6-22-2005

Practical secure information flow in programming languages

Zhenyue Deng
Florida International University

DOI: 10.25148/etd.FI14062241

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

PRACTICAL SECURE INFORMATION FLOW IN PROGRAMMING LANGUAGES

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Zhenyue Deng

2005

To: Interim Dean Mark D. Szuchman
College of Arts and Sciences

This dissertation, written by Zhenyue Deng, and entitled Practical Secure Information Flow in Programming Languages, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Yi Deng

Xudong He

Dev K. Roy

Geoffrey Smith, Major Professor

Date of Defense: June 22, 2005

The dissertation of Zhenyue Deng is approved.

Interim Dean Mark D. Szuchman
College of Arts and Sciences

Dean Douglas Wartzok
University Graduate School

Florida International University, 2005

ACKNOWLEDGMENTS

I would like to thank all my committee members for their support and comments. I would like to thank my major professor, Dr. Geoffrey Smith, for leading me into the exciting area of secure information flow, all his help, support and countless hours of stimulating talks. I would like to thank all the staff and faculty at the School of Computer Science for providing an excellent learning and research environment.

ABSTRACT OF THE DISSERTATION

PRACTICAL SECURE INFORMATION FLOW IN PROGRAMMING LANGUAGES

by

Zhenyue Deng

Florida International University, 2005

Miami, Florida

Professor Geoffrey Smith, Major Professor

If we classify variables in a program into various security levels, then a secure information flow analysis aims to verify statically that information in a program can flow only in ways consistent with the specified security levels. One well-studied approach is to formulate the rules of the secure information flow analysis as a type system. A major trend of recent research focuses on how to accommodate various sophisticated modern language features. However, this approach often leads to overly complicated and restrictive type systems, making them unfit for practical use. Also, problems essential to practical use, such as type inference and error reporting, have received little attention. This dissertation identified and solved major theoretical and practical hurdles to the application of secure information flow.

We adopted a minimalist approach to designing our language to ensure a simple lenient type system. We started out with a small simple imperative language and only added features that we deemed most important for practical use. One language feature we addressed is arrays. Due to the various leaking channels associated with array operations, arrays have received complicated and restrictive typing rules in other secure languages. We presented a novel approach for lenient array operations, which lead to simple and lenient typing of arrays.

Type inference is necessary because usually a user is only concerned with the security types for input/output variables of a program and would like to have all types for auxiliary variables inferred automatically. We presented a type inference algorithm B and proved its soundness and completeness. Moreover, algorithm B stays close to the program and the type system and therefore facilitates informative error reporting that is generated in a cascading fashion. Algorithm B and error reporting have been implemented and tested.

Lastly, we presented a novel framework for developing applications that ensure user information privacy. In this framework, core computations are defined as code modules that involve input/output data from multiple parties. Incrementally, secure flow policies are refined based on feedback from the type checking/inference. Core computations only interact with code modules from involved parties through well-defined interfaces. All code modules are digitally signed to ensure their authenticity and integrity.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

In the context of language-based secure information flow, variables in a program are associated with security levels. Information in variables of lower security levels are allowed to flow to variables of higher security levels, but not vice versa. We call all the rules that govern how information may flow the *secure policy.*

In an imperative language, we are mainly concerned with two channels of information flow: explicit flows and implicit flows. Explicit flows occur through direct assignments. For example, in assignment $hi := lo$, information in variable lo flows to variable hi. Implicit flows occur through *program context*, that is, whether or not a statement gets executed may reflect the value of some variables. For example, **if** $a > 0$ **then** $b := 1$ **else** $b := 0$, we can tell whether a is greater than 0 by examining the value of b after the execution of the statement.

