text{(CMD}^-\text{)} \quad \frac{\vdash \tau \subseteq \tau'}{\vdash \tau' \ cmd \subseteq \tau \ cmd}$$

$$\text{(SUBTYPE)} \quad \frac{\begin{array}{l}\lambda; \gamma \vdash p : \rho, \\ \vdash \rho \subseteq \rho'\end{array}}{\lambda; \gamma \vdash p : \rho'}$$

Figure 3. Subtyping rules

induction again, ρ' is of the form τ' and $\tau'' \leq \tau'$. And since $\leq$ is transitive, $\tau \leq \tau'$.

2. If ρ is of the form $\tau \ var$, then by induction $\rho'' = \rho$. So by induction again, $\rho' = \rho''$, and hence $\rho' = \rho$.
3. If ρ is of the form $\tau \ cmd$, then by induction ρ'' is of the form $\tau'' \ cmd$ and $\tau'' \leq \tau$. So by induction again, ρ' is of the form $\tau' \ cmd$ and $\tau' \leq \tau''$. So, since $\leq$ is transitive, $\tau' \leq \tau$.

Finally, suppose the derivation ends with (CMD$^-$). Then ρ is of the form $\tau \ cmd$, ρ' is of the form $\tau' \ cmd$, and $\vdash \tau' \subseteq \tau$ by the hypothesis of the rule. By induction, $\tau' \leq \tau$. □

Lemma 4.2 $\subseteq$ *is a partial order.*

PROOF. Reflexivity and transitivity follow directly from rules (REFLEX) and (TRANS). Antisymmetry follows from Lemma 4.1 and the antisymmetry of $\leq$. □

5. The Formal Semantics

The soundness of our type system is established with respect to a natural semantics for closed phrases in the core language. We say that a phrase is *closed* if it has no free identifiers. A closed phrase is evaluated relative to a *memory* μ, which is a finite function from locations to values. The contents of a location $l \in dom(\mu)$ is the value $\mu(l)$, and we write $\mu[l := n]$ for the memory that assigns value n to location l, and value $\mu(l')$ to a location $l' \neq l$; note that $\mu[l := n]$ is an *update* of μ if $l \in dom(\mu)$ and an *extension* of μ otherwise.

The evaluation rules are given in Figure 4. They allow us to derive judgments of the form $\mu \vdash e \Rightarrow n$ for expressions and $\mu \vdash c \Rightarrow \mu'$ for commands. These judgments assert that evaluating closed expression e in memory μ results in integer n and that evaluating closed command c in memory μ results in a new memory μ'. Note that expressions cannot cause side effects and commands do not yield values.

$$(\textsc{base}) \qquad \mu \vdash n \Rightarrow n$$

$$(\textsc{contents}) \qquad \mu \vdash l \Rightarrow \mu(l) \qquad \text{if } l \in dom(\mu)$$

$$(\textsc{add}) \qquad \frac{\mu \vdash e \Rightarrow n, \;\; \mu \vdash e' \Rightarrow n'}{\mu \vdash e + e' \Rightarrow n + n'}$$

$$(\textsc{update}) \qquad \frac{\mu \vdash e \Rightarrow n, \;\; l \in dom(\mu)}{\mu \vdash l := e \Rightarrow \mu[l := n]}$$

$$(\textsc{sequence}) \qquad \frac{\mu \vdash c \Rightarrow \mu', \;\; \mu' \vdash c' \Rightarrow \mu''}{\mu \vdash c;\, c' \Rightarrow \mu''}$$

$$(\textsc{branch}) \qquad \frac{\mu \vdash e \Rightarrow 1, \;\; \mu \vdash c \Rightarrow \mu'}{\mu \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' \Rightarrow \mu'}$$

$$\frac{\mu \vdash e \Rightarrow 0, \;\; \mu \vdash c' \Rightarrow \mu'}{\mu \vdash \textbf{if } e \textbf{ then } c \textbf{ else } c' \Rightarrow \mu'}$$

$$(\textsc{loop}) \qquad \frac{\mu \vdash e \Rightarrow 0}{\mu \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu}$$

$$\frac{\begin{array}{l}\mu \vdash e \Rightarrow 1, \\ \mu \vdash c \Rightarrow \mu', \\ \mu' \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu''\end{array}}{\mu \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu''}$$

$$(\textsc{bindvar}) \qquad \frac{\begin{array}{l}\mu \vdash e \Rightarrow n, \\ l \text{ is the first location not in } dom(\mu), \\ \mu[l := n] \vdash [l/x]c \Rightarrow \mu'\end{array}}{\mu \vdash \textbf{letvar } x := e \textbf{ in } c \Rightarrow \mu' - l}$$

