Type Inference and Informative Error Reporting for Secure Information Flow

Zhenyue Deng Geoffrey Smith
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
zdeng01@cis.fiu.edu, smithg@cis.fiu.edu

ABSTRACT

If we classify the variables of a program into various security levels, then a *secure information flow analysis* aims to verify statically that information in the program can flow only in ways consistent with the specified security levels. To make such analysis more practical, this paper proposes a novel type inference approach that gives programmers the freedom to specify the security levels of whichever variables are of interest, leaving the security levels of other variables to be inferred automatically. Type inference in this context is not new, but previous approaches have been based on gathering a set of subtyping constraints from the program, and then solving them with an abstract constraint solver. As a result, it has been difficult to report type errors to users in an informative way. Our inference approach stays closer to the original program, making it easier for us to explain precisely the *source* of each type error. We develop our type inference algorithm for a small imperative language with arrays, and prove that it is sound and complete. We also discuss our techniques for informative error reporting, and illustrate their effectiveness through examples.

1. INTRODUCTION

In today's world of the Internet, the World-Wide Web, and Google, protecting the privacy of sensitive information stored on computer systems is an ever-greater challenge. Techniques such as access control and encryption are useful, of course, but they have a fundamental limitation—they can prevent information from being *released*, but they cannot prevent it from being *propagated*. If a program legitimately needs access to a piece of information, how can we be sure that it will not somehow leak it?

In recent years, there has been a great deal of research into an alternative approach, known as *secure information flow analysis*. (See, for example, the survey by Sabelfeld and Myers [8].) The idea is to classify the variables of a program into various security levels, and then to verify statically (typically through a type-based analysis) that information in variables of higher security levels cannot leak into variables of lower security levels.

To make such secure information flow analysis practical, the programming language needs to be reasonably expressive, and the type system needs to be understandable to programmers. Moreover, the programmer probably wants to specify security levels only for input/output variables, leaving the security levels of auxiliary variables unspecified. Hence it is very convenient for the language to provide *type inference*, allowing the security levels of such auxiliary variables to be inferred automatically.

Type inference for secure information flow analysis has been studied in previous works, such as [12, 3, 5]. In these prior works, type inference uses a constraint-based approach, in which a set of constraints of the form $\alpha \leq \beta$ (where α and β are type terms) is gathered during the type inference process. Then the constraints are analyzed by a constraint solver for satisfiability. This way, the type inference problem is reduced to a more abstract constraint-solving problem. Constraint solving is not always tractable, but it has been shown that atomic constraints over a lattice can be solved in linear time [2].

Though well studied and powerful, constraint-based approaches to security type inference have some drawbacks. First, information about the original program may be lost when the problem is reduced to a constraint-solving problem. Second, while traditional constraint-solving algorithms are good at saying whether or not a set of constraints is satisfiable, they have not been so good at explaining *why*. Particularly when a third-party constraint-solving engine is used, there may be little control over the details of constraint solving and the generation of error messages. The result is that it is very difficult to report type errors back to the programmer in an understandable way. Indeed, good error reporting has been a long-standing challenge for type inference in general [13].

In this work, we propose a novel, non-constraint-based type inference algorithm B. We develop algorithm B for the simple imperative language with arrays that is considered in [1], but we expect that similar languages could be accommodated without difficulty. Our algorithm is tightly based on the security type system and stays close to the original program, allowing us to track detailed information about the program. As a result, we are able to give informative error messages.

Algorithm B relies crucially on the assumption that the

ACM SE'06 March 10-12, 2006, Melbourne, Florida, USA

set of security levels is a lattice. (Indeed, if the set of security levels is the "2-crown" given by $\{C \leq A, C \leq B, D \leq A, D \leq B\}$, then it is easy to see that our inference problem is NP-complete, using a result of Pratt and Tiuryn [6].) A lattice allows us to represent the set of possible types for each inferable variable using an *interval*, which we gradually narrow. In fact, we found that it is enough to maintain just a *lower bound* for each inferable variable, leading to a surprisingly simple algorithm. Initially, B sets all the lower bounds to $\bot$, the lowest level in the security lattice. During inference, it boosts the lower bounds by the minimum amount required by the typing rules; in this way, it achieves sound and complete type inference.

