Secure Flow Typing †

Dennis Volpano
Cynthia Irvine
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943, USA

Abstract

Some of the most promising work in the area of enforcing secure information flow in programs is based on static analyses of source code. However, as yet, these efforts have not had much impact in practice. We present a new approach to analyzing programs statically for secrecy and integrity flow violations. The analysis is characterized as a form of type inference in a *secure flow type system*. The type system provides a uniform framework for traditional type checking of programs and information flow control. Type-correct programs have principal types that characterize how they can be called securely. Applications of the type system include flow analysis of legacy code as well as code written in newly-emerging Web languages like *Java*(tm).

Keywords: secure information flow, certification, type systems, Web programming

1 Introduction

Secure information flow within systems having multiple sensitivity levels has long been a widely-recognized problem. The classical problem is that of a multilevel subject, one with a range of security classes, executing code that either accidentally or maliciously leaks or corrupts sensitive data. Leaking such data is a violation of *secrecy* while its corruption is a violation of *integrity* [5, 10]. Early work in the area was motivated by the need to securely handle information classified at different levels within the government, the military in particular. But the problem is now also apparent within the context of Internet programming and newly-emerging Web programming languages like *Java* [14].

With *Java*, programs, called *Java applets*(tm), can be downloaded from the Internet and executed with user privileges by a *Java*-compatible Web browser like *HotJava*(tm) or Netscape Navigator 2.0. There are obvious secrecy and integrity problems here. For instance a downloaded applet may attempt to make the contents of a user's private files, such as mailfolders or log files, public by mailing them to remote sites. Currently, users have the option, as in *HotJava*, to forbid downloaded code from accessing any local files. This is called the `Applet Host` mode. In fact, this is the only "security mode" available in the Netscape Navigator beta release. But such steps will undoubtedly prove to be too impractical. Useful applets, like mail and other transaction tools, will need access to private files in order to perform their tasks.

This is where the static analysis of code for secure information flow can provide a finer level of security in terms of secrecy as well as integrity. For instance, it can allow access to private files

†Appeared in *Computers and Security*, Vol. 16, No. 2, pp. 137–144, 1997. This material is based upon activities supported by the National Science Foundation under Agreement No. CCR-9612345. 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.

yet guarantee that their contents are not stored in public places. We begin by examining some of these analyses in the next section. Unfortunately, they have had little impact in practice. In Section 3, we propose an alternative form of static analysis based on a powerful, yet very practical, notion called type inference.

2 Information Flow Control Strategies

Secure information flow within systems was studied years ago by Bell and LaPadula [4, 3]. This was followed by the work of Denning who presented a lattice model of secure information flow certification and gave an algorithm for analyzing programs for secure implicit and explicit flows [6, 7, 8]. Denning provided an informal treatment of the soundness of secure flow certification. A more formal treatment of its soundness was given by Mizuno and Schmidt [12], which is discussed in [15].

Andrews and Reitman [1], in a related effort, proposed extending a traditional axiomatic logic for program correctness with secure flow certification. Their emphasis was on a formal specification of certification in a programming logic. They did not consider any practical algorithms for their extended logic, nor was soundness of the logic addressed. These are major drawbacks of their work.

The more recent work of Banâtre, Bryce, and Le Métayer [2] is based on statically detecting flow violations from an accessibility graph constructed for a program. These graphs record information flows among variables at certain program points. This work and secure flow certification are discussed in more detail in the next two sections.

2.1 Denning's Secure Flow Certification

In the lattice model of secure flow certification, a *flow policy* is represented by a poset $< S, \rightarrow >$ where S is a set of security classes and $\rightarrow$ is a partial order, called the *flow relation*, specifying permissible flows between classes. Every variable x is assigned a security class denoted by $\underline{x}$ and it is assumed that $\underline{x}$ is *static* and can be determined from **class** declarations given in a program. If x and y are variables and there is a flow of information from x to y then it is a permissible flow iff $\underline{x} \rightarrow \underline{y}$.