Figure 4. The evaluation rules

We write $[e/x]c$ to denote the capture-avoiding substitution of e for all free occurrences of x in c, and let $\mu - l$ be memory μ with location l deleted from its domain. Note the use of substitution in rule (BINDVAR), which governs the evaluation of **letvar** $x := e$ **in** c. A new location l is substituted for all free occurrences of x in c. The result $[l/x]c$ is then evaluated in the extended memory $\mu[l := n]$, where n is the value of e. By using substitution, we avoid having to introduce an environment mapping x to l. One can view $[l/x]c$ as a partially-evaluated command, perhaps containing other free locations.

6. Type Soundness

We now establish the soundness of the type system with respect to the semantics of the core language. The soundness theorem states that if $\lambda(l) = \tau$, for some location l, then one can arbitrarily alter the initial value of any location l' such

$$(\text{R-VAL}')\quad \frac{\lambda;\gamma \vdash e : \tau\ var,\quad \tau \leq \tau'}{\lambda;\gamma \vdash e : \tau'}$$

$$(\text{ASSIGN}')\quad \frac{\lambda;\gamma \vdash e : \tau\ var,\quad \lambda;\gamma \vdash e' : \tau,\quad \tau' \leq \tau}{\lambda;\gamma \vdash e := e' : \tau'\ cmd}$$

$$(\text{IF}')\quad \frac{\lambda;\gamma \vdash e : \tau,\quad \lambda;\gamma \vdash c : \tau\ cmd,\quad \lambda;\gamma \vdash c' : \tau\ cmd,\quad \tau' \leq \tau}{\lambda;\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau'\ cmd}$$

$$(\text{WHILE}')\quad \frac{\lambda;\gamma \vdash e : \tau,\quad \lambda;\gamma \vdash c : \tau\ cmd,\quad \tau' \leq \tau}{\lambda;\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau'\ cmd}$$

Figure 5. Syntax-directed typing rules

that $\lambda(l')$ is not a subtype of τ, execute the program, and the final value of l will be the same provided the program terminates successfully.

To facilitate the soundness proof, we introduce a *syntax-directed* set of typing rules. The rules of this system are just the rules of Figure 2 with rules (R-VAL), (ASSIGN), (IF), and (WHILE) replaced by their syntax-directed counterparts in Figure 5. The subtyping rules in Figure 3 are not included in the syntax-directed system. We shall write judgments in the syntax-directed system as $\lambda;\gamma \vdash_s p : \rho$. The benefit of the syntax-directed system is that the last rule used in the derivation of a typing $\lambda;\gamma \vdash_s p : \rho$ is uniquely determined by the form of p and of ρ. For example, if p is a **while** loop, then the derivation can only end with rule (WHILE$'$), as opposed to (WHILE) or (SUBTYPE) in the original system. The syntax-directed rules also suggest where a type inference algorithm should introduce coercions.

Next we establish that the syntax-directed system is actually equivalent to our original system. First we need another lemma:

Lemma 6.1 *If $\lambda;\gamma \vdash_s p : \rho$ and $\vdash \rho \subseteq \rho'$, then $\lambda;\gamma \vdash_s p : \rho'$.*

Proof. By induction on the height of the derivation of $\lambda;\gamma \vdash_s p : \rho$.

If the derivation ends with $\lambda;\gamma \vdash_s n : \tau$ by rule (INT), then by Lemma 4.1 ρ' is of the form τ', and $\lambda;\gamma \vdash_s n : \tau'$ by rule (INT).

If the derivation ends with $\lambda;\gamma \vdash_s e : \tau\ var$ either by rule (VAR) or (VARLOC), then $\rho' = \rho$ by Lemma 4.1.

If the derivation ends with $\lambda;\gamma \vdash_s e + e' : \tau$ by rule (ARITH), then $\lambda;\gamma \vdash_s e : \tau$ and $\lambda;\gamma \vdash_s e' : \tau$. By Lemma 4.1, ρ' is of the form τ'. So by induction, $\lambda;\gamma \vdash_s e : \tau'$

and $\lambda; \gamma \vdash_s e' : \tau'$. Thus, $\lambda; \gamma \vdash_s e + e' : \tau'$ by rule (ARITH). The cases where the derivation ends with rule (COMPOSE) or (LETVAR) are similar.