The rest of the paper is organized as follows. In Section 2, we review the language and type system from [1]. We then present algorithm B in Section 3 and prove its soundness and completeness in Section 4; we also argue that B takes quadratic time in the worst case. In Section 5, we describe our techniques for error reporting. In Section 6, we illustrate the capabilities of our approach on an example program that processes some medical information. Finally, Section 7 discusses related work and Section 8 concludes.

2. REVIEW OF THE LANGUAGE AND ITS TYPE SYSTEM

We develop type inference for the simple language and type system presented in [1]. However, we expect that our type inference techniques could be applied to other languages and type systems without too much difficulty.

The language syntax is as follows:

(*phrases*)	p	::=	$e \mid c$
(*expressions*)	e	::=	$x \mid n \mid x[e] \mid x.\mathbf{length} \mid e_1/e_2 \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 = e_2 \mid \ldots$
(*commands*)	c	::=	$x := e \mid x[e_1] := e_2 \mid \mathbf{allocate}\ x[e] \mid \mathbf{skip} \mid \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ e\ \mathbf{do}\ c \mid c_1; c_2$

The only novelty in this language concerns arrays. The lenient semantics given in [1] specifies that out-of-bounds array indices do not disturb the flow of control—erroneous array reads return 0, and erroneous array writes are skipped. As a result, the type system can be simple and permissive.

The *data types* τ of the type system range over some finite lattice of security levels ordered by $\leq$, with join operation $\vee$, least element $\bot$, and greatest element $\top$. The *phrase types* are as follows:

(*phrase types*) $\rho ::= \tau \mid \tau\ var \mid \tau\ cmd \mid \tau_1\ arr\ \tau_2$

Type $\tau\ cmd$ is the type of a command that assigns only to variables of level τ or above. Type $\tau_1\ arr\ \tau_2$ is the type of an array whose contents are of level τ_1 and whose length is of level τ_2. Note that every array type $\tau_1\ arr\ \tau_2$ is subject to a *Global Constraint* that $\tau_2 \subseteq \tau_1$. (Here $\subseteq$ denotes the subtype relation.) The intuitive reason is that an array's contents implicitly includes its length, so the security level of the contents should be at least as great as the security level of the length.

The typing rules are as follows:

(R-VAL) $$\frac{\gamma(x) = \tau\ var}{\gamma \vdash x : \tau}$$

(SUBSCR) $$\frac{\gamma(x) = \tau_1\ arr\ \tau_2,\ \ \gamma \vdash e : \tau_3,}{\gamma \vdash x[e] : \tau_1 \vee \tau_3}$$

(INT) $$\gamma \vdash n : \bot$$

(QUOTIENT) $$\frac{\gamma \vdash e_1 : \tau,\ \ \gamma \vdash e_2 : \tau}{\gamma \vdash e_1/e_2 : \tau}$$

(ASSIGN) $$\frac{\gamma(x) = \tau\ var,\ \ \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ cmd}$$

(ASSIGN-ARR) $$\frac{\gamma(x) = \tau_1\ arr\ \tau_2 \quad \gamma \vdash e_1 : \tau_1 \quad \gamma \vdash e_2 : \tau_1}{\gamma \vdash x[e_1] := e_2 : \tau_1\ cmd}$$

(LENGTH) $$\frac{\gamma(x) = \tau_1\ arr\ \tau_2}{\gamma \vdash x.\mathbf{length} : \tau_2}$$

(ALLOCATE) $$\frac{\gamma(x) = \tau_1\ arr\ \tau_2,\ \ \gamma \vdash e : \tau_2}{\gamma \vdash \mathbf{allocate}\ x[e] : \tau_2\ cmd}$$

(SKIP) $$\gamma \vdash \mathbf{skip} : \top\ cmd$$

(IF) $$\frac{\gamma \vdash e : \tau \quad \gamma \vdash c_1 : \tau\ cmd \quad \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau\ cmd}$$

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

(COMPOSE) $$\frac{\gamma \vdash c_1 : \tau\ cmd \quad \gamma \vdash c_2 : \tau\ cmd}{\gamma \vdash c_1; c_2\ :\ \tau\ cmd}$$

Here γ denotes an *identifier typing*, which maps identifiers to phrase types of the form $\tau\ var$ or $\tau_1\ arr\ \tau_2$.