Every programming construct has a *certification* rule. Some rules certify *explicit flows* while others certify *implicit flows*. For example, an assignment statement $y := x$ is certified iff $\underline{x} \rightarrow \underline{y}$, that is, the flow of information from the security class of x to that of y is prescribed by the flow relation. This is an example of certifying an explicit flow. The rules for conditional constructs, such as **if** statements and **while** loops certify implicit flows. For example, the conditional statement

$$\textbf{if } x = 0 \textbf{ then } y := 0 \textbf{ else } z := 0$$

is certified iff $\underline{x} \rightarrow \underline{y}$ and $\underline{x} \rightarrow \underline{z}$.

If the poset $< S, \rightarrow >$ is a lattice, so that for any pair of classes there are unique upper and lower bound classes, then a simple attribute grammar can be given to certify programs. It consists only of synthesized attributes which are security classes computed using least upper bound ($\oplus$) and greatest lower bound ($\otimes$) operators [8]. For instance, the certification condition for the **if** statement above would become the single condition $\underline{x} \rightarrow \underline{y} \otimes \underline{z}$.

2.1.1 Limitations of Certification

One drawback of Denning's flow certification is that it requires security classes of variables to be known at certification time. So it is unsuitable for analyzing legacy code unless the code is preprocessed to include appropriate security class declarations for all program objects.

Another, perhaps more serious practical drawback, is its treatment of procedures. Procedure calls, in flow certification, have the form

$$\textbf{call } q\,(a_1, \ldots, a_m;\; b_1, \ldots, b_n)$$

where the actual input parameters are $a_1, \ldots, a_m$ and the actual output parameters are $b_1, \ldots, b_n$. If procedure q has formal input parameters $x_1, \ldots, x_m$ and formal output parameters

$y_1, \ldots, y_n$, then the security of the call requires that three conditions be verified:

(a) q is secure,

(b) $\underline{a_i} \to \underline{x_i}$ for $i = 1, \ldots, m$, and

(c) $\underline{y_j} \to \underline{b_j}$ for $j = 1, \ldots, n$.

Conditions (b) and (c) certify flow into and out of the procedure respectively. Notice that q can output results of a higher class than the inputs. For example, suppose S consists of security classes L (low) and H (high), with $L \to H$, and consider

procedure *copy*(**in** $x : int\ \ L$, **out** $y : int\ \ H$)
    $y := x$
end

Procedure *copy* copies its input x, declared with security class L, to its output y of class H.

We can imagine erasing the **class** declarations from the procedure so that it could be called to copy from H to H or from L to L, giving us a form of *polymorphism* with respect to classes. To certify calls of procedures that are generic with respect to security classes, Denning proposes replacing conditions (b) and (c) by the single condition

(b') $\underline{a_1} \oplus \cdots \oplus \underline{a_m} \to \underline{b_1} \otimes \cdots \otimes \underline{b_n}$

under the assumptions that

1. procedure q's output parameters are derived solely from the input parameters and information in a least class,
2. any local variables of q are erased upon return, and
3. q does not write into any nonlocal objects.

Notice that (b'), unlike (b) and (c), does not mention the formal parameters of q, only the actual parameters of a call. Thus different calls to q can induce different instances of (b'), so polymorphism is achieved but at a very high cost. Assumptions 1–3 are too restrictive in practice. Useful procedures like a monitor for controlling access to a shared buffer are prohibited. Also compilers normally make no attempt to erase the value of a local variable on the stack. Besides, these assumptions still have to be verified before (b') can be used which is clearly outside the realm of certification.

Mizuno proposed a more flexible strategy for certifying recursive procedures in the style of Denning [11]. It involves generating flow constraints for a program by computing the least fixed point of a set of symbolic flow equations. The equations are constructed according to Denning's certification rules. The strategy, though, is limited to procedures whose arguments are passed by value or by result.