If the derivation ends with $\lambda; \gamma \vdash_s e : \tau$ by rule (R-VAL$'$), then there is a type τ'' such that $\lambda; \gamma \vdash_s e : \tau''\ var$ and $\tau'' \leq \tau$. By Lemma 4.1, ρ' is of the form τ' and $\tau \leq \tau'$. Since $\leq$ is transitive, $\tau'' \leq \tau'$ and so $\lambda; \gamma \vdash_s e : \tau'$ by rule (R-VAL$'$).

If the derivation ends with $\lambda; \gamma \vdash_s e := e' : \tau\ cmd$ by rule (ASSIGN$'$), then there is a type τ'' such that $\lambda; \gamma \vdash_s e : \tau''\ var$, $\lambda; \gamma \vdash_s e' : \tau''$ and $\tau \leq \tau''$. By Lemma 4.1, ρ' is of the form $\tau'\ cmd$ and $\tau' \leq \tau$. Since $\leq$ is transitive, $\tau' \leq \tau''$ and so $\lambda; \gamma \vdash_s e := e' : \tau'\ cmd$ by (ASSIGN$'$). Derivations ending with (IF$'$) and (WHILE$'$) are handled similarly. □

Equivalence is now expressed by the following theorem.

Theorem 6.2 *$\lambda; \gamma \vdash p : \rho$ iff $\lambda; \gamma \vdash_s p : \rho$.*

PROOF. If $\lambda; \gamma \vdash_s p : \rho$, then it is easy to see that $\lambda; \gamma \vdash p : \rho$, because each use of the syntax-directed rules (R-VAL$'$), (ASSIGN$'$), (IF$'$), or (WHILE$'$) can be simulated by a use of (R-VAL), (ASSIGN), (IF), or (WHILE), followed by a use of (SUBTYPE). For example, a use of (ASSIGN$'$)

$$\frac{\begin{array}{l}\lambda; \gamma \vdash e : \tau\ var, \\ \lambda; \gamma \vdash e' : \tau, \\ \tau' \leq \tau\end{array}}{\lambda; \gamma \vdash e := e' : \tau'\ cmd}$$

can be simulated by using (ASSIGN) to show $\lambda; \gamma \vdash e := e' : \tau\ cmd$, using (BASE) and (CMD$^-$) to show $\vdash \tau\ cmd \subseteq \tau'\ cmd$, and using (SUBTYPE) to show $\lambda; \gamma \vdash e := e' : \tau'\ cmd$.

Now suppose that $\lambda; \gamma \vdash p : \rho$. We will prove that $\lambda; \gamma \vdash_s p : \rho$ by induction on the height of the derivation of $\lambda; \gamma \vdash p : \rho$.

If the derivation ends with (INT), (VAR) or (VARLOC), then $\lambda; \gamma \vdash_s p : \rho$ is immediate, and it follows directly by induction if the derivation ends with (ARITH), (COMPOSE) or (LETVAR).

If the derivation ends with (R-VAL), (ASSIGN), (IF), or (WHILE), then $\lambda; \gamma \vdash p : \rho$ follows by an application of the corresponding syntax-directed rule, using the fact that $\leq$ is reflexive.

Finally, suppose the derivation of $\lambda; \gamma \vdash p : \rho$ ends with (SUBTYPE). Then by the hypotheses of this rule, there is a type ρ' such that $\lambda; \gamma \vdash p : \rho'$ and $\vdash \rho' \subseteq \rho$. By induction, $\lambda; \gamma \vdash_s p : \rho'$. Thus, $\lambda; \gamma \vdash_s p : \rho$ by Lemma 6.1. □

From now on, we shall assume that all typing derivations are done in the syntax-directed type system, and therefore shall take $\vdash$ to mean $\vdash_s$.

As final preparation, we establish the following properties of the type system and semantics.

Lemma 6.3 *(Simple Security) If $\lambda \vdash e : \tau$, then for every l in e, $\lambda(l) \leq \tau$.*

PROOF. By induction on the structure of e. Suppose $\lambda \vdash l : \tau$ by rule (R-VAL$'$). Then there is a type τ' such that $\lambda \vdash l : \tau'\ var$ and $\tau' \leq \tau$. Now $\lambda(l) = \tau'$ by rule (VARLOC), so $\lambda(l) \leq \tau$.