The subtyping rules are as follows:

(BASE) $\tau_1 \subseteq \tau_2$ if $\tau_1 \leq \tau_2$ in the lattice.

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

(REFLEX) $$\rho \subseteq \rho$$

(TRANS) $$\frac{\rho_1 \subseteq \rho_2,\ \ \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$$

(SUBSUMP) $$\frac{\gamma \vdash p : \rho_1,\ \ \rho_1 \subseteq \rho_2}{\gamma \vdash p : \rho_2}$$

It is shown in [1] that well-typed programs satisfy *noninterference*, which means that changing the initial values of high variables cannot affect the final values of low variables (though it can affect termination).

3. TYPE INFERENCE

In practice, the identifier typing γ will be determined by program declarations that specify the security levels of the program's variables. We allow some of these levels to be

inferred automatically by allowing declarations to include "?". For example, the declaration $a : H\ arr\ ?$ declares that the security level of a's contents is H and the security level of a's length is to be inferred. Given such declarations, we define three sets to record which levels are inferable and which are fixed:

DEFINITION 3.1. FIXED *is the set of variables whose declarations are of the form* $\tau\ var$. FIXED-CONTENTS *is the set of array variables whose declarations are of the form* $\tau\ arr\ ?$ *or* $\tau_1\ arr\ \tau_2$. FIXED-LENGTH *is the set of array variables whose declarations are of the form* $?\ arr\ \tau$, *or* $\tau_1\ arr\ \tau_2$.

Our basic approach toward type inference is to start with an identifier typing γ in which all inferable security levels are set to $\bot$, and then to raise such levels as necessary. As a result, we need to extend $\leq$ to identifier typings:

DEFINITION 3.2. *We say that* $\gamma \leq \gamma'$ *if* $dom(\gamma) = dom(\gamma')$ *and for* $x \in dom(\gamma)$, *either*

- $\gamma(x) = \tau\ var$, $\gamma'(x) = \tau'\ var$, *and* $\tau \leq \tau'$; *moreover, if* $x \in$ FIXED, *then* $\tau = \tau'$, *or*
- $\gamma(x) = \tau_1\ arr\ \tau_2$, $\gamma'(x) = \tau_1'\ arr\ \tau_2'$, $\tau_1 \leq \tau_1'$, *and* $\tau_2 \leq \tau_2'$; *moreover, if* $x \in$ FIXED-CONTENTS, *then* $\tau_1 = \tau_1'$, *and if* $x \in$ FIXED-LENGTH, *then* $\tau_2 = \tau_2'$.

Now we are ready to describe our inference algorithm B. It takes three parameters: an identifier typing γ, a security level pc, and a command c. The purpose of pc ("program counter") is to address implicit flows; it indicates the lowest level of variables that c is allowed to assign to. Initially, pc will be $\bot$. B returns a new identifier typing γ' in which the inferable security levels have been boosted as necessary to get a well-typed program. Algorithm B is given in Figure 1. It uses an auxiliary function B_0 that makes one pass over the program; B simply calls B_0 repeatedly until no more changes to γ occur. B_0 makes use of an auxiliary function $Lev(\gamma, e)$ that returns the minimal type of e under γ.

Algorithm B_0 can be understood by looking at the corresponding command typing rules. We comment on some of the cases:

$B_0(\gamma, pc, x := e)$ handles the assignment command. If x is a fixed variable, the command fails if the join of pc and $Lev(\gamma, e)$ is not less than or equal to $\gamma(x)$. If x is an inferable variable, then $\gamma(x)$ is simply updated to the join of $\gamma(x)$, pc, and $Lev(\gamma, e)$.

$B_0(\gamma, pc, \textbf{allocate}\ x[e])$ is similar to $B_0(\gamma, pc, x := e)$ because array allocation assigns the new array size to the array **length** variable. The content type of the array may also need to be raised in accordance with the Global Constraint on array types (in $\tau_1\ arr\ \tau_2$, we must have $\tau_2 \leq \tau_1$).