2.2 Banâtre et al's Information Flow Detection

Banâtre, Bryce, and Le Métayer give a compile-time algorithm for detecting information flow in sequential programs whereby variables need not be annotated with security classes [2]. What makes their work appealing is that the algorithm is derived, through a sequence of steps, from an initial axiomatic, information-flow logic. The result is an inference system whose rules are used to transform information flow graphs, called *accessibility graphs*. The result of applying these transformations to an initial graph for a given program is a final accessibility graph indicating whether the contents of one variable at some point in the program can flow into an instance of a variable at some other point. The drawback here is that the number of vertices in the final accessibility graph is at least linear in the size of the program's abstract syntax tree. This means that final graphs are extremely sensitive to program size as we shall see in Section 3.1. Thus they are unsuitable as specifications of the secure flow properties of programs.

3 Secure Flow Typing

Type systems have been used to capture a variety of different kinds of program analyses. A *type system* is basically a set of inference rules with which one infers various properties of programs. Secure information flow is a program property, so we can characterize it as a type correctness issue.

The secure flow type system overcomes the limitations of flow certification mentioned above and does not require calculation of least fixed points as in Mizuno's approach. It is a uniform framework for traditional type checking in programming languages and secure flow enforcement. That is, the issue of secure flows is no longer orthogonal to the more traditional type correctness issue of whether a program is well formed. Further, standard type inference techniques can be used to automate secure flow analysis in a way that makes it more practical.

In the secure flow type system, security classes are basic types, which we denote here by τ, and the typing rules enforce secure information flow. For example, consider the typing rule for an assignment statement $x := e$:

$$\frac{\gamma \vdash x : \tau\ acc, \quad \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ cmd} \tag{1}$$

In order for the assignment to be well typed, it must be that

- x is a variable of type τ *acc*(eptor), meaning x is capable of storing information at security level τ, and
- expression e has type (security class) τ.

Information about x is provided by γ which maps identifiers to types. So, the rule states that in order for the assignment $x := e$ to be judged secure, x must be a variable that stores information at the same security level as e. If this is true, then the rule allows us to ascribe type $\tau\ cmd$ to the entire assignment statement. In general, a statement has type $\tau\ cmd$ only if every variable assigned to in the statement is capable of storing information at security level τ or higher. Note that Denning's flow certification would not be concerned with whether x of $x := e$ is a variable or a constant. But this is not true of rule (1), which addresses the issue of well formedness as well as secure flow.

Another novel aspect of the type system is its use of subtyping. A flow policy $< S, \rightarrow >$ naturally corresponds to a subtype relation. If s_1 and s_2 are members of S, or in other words are basic types, such that $s_1 \rightarrow s_2$, then we say s_1 is a subtype of s_2, written $s_1 \subseteq s_2$.

Notice that typing rule (1) requires x and e to have the same security level. This might appear too restrictive for the rule prevents an *upward* flow from e to x, say for example, if x were high and e low. This is where subtyping comes into play. It allows us to coerce the type of e from low to high to get agreement. A detailed formal treatment of all typing rules and the subtyping logic is outside the scope of this paper and can be found instead in [15].

3.1 Polymorphism and Type Inference

A major advantage of the secure flow type system is that it can be implemented using powerful type inference techniques. A type inference algorithm not only proves whether a procedure is typable, or free of illegal flows, but it also produces a *principal type*, which characterizes how the procedure can be called securely.

A principal type usually comprises a set of *subtype constraints* among security classes, each of the form $\tau_1 \subseteq \tau_2$. Subtype constraints may be generic and involve *type variables* that range over all security classes. These variables can be specialized in many different ways, depending on the procedure's calling context. A context will require them to be specialized in a certain way. As long as the specialization satisfies the constraints, the procedure can be executed without causing any illegal flows. So a procedure is effectively parameterized on the security classes of its formal parameters. In this sense, it is *polymorphic*.