Suppose $\lambda \vdash e + e' : \tau$. Then $\lambda \vdash e : \tau$ and $\lambda \vdash e' : \tau$. By two uses of induction, $\lambda(l) \leq \tau$, for every l in e, and for every l in e'. So $\lambda(l) \leq \tau$ for every l in $e + e'$. $\Box$

Simple security applies to both secrecy and integrity. In the case of secrecy, it says that only locations at level τ or lower will have their contents read when e is evaluated (no read up). So if $L \leq H$ and $\tau = L$, then e can be evaluated without reading any H locations.

In the case of integrity, it says that if e has integrity level τ, then every location in e stores information at integrity level τ. For example, if $T \leq U$, where T is trusted and U untrusted, and $\tau = T$, then the lemma states that every location in e stores trusted information.

Lemma 6.4 *(Confinement) If $\lambda; \gamma \vdash c : \tau\ cmd$, then for every l assigned to in c, $\lambda(l) \geq \tau$.*

PROOF. By induction on the structure of c. Suppose $\lambda; \gamma \vdash l := e : \tau\ cmd$ by (ASSIGN$'$). Then there is a type τ' such that $\lambda; \gamma \vdash l : \tau'\ var$, $\lambda; \gamma \vdash e : \tau'$ and $\tau \leq \tau'$. By rule (VARLOC), $\lambda(l) = \tau'$, so $\lambda(l) \geq \tau$.

The lemma follows directly by induction if c is the composition of two commands or a **letvar** command.

Suppose $\lambda; \gamma \vdash$ **while** e **do** $c' : \tau\ cmd$ by (WHILE$'$). Then there is a type τ' such that $\lambda; \gamma \vdash e : \tau'$, $\lambda; \gamma \vdash c' : \tau'\ cmd$ and $\tau \leq \tau'$. By induction, $\lambda(l) \geq \tau'$ for every l assigned to in c'. So, since $\geq$ is transitive, $\lambda(l) \geq \tau$ for every l assigned to in c' and hence for every l assigned to in **while** e **do** c'. The case when c is a conditional is handled similarly. $\Box$

Confinement applies to both secrecy and integrity as well. In the case of secrecy, it says that no location below level τ is updated in c (no write down). For integrity, it states that every location assigned to in c can indeed be updated by information at integrity level τ. So, for example, if $\tau = U$, then the lemma says that no trusted location will be updated when c is evaluated.

The following lemma is a straightforward variant of a lemma given in [10].

Lemma 6.5 *(Substitution) If $\lambda; \gamma \vdash l : \tau\ var$ and $\lambda; \gamma[x : \tau\ var] \vdash c : \tau'\ cmd$, then $\lambda; \gamma \vdash [l/x]c : \tau'\ cmd$.*

Lemma 6.6 *If $\mu \vdash c \Rightarrow \mu'$, then $dom(\mu) = dom(\mu')$.*

Lemma 6.7 *If $\mu \vdash c \Rightarrow \mu'$, $l \in dom(\mu)$, and l is not assigned to in c, then $\mu(l) = \mu'(l)$.*

The preceding two lemmas can be easily shown by induction on the structure of the derivation of $\mu \vdash c \Rightarrow \mu'$. Now we are ready to prove the soundness theorem.

Theorem 6.8 *(Type Soundness) Suppose*
(a) $\lambda \vdash c : \rho$,
(b) $\mu \vdash c \Rightarrow \mu'$,
(c) $\nu \vdash c \Rightarrow \nu'$,
(d) $dom(\mu) = dom(\nu) = dom(\lambda)$, *and*
(e) $\nu(l) = \mu(l)$ *for all* l *such that* $\lambda(l) \leq \tau$.

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

PROOF. By induction on the structure of the derivation of $\mu \vdash c \Rightarrow \mu'$. Here we show just three cases: (UPDATE), (LOOP), and (BINDVAR). The remaining evaluation rules are treated similarly.