In $B_0(\gamma, pc, \textbf{while}\ e\ \textbf{do}\ c_1)$, we simply call B_0 on c_1 with the join of pc and $Lev(\gamma, e)$ as the new pc; we raise pc in this way to reflect the implicit flow of information to c_1.

In $B_0(\gamma, pc, c_1; c_2)$, B_0 is first run on c_1 with pc and γ, returning some γ_1, which is used in turn to run B_0 on c_2.

Finally, notice that the security levels for inferable variables only increase as B_0 executes. Also, whenever B_0 fails, it must be the result of trying to boost the level of a variable in FIXED, or the level of the contents of an array variable in FIXED-CONTENTS, or the level of the length of an array variable in FIXED-LENGTH.

$B(\gamma, pc, c) =$
  let $\gamma_1 = B_0(\gamma, pc, c)$ in
    if $\gamma_1 = \gamma$ then γ else $B(\gamma_1, pc, c)$

$B_0(\gamma, pc, c) =$ case c of
  $x := e$:
    let $\tau\ var = \gamma(x)$
    in if $x \in$ FIXED and $pc \vee Lev(\gamma, e) \not\leq \tau$ then *fail*;
      $\gamma[x := (\tau \vee pc \vee Lev(\gamma, e))\ var]$
  $x[e_1] := e_2$:
    let $\tau_1\ arr\ \tau_2 = \gamma(x)$
    in if $x \in$ FIXED-CONTENTS and
        $\tau_2 \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2) \not\leq \tau_1$ then *fail*;
      $\gamma[x := (\tau_1 \vee \tau_2 \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2))\ arr\ \tau_2]$
  allocate $x[e]$:
    let $\tau_1\ arr\ \tau_2 = \gamma(x)$
    in if $x \in$ FIXED-CONTENTS and
        $\tau_2 \vee pc \vee Lev(\gamma, e) \not\leq \tau_1$ then *fail*;
      if $x \in$ FIXED-LENGTH and
        $pc \vee Lev(\gamma, e) \not\leq \tau_2$ then *fail*;
      $\gamma[x := (\tau_1 \vee \tau_2 \vee pc \vee Lev(\gamma, e))\ arr\ (\tau_2 \vee pc \vee Lev(\gamma, e))]$
  skip:
    γ
  if e **then** c_1 **else** c_2:
    let $\gamma_1 = B_0(\gamma, pc \vee Lev(\gamma, e), c_1)$
    in $B_0(\gamma_1, pc \vee Lev(\gamma_1, e), c_2)$
  while e **do** c_1:
    $B_0(\gamma, pc \vee Lev(\gamma, e), c_1)$
  $c_1; c_2$:
    $B_0(B_0(\gamma, pc, c_1), pc, c_2)$

$Lev(\gamma, e) =$ case e of
  x: τ, where $\gamma(x) = \tau\ var$
  $e_1 + e_2 : Lev(\gamma, e_1) \vee Lev(\gamma, e_2)$
  x.**length** : τ_2, where $\gamma(x) = \tau_1\ arr\ \tau_2$
  $x[e] : \tau_1 \vee Lev(\gamma, e)$, where $\gamma(x) = \tau_1\ arr\ \tau_2$

Figure 1: Type Inference Algorithm B

4. PROPERTIES OF ALGORITHM B

In this section, we establish the soundness and completeness of algorithm B with respect to the typing rules, and we show that B runs in quadratic time. Proofs are omitted due to space restrictions.

LEMMA 4.1. $B_0(\gamma, pc, c)$ *either fails or returns* γ' *such that* $\gamma \leq \gamma'$.

LEMMA 4.2. $B(\gamma, pc, c)$ *either fails or returns* γ' *such that* $\gamma \leq \gamma'$.

LEMMA 4.3. $Lev(\gamma, e)$ *returns the least type* τ *such that* $\gamma \vdash e : \tau$.

LEMMA 4.4. *If* $\gamma \leq \gamma'$, *then* $Lev(\gamma, e) \leq Lev(\gamma', e)$.

LEMMA 4.5 (SOUNDNESS). *If* $B_0(\gamma, pc, c)$ *succeeds and returns* γ, *then* $\gamma \vdash c$: *pc cmd*.