For example, consider the procedure

> **procedure** *copy*(**in** x : *int*, **out** y : *int*)
> **begin**
> $y := x$
> **end**

It has the inferred principal type

$$\forall \alpha, \beta \ \textbf{with}\ \alpha \subseteq \beta \,.\, \beta\ \textbf{proc}(\alpha, \beta\ acc)$$

where α and β are type variables such that α corresponds to the security class of x and β to the security class of y. The principal type succintly conveys how the procedure can be securely called. Any call can be executed securely providing the arguments of the call have security classes that, when substituted for the bound variables α and β of the type, satisfy the constraint $\alpha \subseteq \beta$. The call itself will have type $\beta\ cmd$. For instance, *copy* has type $L\ \mathbf{proc}(L, L\ acc)$ and therefore can be called to copy from low to low, with the call itself being regarded as a low command of type $L\ cmd$. It also has type $H\ \mathbf{proc}(L, H\ acc)$, so it can be called to copy from low to high. But it cannot be called to copy from high to low because $H \not\subseteq L$.

Notice that in a call to *copy*, type variables α and β can be specialized respectively to any security classes τ_1 and τ_2 as long as the subtype constraint $\tau_1 \subseteq \tau_2$ is satisfied. The constraint, in Denning's model, translates into $\tau_1 \rightarrow \tau_2$, which is precisely condition (b'). However, in the secure flow type system, the constraint is inferred automatically as a consequence of typing the assignment $y := x$ by rule (1). *Denning's approach to polymorphism effectively limits all typings of procedures to at most one subtype constraint. namely the constraint corresponding to flow condition* (b'). This greatly limits the kind of polymorphic procedures one can write. In general, a procedure can induce multiple subtype constraints, depending on its definition.

It might appear that the number of subtype constraints in a principal type would grow too quickly, in the size of the program, to be useful in practice. After all, there are programs in the context of traditional subtyping that require the number of constraints in their principal types to grow at least linearly in program size [9]. Obviously, the utility of a type inference algorithm that produces principal types with this many constraints is severely limited. However, our experience has been that this kind of growth is not an issue for secure flow typing in practice. An inferred principal type for a program is typically much smaller than the program itself due to type simplification [13]. Principal types are relatively insensitive to program size.

For instance, consider a new version of procedure *copy*. The original version has an explicit flow from x to y. Suppose we accomplish the same effect through an implicit flow instead. This can be achieved as follows:

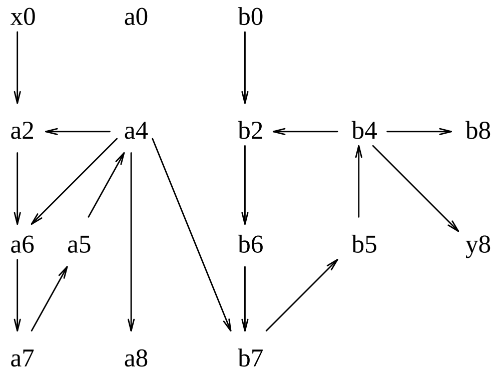

Figure 1: Accessibility graph for new *copy*

procedure *copy*(**in** x : *int*, **out** y : *int*)
    var $a := x$
    var $b := 0$
begin
    while $a > 0$ **do**
        $a := a - 1;$
        $b := b + 1$
    od;
    $y := b$
end

The new version has the same inferred principal type as the original version even though its definition is quite different.

Contrast this insensitivity with that of calculating an accessibility graph for the program as described in [2]. The graph calculated for the original version of *copy* has only the single edge $(x0, y1)$, conveying there is a flow from x at entry point p_0 to y at point p_1. These program points arise from the procedure body expressed with explicit program points, as in $(p_1, y := x)$; there is also a distinguished point, p_0, which represents the entry point of the procedure.