(UPDATE). Suppose the evaluation under μ ends with

$$\frac{\begin{array}{l}\mu \vdash e \Rightarrow n,\\ l \in dom(\mu)\end{array}}{\mu \vdash l := e \Rightarrow \mu[l := n]}$$

and the evaluation under ν ends with

$$\frac{\begin{array}{l}\nu \vdash e \Rightarrow n',\\ l \in dom(\nu)\end{array}}{\nu \vdash l := e \Rightarrow \nu[l := n']}$$

and the typing ends with an application of rule (ASSIGN$'$):

$$\frac{\begin{array}{l}\lambda \vdash l : \tau_2\ var,\\ \lambda \vdash e : \tau_2,\\ \tau_1 \leq \tau_2\end{array}}{\lambda \vdash l := e : \tau_1\ cmd}$$

There are two cases:

1. $\tau_2 \leq \tau$. By the Simple Security Lemma, $\lambda(l') \leq \tau_2$ for every l' in e. Since $\leq$ is transitive, $\lambda(l') \leq \tau$ for every l' in e. Thus, by hypothesis (e), $\mu(l') = \nu(l')$ for every l' in e, so $n = n'$. Therefore, $\mu[l := n](l') = \nu[l := n'](l')$ for all l' such that $\lambda(l') \leq \tau$.
2. $\tau_2 \not\leq \tau$. By rule (VARLOC), $\lambda(l) = \tau_2$, so $\lambda(l) \not\leq \tau$. So by hypothesis (e), $\mu[l := n](l') = \nu[l := n'](l')$ for all l' such that $\lambda(l') \leq \tau$.

(LOOP). Suppose $\mu \vdash$ **while** e **do** $c \Rightarrow \mu'$, $\nu \vdash$ **while** e **do** $c \Rightarrow \nu'$, and the typing derivation ends with an application of rule (WHILE$'$):

$$\frac{\begin{array}{l}\lambda \vdash e : \tau_2,\\ \lambda \vdash c : \tau_2\ cmd,\\ \tau_1 \leq \tau_2\end{array}}{\lambda \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau_1\ cmd}$$

Again there are two cases:

1. $\tau_2 \leq \tau$. By the Simple Security Lemma, $\lambda(l) \leq \tau_2$ for every l in e. Since $\leq$ is transitive, $\lambda(l) \leq \tau$ for every l in e. Thus, by hypothesis (e), $\mu(l) = \nu(l)$ for every l in e, and hence $\mu \vdash e \Rightarrow n$ and $\nu \vdash e \Rightarrow n$. Therefore, either the evaluation under μ ends with

$$\frac{\mu \vdash e \Rightarrow 0}{\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu}$$

and under ν with

$$\frac{\nu \vdash e \Rightarrow 0}{\nu \vdash \textbf{while } e \textbf{ do } c \Rightarrow \nu}$$

or it ends under μ with

$$\frac{\begin{array}{l}\mu \vdash e \Rightarrow 1,\\ \mu \vdash c \Rightarrow \mu_1,\\ \mu_1 \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu_2\end{array}}{\mu \vdash \textbf{while } e \textbf{ do } c \Rightarrow \mu_2}$$

and under ν with

$$\frac{\begin{array}{l}\nu \vdash e \Rightarrow 1,\\ \nu \vdash c \Rightarrow \nu_1,\\ \nu_1 \vdash \textbf{while } e \textbf{ do } c \Rightarrow \nu_2\end{array}}{\nu \vdash \textbf{while } e \textbf{ do } c \Rightarrow \nu_2}$$

In the first case, $\mu(l) = \nu(l)$ for all l such that $\lambda(l) \leq \tau$ by hypothesis (e), so we're done. In the second case, by induction, $\mu_1(l) = \nu_1(l)$ for all l such that $\lambda(l) \leq \tau$. By Lemma 6.6, $dom(\mu) = dom(\mu_1)$ and $dom(\nu) = dom(\nu_1)$. So by hypothesis (d), $dom(\mu_1) = dom(\nu_1) = dom(\lambda)$. Thus, by induction again, $\mu_2(l) = \nu_2(l)$ for all l such that $\lambda(l) \leq \tau$.

2. $\tau_2 \not\leq \tau$. By the Confinement Lemma, $\lambda(l) \geq \tau_2$ for every l assigned to in c. Thus, for every l assigned to in c, $\lambda(l) \not\leq \tau$ since otherwise we would have $\tau_2 \leq \tau$ since $\leq$ is transitive. So if $l \in dom(\lambda)$ and $\lambda(l) \leq \tau$, then l is not assigned to in c, and hence is not assigned to in **while** e **do** c. By Lemma 6.7, we have $\mu'(l) = \mu(l)$ and $\nu'(l) = \nu(l)$ for all l such that $\lambda(l) \leq \tau$. Therefore, $\mu'(l) = \nu'(l)$ for all l such that $\lambda(l) \leq \tau$ by hypothesis (e).