Another channel for information flow is program termination. For example, if a program that contains this code segment:

```
a := b;
while ( a > 0 ) {}
```

does not terminate in a reasonable amount of time, we can say that it is likely that the program ends up in an infinite loop and variable a is greater than 0. Because non-termination requires close observation of the running of the program and is usually not something demonstrated by normal programs, we do not consider this channel in our study. We assume all programs terminate. We call our study *termination insensitive.*

The problem of secure information flow in programs was first studied in the 1970's by the Dennings [1]. Their original study had no formal definition of information flow and used syntactical analysis to check programs for flow violations. Their method of syntactical

analysis does not facilitate a soundness proof. The research area received little attention until the break-through work in 1996 by Volpano, Smith, and Irvine [2]. They used the notion of *noninterference* to establish formally what it means for a program to be considered free of secure information flow violations. Noninterference says that if we run a program side by side under any two memories that only differs in variables of higher security levels, after termination, the two memories should still agree on values in variables of lower security levels. In a novel approach, they used type systems to describe and enforce security policies, which led to the proof of the noninterference property. Since then, the research in the area has exploded. In a survey paper published in early 2003 by Sabelfeld and Myers [3], about 150 papers were cited.

Secure information flow has been studied in various language settings. Numerous works have been carried out in the context of functional languages [4, 5, 6, 7]. Secure information flow has also been studied in π-calculus [8, 9].

One of the major trends of active research in recent years studies how to accommodate various features of modern languages. For example, Volpano and Smith [10, 11] studied type systems for secure information flow in the setting of multi-threaded languages. Myers [12] developed JFlow (now known as Jif), an object-oriented Java-like language and its secure type system. Banerjee and Naumann [13] presented a Java-like language with proved security properties.

1.1 A Less-travelled Path - Making Secure Information Flow Practical

While all these research works in expanding the expressiveness of secure information flow languages are exciting and significant, the sophisticated language features lead to more complicated and often more restrictive type systems. For example, in JFlow [12], the type

system is so complicated that it is arguably beyond the grasp of ordinary programmers. Moreover, errors in secure information flow are often not localized due to implicit flows, making it harder to explain in the context of a more complicated type system. Some language features call for very restrictive type systems. For example, in the concurrent language proposed by Smith [11], **while** statements with variables of higher security levels in their guard expression cannot be followed by assignments to lower variables. This restriction surely will disqualify a great number of programs.

In this dissertation, we pursue a less-travelled direction - making secure inforamtion flow more practical. In order to be practical, we aim to achieve the following goals:

1. The language should be reasonably computationally powerful.
2. The type system should be simple enough for regular programmers.
3. The type system should be lenient to accept as many programs as possible.
4. Type inference should be available for typing all auxiliary variables.
5. Informative error reporting should lead the programmer quickly to the root of an error.

Complex language features tend to result in more complicated and restrictive type systems. To make our type system simple and lenient, we adopt a minimalist approach: We start out with the most basic language, and add language features that are most essential to computation. For this matter, we start with the secure language presented in [2], which features variable of integer data type, assignments, if-then-else and while statements. Its type system and security property proofs are simple and elegant. One missing language feature we identified as essential to computation is arrays. In chapter 2, we incorporate

arrays into the basic language and prove its security properties. We discuss the impact of adding other language features to our language in Chapter 5.

When writing programs in a secure language, usually the programmer is only concerned with the security levels for input/output variables. It would be ideal that the programmer only need to specify the types of variables he or she is interested in, leaving all others to be inferred automatically. In Chapter 4, we present a novel non-constraint-based type inference algorithm that infers security types for all unspecified variables.

Type systems for secure information flow are inherently more complex than data typings in a language like C, because the errors may not be localized due to implicit flows. When type inference fails, therefore, it is of vital importance to give the programmer informative error reporting that traces back to the root of the error. One of the major advantages of our type inference algorithm is that it facilitates informative error reporting. We have developed data structures and algorithms for displaying error traces in a cascading fashion.

We complete the picture of practical secure information flow in Chapter 5 with a novel framework for developing secure information flow applications. We identify the code module of an application that requires secure information flow as the *core computation*. Core computations communicate with other non-secure-information-flow code modules only through well-defined interfaces. We also demonstrate how to use error reporting as a guideline to relax security specifications to achieve a well typed program.