Now compare this simple graph with the graph in Figure 1, which is obtained from the new version of *copy*. The new graph is constructed from *copy* with explicit program points introduced as follows:

```
procedure copy(in x : int, out y : int)
    var a
    var b := 0
    (p1, (p2, a := x);
        (p3, (p4, while a > 0 do
                (p5, (p6, a := a − 1);
                    (p7, b := b + 1)
                )
            od);
            (p8, y := b)
        )
    )
```

The constructed graph has grown in size proportional to the number of program points (there are now 9 such points). Notice that there is a path from $x0$ to $y8$ confirming the implicit flow from x at point p_0 to y at point p_8 in the program. However, the graph does not tell us how *copy* can be called securely, only that there is a flow from x to y. For these reasons, accessibility graphs are unsuitable as user-level specifications of a procedure's information flow properties. Moreover, one can see that the focus of this approach is on identifying flows among instances of program variables. This leads to other problems that do not arise with secure flow typing. For example, pointers and aliasing of locations complicate graph construction, requiring some form of pointer aliasing analysis.

As a final example, consider the library decryption procedure in Figure 2, taken from [2]. The encrypted character array *cipher* is decrypted using *key* and stored in *clear*. We assume that the decryption is done by D and that the cost of doing the decryption is stored in variable *charge*.

The principal type inferred for *decrypt* is

$$\begin{array}{l} \forall\ \alpha, \beta, \nu, \gamma \\ \quad \textbf{with}\ \beta \subseteq \nu, \beta \subseteq \alpha, \gamma \subseteq \beta. \\ \qquad \beta\ \textbf{proc}(\nu, \gamma\ arr, \nu\ arr, \alpha\ var) \end{array}$$

The type has three subtype constraints that govern how *decrypt* can be used securely. Any call of the procedure can be executed securely provided the arguments of the call have security classes that satisfy all the constraints. The call itself will have type $\beta\ cmd$. For instance, the substitution

$$\beta = L,\ \ \nu = H,\ \ \gamma = \alpha = L$$

```
procedure decrypt(in key : int,
        inout cipher, clear : array of char,
        inout charge : int)
    var i := 0
    var unit := unit rate constant
begin
    charge := unit;
    while cipher[i] > 0 do
        if encrypted(cipher[i]) then
            charge := charge + 2 * unit;
            clear[i] := D(cipher[i], key)
        else
            charge := charge + unit;
            clear[i] := cipher[i]
        fi;
        i := i + 1
    od
end
```

Figure 2: The library decryption procedure

satisfies the constraints, so *decrypt* can be called as a procedure with type

$$L\ \textbf{proc}(H, L\ arr, H\ arr, L\ var)$$

and the call will have type $L\ cmd$. The call cannot, however, be typed as $H\ cmd$ unless the argument corresponding to *charge* is high.

Another very useful facet of type inference in this setting is its ability to reveal suspicious code through changes in principal types. For example, if we change the expression

$$charge := charge + 2 * unit;$$

in procedure *decrypt* to the expression

$$charge := charge + key + 2 * unit;$$

in an attempt to deduce the key from the output charge, then the principal type of *decrypt* becomes

$$\begin{array}{l} \forall\ \alpha, \beta, \delta, \nu, \gamma \\ \quad \textbf{with}\ \beta \subseteq \nu, \beta \subseteq \alpha, \gamma \subseteq \beta, \delta \subseteq \nu, \delta \subseteq \alpha. \\ \qquad \beta\ \textbf{proc}(\delta, \gamma\ arr, \nu\ arr, \alpha\ var) \end{array}$$

Notice the two additional subtype constraints $\delta \subseteq \nu$ and $\delta \subseteq \alpha$. The former says that the security level of the *clear* array must be at least that of the input key. This constraint stems from the assignment to *clear* involving a call to the decryption procedure D with the key as an argument. It did indeed arise for the original version of *decrypt* as well, but it was eliminated through type simplification; δ was replaced by ν, giving $\nu \subseteq \nu$ which was deleted. The latter constraint, on the other hand, is new. It says that the security level of the charge parameter must be at least that of the input key. So now any procedure call with a high key will no longer be well typed unless the charge parameter is also high. Such a restriction would likely be unacceptable, but the point here is that type inference clearly reveals it.