(BINDVAR). Suppose the evaluation under μ ends with

$$\frac{\begin{array}{l}\mu \vdash e \Rightarrow n,\\ l \text{ is the first location not in } dom(\mu),\\ \mu[l := n] \vdash [l/x]c \Rightarrow \mu'\end{array}}{\mu \vdash \textbf{letvar } x := e \textbf{ in } c \Rightarrow \mu' - l}$$

and, since $dom(\mu) = dom(\nu)$, the evaluation under ν ends with

$$\frac{\begin{array}{l}\nu \vdash e \Rightarrow n',\\ l \text{ is the first location not in } dom(\nu),\\ \nu[l := n'] \vdash [l/x]c \Rightarrow \nu'\end{array}}{\nu \vdash \textbf{letvar } x := e \textbf{ in } c \Rightarrow \nu' - l}$$

and the typing ends with an application of rule (LETVAR):

$$\frac{\begin{array}{l}\lambda \vdash e : \tau_1,\\ \lambda; [x : \tau_1 \ var] \vdash c : \tau_2 \ cmd\end{array}}{\lambda \vdash \textbf{letvar } x := e \textbf{ in } c : \tau_2 \ cmd}$$

Clearly $\lambda[l : \tau_1] \vdash l : \tau_1\ var$ by (VARLOC). By hypothesis (d) and since $l \notin dom(\mu)$, we have $l \notin dom(\lambda)$. Thus, $\lambda[l : \tau_1]; [x : \tau_1\ var] \vdash c : \tau_2\ cmd$. So by Lemma 6.5, $\lambda[l : \tau_1] \vdash [l/x]c : \tau_2\ cmd$. Also, $dom(\mu[l := n]) = dom(\nu[l := n']) = dom(\lambda[l : \tau_1])$. To apply induction, we just need to show that

$$\nu[l := n'](l') = \mu[l := n](l')$$

for all l' such that $\lambda[l : \tau_1](l') \leq \tau$. If $l' \neq l$ then it follows by hypothesis (e). Otherwise, if $l' = l$, then we must show $n = n'$ if $\tau_1 \leq \tau$. By the Simple Security Lemma, $\lambda(l'') \leq \tau_1$ for every l'' in e. So, if $\tau_1 \leq \tau$, then $\lambda(l'') \leq \tau$ for every l'' in e, since $\leq$ is transitive. Thus by hypothesis (e), $\mu(l'') = \nu(l'')$ for every l'' in e, hence $n = n'$. So by induction, $\nu'(l'') = \mu'(l'')$ for all l'' such that $\lambda[l : \tau_1](l'') \leq \tau$. Therefore, $\nu' - l(l'') = \mu' - l(l'')$ for all l'' such that $\lambda(l'') \leq \tau$. $\Box$

7. Discussion

The early work of Denning [5][6][7] and Andrews and Reitman [1] treated soundness intuitively. More recently, Mizuno and Schmidt [17] and Ørbæk [18] have attempted to give rigorous soundness proofs for Denning-style secure flow analysis. However, both of these works take as their starting point an "instrumented semantics", in which every value is tagged with a security class at runtime; the security tags are updated at runtime in accordance with Denning's certification conditions. Soundness then amounts to the issue of whether their static flow analysis is consistent with the instrumented semantics. But this approach begs the question of whether the flow analysis embodied in the instrumented semantics is, in fact, correct.

In contrast, we use a completely standard semantics for the language, and the type soundness theorem gives a precise operational characterization of the significance of the flow analysis: it tells us that altering the initial values of locations of type τ cannot affect the final values of any locations of type τ', provided that $\tau \not\leq \tau'$. This approach allows us to adopt typing rules whose correctness is not intuitively obvious. For example, our (LETVAR) rule allows the program of Figure 1 to be typed with $x : H$ and $y : L$, even though there is an implicit flow from x to y. But this is not a problem, because our soundness theorem assures us that the implicit flow is harmless. If we had instead used an instrumented semantics, then our (LETVAR) rule would essentially be incorporated into the semantics, where its correctness would have to be taken on faith.