COROLLARY 4.6 (SOUNDNESS). *If $B(\gamma, pc, c)$ succeeds and returns γ', then $\gamma' \vdash c : pc\ cmd$.*

LEMMA 4.7 (COMPLETENESS). *If $\gamma' \vdash c : \tau\ cmd$ and $\gamma \leq \gamma'$, then $B_0(\gamma, \tau, c)$ succeeds and returns γ'' such that $\gamma'' \leq \gamma'$.*

COROLLARY 4.8 (COMPLETENESS). *If $\gamma' \vdash c : \tau\ cmd$ and $\gamma \leq \gamma'$, then $B(\gamma, \tau, c)$ succeeds and returns γ'' such that $\gamma'' \leq \gamma'$.*

We can calculate the running time of algorithm B as follows. Suppose that the length of c is n. Then $B_0(\gamma, pc, c)$ runs in time $O(n)$, assuming that the lattice operations $\vee$ and $\leq$ can be done in constant time. The running time of $B(\gamma, pc, c)$ is therefore determined by the number of calls to B_0 that are required. If the height of the lattice is h, then it is easy to see that the maximum number of calls to B_0 is at $O(nh)$, since there are at most $O(n)$ inferable variables, each of which can be boosted at most h times. Hence B's total running time is $O(n^2h)$ in the worst case. (We have in fact constructed a sequence of assignments whose type inference requires quadratic time, but we suspect that such cases are rare.)

5. ERROR REPORTING

A implementation of our type inference algorithm B has been developed in Java. In this section, we describe the techniques that we have developed to do good error reporting.

As discussed above, algorithm B begins by setting all inferable security levels to $\bot$. As it runs, it raises these levels as necessary. It fails whenever it finds that a fixed security level needs to be raised. When such a failure occurs, it is important to be able to explain the *source* of the failure to the programmer, so that he or she can understand the problem and can remedy it.

For example, suppose that $\gamma(x) = L\ var$ and $x \in$ FIXED. Then B will fail if it finds that the type of x needs to be raised to $H\ var$, where $H \not\leq L$. How might the programmer remedy this problem? The most obvious thing to try is to change the declaration of x to $H\ var$. Of course this may not work, because it could lead to other failures. More seriously, the programmer presumably has a reason for declaring x as $L\ var$—for example, perhaps x is intended to be output on a public channel. In this case, raising the level of x is not an option.

What is more useful is to understand precisely *why* B finds it necessary to raise the level of x. With such an understanding, the programmer may be able to modify the program so that x no longer needs to be raised. We believe that our approach to type inference has an advantage over the constraint-gathering approaches that have been used in previous work, because it is easier for us to explain type errors in terms of the source program. To this end, we introduce the notions of *principal variables* and *security level history*.

First, the *principal variables* of an expression e are any minimal set of variables within e that determine its type. For example, suppose we use lattice shown in Figure 2. If $m : M$, $w1 : W1$, and $w2 : W2$, then the expression $m + w1 * w2$ has type H and either $\{m, w2\}$ or $\{m, w1\}$ could be taken as its principal variables. The intent is simply to find *some* small explanation for the type of the expression.

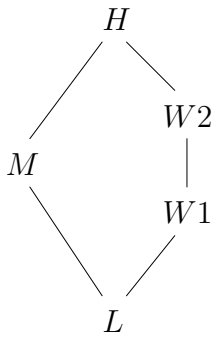

Figure 2: An example security lattice

Next, each inferable variable has a *security level history* which is a list of sets of principal variables that contributed to the rise of the variable. Whenever the inferable variable is raised to a new higher security level, the set of principal variables for the responsible expression is added to the front of the security level history. For example, if the assignment $x := e$ causes the type of x to be raised because of the explicit flow from e, then the principal variables of e are added to the front of x's security level history.

But suppose in this case that x is a fixed variable. Then the attempt to raise the type of x will cause algorithm B to fail. Now we can "blame" the principal variables of e for the type error. Of course, we must also explain why those principal variables got the types they have; this requires that we consult *their* security level histories as well. Repeating this process, we generate the complete error report in a cascading manner.