4 Discussion

As Denning pointed out, the secure flow problem for a typical programming language is undecidable [8]. Consequently, any sound and recursive logic for proving that programs have no secure flow violations is necessarily incomplete. This is a common tradeoff for the soundness and decidability of a logical system. So like Denning's certification, the type system is incomplete. This means that the type system may rule out some secure programs. Although more experience is needed, the type system has been designed to reduce the number of false positives [15].

The utility of the type system rests on the proper classification of information. Sometimes an algorithm will produce sensitive data from inputs that are not considered sensitive. Examples range from functions that generate cryptographic keys to signal processing algorithms designed to extract target signatures from background noise. Perhaps neither the inputs nor the arithmetic operations used in these algorithms would, separately, be considered sensitive. But they may be used to calculate sensitive data. The type system will not detect that such data are actually sensitive. However, the algorithm can be packaged as a procedure whose type can be asserted to reflect the different security levels of the inputs and outputs. Then the type system can ensure that it is called securely.

We envision secure flow typing being used within Web browsers, specifically, within *Java*-compatible browsers. One approach being investigated is to incorporate it into the Class Loader for *Java* applets. A Class Loader could perform secure flow typing on applet bytecodes and subsume the level of type checking currently done on instructions of the virtual machine. Such typing would ensure secrecy and integrity of downloaded *Java* applets. In the case of integrity, for instance, one can imagine financial centers serving applets to users that perform, say, financial transactions. Some applet may need to make an entry into a financial audit trail and the integrity of the audit trail must be assured. Secure flow typing could be used to certify that an applet does not corrupt the audit trail with low integrity information of a transaction.

5 Summary

Approaches to the static analysis of code for secure information flow have had little impact in practice so far. This paper has described an alternative static analysis that we have argued is an improvement over these other approaches. Secure flow analysis is characterized as a type inference problem. Procedures that have no illegal flows are given principal types that convey how they can be called securely in different contexts. These types can serve as specifications for the secure flow properties of programs. The role of secure flow typing in Web programming, like that encouraged by *Java*, needs further investigation. However, it is clear that such type inference can provide a finer level of security for clients than is currently available in certain Web browsers.

References

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

[2] J. Banâtre, C. Bryce, and D. Le Métayer. Compile-time Detection of Information Flow in Sequential Programs. In *Proceedings 3rd European Symposium on Research in Computer Security*, pages 55–73, Brighton, UK, November 1994. Lecture Notes in Computer Science 875.

[3] D. Bell and E. Burke. A Software Validation Technique for Certification: The Methodology. Technical Report MTR-2932, MITRE Corp., Bedford, MA, 1974.

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

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

[6] D. Denning. *Secure Information Flow in Computer Systems*. PhD thesis, Purdue University, West Lafayette, IN, May 1975.

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

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

[9] M. Hoang and J. Mitchell. Lower Bounds on Type Inference with Subtypes. In *Proceedings 22nd Symposium on Principles of Programming Languages*, pages 176–185, San Francisco, CA, 1995.

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

[11] M. Mizuno. A Least Fixed Point Approach to Inter-Procedural Information Flow Control. In *Proceedings 12th National Computer Security Conference*, pages 558–570, 1989.

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

[13] G. Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping.* PhD thesis, Cornell University, Ithaca, NY, 1991. Tech Report 91-1230.

[14] A. van Hoff, S. Shaio, and O. Starbuck. *Hooked on Java.* Addison-Wesley, 1996.

[15] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *submitted to J. Computer Security*, 1996.