Banâtre *et al.* [2] also take a noninterference approach to soundness, but they consider a nondeterministic language. They associate with a program variable v, a set called the *security variable* of v, denoted $\bar{v}$. Roughly speaking, it is the set of all variables whose values can influence the value of v, either directly or indirectly. They describe an axiomatic, information flow logic for deducing whether a variable is a member of $\bar{v}$, for some variable v. For example, one can deduce that

$$\vdash_1 \{x \notin \bar{z}\}\ y := z\ \{x \notin \bar{y}\}$$

A soundness proposition (Proposition 1, p. 58 [2]) is given that basically says that if $x \notin \bar{y}$, for a given program, then executing the program with any two initial values of x will produce the same sets of final values for y, as long as the program *may* terminate successfully under both initial values. However, the proposition is actually false. The problem is that their language is nondeterministic and although there may be an execution path that leads to successful termination, other paths may not terminate. So it is possible to get different sets of final values for y. For instance, consider the statement

$$
\begin{array}{l}
[\textbf{true} \;\rightarrow\; y := 1 \\
\;\Box \\
\;\textbf{true} \;\rightarrow\; *[x = 1 \;\rightarrow\; \textbf{skip}];\;\; y := 2 \\
]
\end{array}
$$

The statement is a nondeterministic *alternative* statement with two guards, each of which is true. The body of the second guard is a repetitive statement with just one guard, that being $x = 1$. If S denotes this statement, then one can show, using the flow logic, that $\vdash_1 \{Init\}\ S\ \{x \notin \bar{y}\}$, where $Init$ is defined as $\forall x, y.\ x \neq y \;\Rightarrow\; x \notin \bar{y}$. Yet, the set of final values for y when $x = 0$ is $y = 1$ and $y = 2$, and when $x = 1$ is just $y = 1$ because the loop does not terminate.

Denning has used concepts such as uncertainty (entropy) from information theory to formalize the notion of information flow in programs [8]. Basically, if a program, executed in state s, yields a state s', then the execution causes an information flow from x to y if new information about x in state s is available from y in state s'. In other words, we are more certain about the contents of x knowing y after execution than knowing y before. In this setting, soundness seems to require an information-theoretic characterization. It is unclear how such a characterization could be proved with respect to a standard programming language semantics. Such a semantics does not make explicit notions like uncertainty. We have demonstrated that it is possible to formulate and prove soundness without resorting to information theory to get a handle on intuitive ideas like information flow. All that one needs to know about what kind of security is guaranteed by our type system is captured entirely by the type soundness theorem.

7.1. *Core Language Extensions*

The core language we consider has been kept simple, perhaps even emasculate, to better explain our basic proof technique. Although one can imagine many ways to extend the language, there is an obligation to also extend the type system and to prove that well-typed programs preserve the security properties of interest. Many interesting research questions arise. For instance, are there extensions of the type system to handle other features like concurrency and nondeterminism? If so, what is the proper notion of soundness, or, in other words, what security guarantees can be made for all well-typed programs?

Some extensions have straightforward typing rules whose soundness can be shown with only minor changes to the soundness theorem. Two examples are procedures and arrays. Adding arrays is fairly easy with variables already in the language. Procedures, though, require a bit more effort, depending on calling conventions. We have extended the core language with procedures in the style of

Ada 83. A procedure has the form

$$\textbf{proc}(\textbf{in } x_1, \textbf{inout } x_2, \textbf{out } x_3)\ c$$

where c is a command. We limit the number of parameters to three, one for each kind of parameter-passing mode, only to simplify the discussion. Procedure types have the form

$$\tau\ \textbf{proc}(\tau_1, \tau_2\ var, \tau_3\ acc)$$

where *acc* is a new antimonotonic type constructor that stands for *acceptor* in the spirit of Forsythe [21]. An acceptor is a variable that can be assigned to but not evaluated. This is true of **out** parameters in Ada 83 but not Ada 95; consequently, *acc* is not antimonotonic in Ada 95. Type τ comes from typing command c as $\tau\ cmd$, assuming x_1, x_2 and x_3 have types τ_1, $\tau_2\ var$ and $\tau_3\ acc$ respectively. Mode **in** requires a small change in the type soundness theorem but the proof methodology is basically the same.

Other language features pose more serious problems for our type soundness theorem. One is the idea of explicit type casting within programs. Palsberg and Ørbæk [19] propose a system for integrity analysis in programs. They introduce a cast operator called **trust** that can be used to explicitly coerce an untrusted value to a trusted value. (Note that the opposite coercion, from trusted to untrusted, can always be made implicitly, since $T \leq U$.) While such a coercion seems useful pragmatically, including it in the language rules out our type soundness theorem. It seems quite difficult to characterize what is being guaranteed by the flow analysis with such a coercion.