We illustrate our techniques by considering two examples. First, we demonstrate how accurate error reporting is given through the use of principle variables. Using the security lattice above, consider the following program, where $l : L$, $m : M$, $w1 : W1$, and $w2 : W2$.

```
while (m + l > 0) do {
  while (w1 + w2 > 0) do {
     l := 7;
     w2 := w2 - 1
  };
  m := m - 1
}
```

We run the type checker on the program and get the following error report:

```
tests/example1:16.5: illegal implicit flow
    l  := 7;
    ^
--left-hand side, l, has type L by declaration
--pc has type H because
  tests/example1:14.8: in guard of while statement
  while (m + l > 0) do {
         ^
  m has type M by declaration
and
  tests/example1:15.15: in guard of while statement
    while (w1 + w2 > 0) do {
               ^
  w2 has type W2 by declaration
```

```
and
  the join of M and W2 is H
```

From the error report, it is clear that the assignment to the L variable l is not allowed in the inner loop body because pc is H. And pc is H as a result of variable m in the outer loop guard and variable $w2$ in the inner loop guard. The error report is accurate and closely based on the source program and the typing rules.

Given a detailed error report, we are in a position to modify the program to try to correct the errors. For example, we may be able to change the declarations of some variables. Or, more interestingly, we may be able to rearrange the code to eliminate an unintended implicit flow.

6. A LARGER EXAMPLE

In this section, we develop a larger example involving the processing of medical data. To begin with, we assume a three-level security lattice with levels L (low), M (medium), and H (high) satisfying $L < M < H$. We store information about a group of patients in several arrays:

- `hiv[]` records the HIV status of each patient.
- `charges[]` records the charges for all of the medical incidents of all of the patients, arranged patient by patient.
- `start[]` tells where the charges for each patient begin.

For example, if patient 0 is HIV-negative and has had three incidents, with charges of 100, 75, and 20 dollars, and if patient 1 is HIV-positive and has had two incidents, with charges of 50 and 90 dollars, then the arrays would look like this:

hiv	0	1	...

charges	100	75	20	50	90	...

start	0	3	5	...

We assume the typings `hiv` : H *arr* L, `charges` : M *arr* L, and `start` : M *arr* L.

Now, suppose we are interested in calculating the median charge for each of the HIV-positive patients and storing these medians into an array, `hivmedians` : H *arr* L. To do this, we might develop the program given in Figure 3.

The program uses a number of auxiliary variables, whose types are to be inferred. It is written straightforwardly, with the exception of the mysterious variable `lk`, which is declared to have type M. Does this program satisfy secure information flow? The reader may wish to think about this before reading further.

We may observe that the final value of `lk` is the length of the last `temp` array that gets allocated. As a result, `lk` ends up holding the number of incidents of the last *HIV-positive* patient. So if we search `start[]` in reverse order until we find an `i` such that `start[i+1]-start[i] = lk`, then we know that all patients after `i` are HIV-negative, and `i` is (perhaps) likely to be HIV-positive. Since `hiv` : H *arr* L and `lk` : M, we see that this program does not satisfy secure information flow.

When we do type inference on this program, type inference fails and we get the output shown in Figure 4. The first message reports that the assignment to `lk` is illegal, because

```
// hiv : H arr L, charges : M arr L
// start : M arr L, hivmedians : H arr L, lk : M
// i : ?, j : ?, k : ?, m : ?, n : ?,
// temp : ? arr ?, t : ?

k := 0;
i := 0;
while i < hiv.length do {
  if hiv[i] then {
    n := start[i+1] - start[i];
    allocate temp[n];
    // Copy patient i's charges into temp:
    j := 0;
    while j < n do {
      temp[j] := charges[i+j];
      j := j+1
    };
    // Sort temp using Bubble sort:
    j := 0;
    while j < n-1 do {
      m := 0;
      while m < n-1-j do {
        if temp[m+1] < temp[m] then {
          t := temp[m];
          temp[m] := temp[m+1];
          temp[m+1] := t
        }
        else {
          skip
        };
        m := m+1
      };
      j := j+1
    };
    // Store median into hivmedians:
    hivmedians[k] := temp[n/2];
    k := k+1
  }
  else {
    skip
  };
  lk := temp.length;
  i := i+1
}
```

Figure 3: Example HIV Program

```
tests/hiv:94.3: illegal explicit flow
  lk := temp.length;
  ^     ^
--left-hand side, lk, has type M by declaration
--right-hand side has type H because
  temp.length has type H because
      tests/hiv:61.6: guard of if statement
        if hiv[i] then {
           ^
      has type H because
        hiv[] has type H by declaration
    and
      tests/hiv:63.5: nested statement
          allocate temp[n];
          ^
      has implicit flow from H guard, forcing the
      type of temp.length to be raised to H
```

Figure 4: Output for HIV Program

the type of `temp.length` is H. The next two messages explain why—`temp` is allocated inside an `if` statement whose guard is H; as a result, the length of `temp` is raised to H.

Interestingly, if the two lines

```
n := start[i+1] - start[i];
allocate temp[n];
```

are moved up, so that they come immediately before `if hiv[i]`, then type inference succeeds, assigning type H *arr* M to `temp`. And notice that this change eliminates the leak from the program—now `lk` simply holds the number of incidents of the last patient, and this is M information.

7. RELATED WORK

Constraint-based type inference approaches have been used in a number of previous works on secure information flow. Volpano and Smith [12] describe sound and complete type inference for a simple sequential language with procedures. Procedures are polymorphic with respect to security types. In Jif [3, 4], a Java-like language, type inference is used to assign labels automatically to the methods of a class. Jif uses the algorithm in [7] to resolve constraints. To our knowledge, there are no formal proofs of the security properties of Jif or typing with type inference. Flow Caml [9, 5] is a language based on Objective Caml, enhanced with security annotations and analysis. Its type system and type inference have been formally proved to be correct. Moreover, its type inference engine [10] has been implemented as a separate module for general use. However, it does not attempt to localize type errors. Sun, Banerjee, and Naumann [11] discusses type inference for class libraries in a sophisticated Java-like language, in which security types for class fields and methods are inferred. Soundness and (limited) completeness are proved. A prototype is under development, but we are unaware of any discussion of error reporting in this system.

8. CONCLUSION

We propose a non-constraint-based type inference algorithm B, which leads to simple proofs, theorems, and accurate, effective error reporting. In future work, it would be interesting to extend B to handle a larger language, for example with procedures. Typing procedures monomorphically (with respect to security levels) would be straightforward. Allowing procedures to be polymorphic would be more challenging—a simple (but inefficient) way to achieve it would be to inline all procedure calls.

This work was partially supported by the National Science Foundation under grant HRD-0317692.

9. REFERENCES

[1] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *Proceedings 17th IEEE Computer Security Foundations Workshop*, pages 115–124, Pacific Grove, California, June 2004.

[2] F. Henglein and J. Rehof. The complexity of subtype entailment for simple types. In *Proc. Twelfth IEEE Symposium on Logic in Computer Science*, pages 352–361, 1997.

[3] A. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings 26th Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, Jan. 1999.

[4] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. *Jif: Java + information flow*. Cornell University, 2004. Available at `http://www.cs.cornell.edu/jif/`.

[5] F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.

[6] V. Pratt and J. Tiuryn. Satisfiability of inequalities in a poset. *Fundamenta Informaticae*, 28(1–2):165–182, 1996.

[7] J. Rehof and T. A. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2–3):191–221, 1999.

[8] A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[9] V. Simonet. Flow Caml in a nutshell. In G. Hutton, editor, *Proceedings of the first APPSEM-II workshop*, pages 152–165, Nottingham, United Kingdom, Mar. 2003.

[10] V. Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. In A. Ohori, editor, *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS'03)*, volume 2895 of *Lecture Notes in Computer Science*, pages 283–302, Beijing, China, Nov. 2003. Springer-Verlag.

[11] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In *Proc. Eleventh International Static Analysis Symposium (SAS)*, Verona, Italy, Aug. 2004.

[12] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621, Apr. 1997.

[13] J. Yang, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *Proc. 12th International Workshop on Implementation of Functional Languages*, Aachen, Germany, Sept. 2000.