Another source of difficulty is the proper treatment of nondeterminism. Observe, for instance, that if we try to extend the core language with a primitive random number generator $rand(\ )$ and allow an assignment such as $z := rand(\ )$ to be well typed when z is low, then the soundness theorem no longer holds. (Executing this assignment twice from the same memory may produce different final values for z.) A weakness of traditional noninterference is that it is unable to model security in nondeterministic systems [13][14]. So perhaps it is not surprising that nondeterministic language features also cause a problem. As mentioned above, Banâtre *et al.* encountered difficulty when attempting to prove a form of noninterference for nondeterministic programs. New security models, such as Generalized Noninterference [12] should be explored as potential notions of type soundness for new type systems that deal with nondeterministic programs.

8. Summary

We have formulated Denning's secure flow analysis as a type system and proved it sound with respect to a standard programming language semantics for a core deterministic language. The type system cleanly separates the specification of secure flow analysis from its implementation. We expect the core language and type system to serve as a basis for provably-secure programming languages.

9. Acknowledgments

This material is based upon activities supported by the National Security Agency and by the National Science Foundation under Agreements No. CCR-9400592 and

CCR-9414421. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. We would like to thank the referees for their helpful comments.

10. References

[1] G. Andrews, R. Reitman, "An Axiomatic Approach to Information Flow in Programs", *ACM Transactions on Programming Languages and Systems* **2, 1**, (1980), 56–76.

[2] J. Banâtre, C. Bryce, D. Le Métayer, "Compile-time Detection of Information Flow in Sequential Programs", pp. 55–73 in *Proceedings of the European Symposium on Research in Computer Security*, Lecture Notes in Computer Science 875, Springer Verlag, Berlin, 1994.

[3] D. Bell, L. LaPadula, *Secure Computer System: Mathematical Foundations and Model*, MITRE Corp. Technical Report M74-244, 1973.

[4] K. Biba, *Integrity Considerations for Secure Computer Systems*, MITRE Corp. Technical Report ESD-TR-76-372, 1977.

[5] D. Denning, *Secure Information Flow in Computer Systems*, Purdue University Ph.D. Thesis, 1975.

[6] D. Denning, "A Lattice Model of Secure Information Flow", *Communications of the ACM* **19, 5**, (1976), 236–242.

[7] D. Denning, P. Denning, "Certification of Programs for Secure Information Flow", *Communications of the ACM* **20, 7**, (1977), 504–513.

[8] D. Denning, *Cryptography and Data Security*, Addison-Wesley, 1983.

[9] J. Goguen, J. Meseguer, "Security Policies and Security Models", pp. 11–20 in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, 1982.

[10] R. Harper, "A Simplified Account of Polymorphic References", *Information Processing Letters* **51**, (1994), 201–206.

[11] T. Lunt, P. Neumann, D. Denning, R. Schell, M. Heckman, W. Shockley, *Secure Distributed Data Views Security Policy and Interpretation for DMBS for a Class A1 DBMS*, Rome Air Development Center Technical Report RADC-TR-89-313, Vol I, 1989.

[12] D. McCullough, "Specifications for Multi-level Security and a Hook-up Property", in *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, 1987.

[13] D. McCullough, "Noninterference and the Composability of Security Properties", pp. 177–186 in *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, 1988.

[14] J. McLean, "Security Models and Information Flow", pp. 180–187 in *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, 1990.

[15] M. Mizuno, "A Least Fixed Point Approach to Inter-Procedural Information Flow Control", pp. 558–570 in *Proceedings of the 12th National Computer Security Conference*, 1989.

[16] M. Mizuno, A. Oldehoeft, "Information Flow Control in a Distributed Object-Oriented System with Statically-Bound Object Variables", pp. 56–67 in *Proceedings of the 10th National Computer Security Conference*, 1987.

[17] M. Mizuno, D. Schmidt, "A Security Flow Control Algorithm and its Denotational Semantics Correctness Proof", *Formal Aspects of Computing* **4, 6A**, (1992), 722–754.

[18] P. Ørbæk, "Can You Trust Your Data?", pp. 575–589 in *Proceedings of the 1995 Theory and Practice of Software Development Conference*, Lecture Notes in Computer Science 915, 1995.

[19] J. Palsberg, P. Ørbæk, "Trust in the λ-calculus", in *Proceedings of the 1995 Static Analysis Symposium*, Lecture Notes in Computer Science 983, 1995.

[20] L. Paulson, *ML for the Working Programmer*, Cambridge, 1991.

[21] J. Reynolds, *Preliminary Design of the Programming Language Forsythe*, Carnegie Mellon University Technical Report CMU-CS-88-159, 1988.