While the problem of secure information flow in and by itself merits theoretical study, it also has the potential real world impact on improving user privacy in computer systems. In today's world of computing, the user does not have any control over how a program handles his personal information. All security concerns are based on trust. Usually if the user trusts

company A, he lets a program from company A handle his personal information. Secure information flow techniques may change the situation by giving the user the guarantee of how his personal information may flow in the program. For example, in 2005, about 53% of the tax returns were filed online [14]. The IRS designated numerous companies to accept e-filings, which then forward electronic tax forms to the IRS. The whole scheme looks like this:

Figure 1: Today's Tax E-filing Scheme

The scheme involves three parties: the IRS, the internal revenue service; the user, which is the person filing a tax return, and TaxCom, which is a company that the IRS authorizes to accept e-filings. The user usually downloads an application from TaxCom or logs on to the website of TaxCom via a secure Internet connection. The tax return is prepared by the user answering a series of questions and entering his or her personal financial data. If a fee is charged for the e-filing service, the user also need to enter his or her tax preparation payment data such as credit card information. All this information is transmitted back to the TaxCom, where TaxCom retains the tax preparation payment information and forwards all the tax related personal financial data to the IRS [15].

There are two major drawbacks with this scheme. First, the user must have total trust in the website or the application that is handling his or her personal and financial information. Second, much information is only intended for the IRS, such as names of dependents, social security numbers and adjusted gross income. TaxCom should not be handling such information in the first place. The fact that TaxCom has access to this

information opens up many security vulnerabilities. For example, employees at TaxCom might browse private personal information and even divulge it. Hackers might break into TaxCom and steal personal information.

In addition, it is likely that the IRS adopted the model to distribute data load and encourage better service and efficiency through competition among the dozens of TaxCom's. However, in this area, the playground may not be leveled as the IRS would like. When people are filing their taxes electronically, it is very likely that the concern for privacy and security outweighs the concern for the quality of services and cost. So companies that can afford big advertising spending have an edge here: because privacy and security are based on trust in e-filing, people might believe that companies that can spend more on advertising would likely spend more on security equipment and procedures. Therefore, for a start-up company in the tax e-filing, it would be hard to attract customers even with better tax filing services, due to lack of reputations and possibly less marketing.

With security information flow techniques, instead of going by trust, the user can actually have the assurance of where his inforamtion might flow, and where his information won't flow. In the improved scheme, now instead of sending all information over to TaxCom, a user can download a tax preparation program from TaxCom to his or her local computer. The program is certified to ensure secure information flow. Now for all information the user provided to the program, the user can make sure it flows only to intended parties. In our tax example, the program would show that social security numbers only flow to the IRS, while credit card information for tax preparation payment only flows to TaxCom.

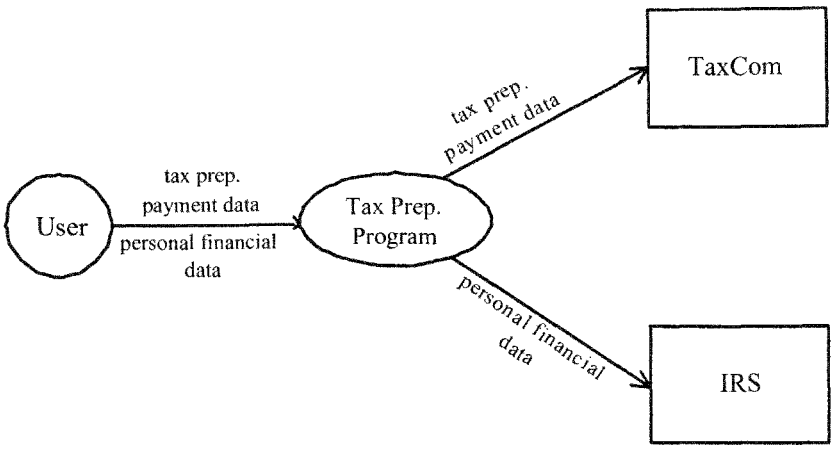


Figure 2: A Better Tax E-filing Scheme

1.2 Contributions

In this dissertation, our goal is to make secure information flow practical. We identify key hurdles to the application of secure information flow and propose solutions. We start out with a simple imperative language. We add arrays with lenient operations which lead to a simple type system and straightforward proofs of the security properties of our language. We present a novel type type inference algorithm that stays close to the type system and facilitates informative error reporting. We identify user information privacy protection as the area for application of secure information flow and present an application development framework. In the framework, code modules that ensure secure information flow are separated from supporting code modules by well-defined interfaces. An incremental process is proposed to refine secure flow policies and resolve errors based on feedback from type checking/inference.

2 Basic Language and its Security Properties

In this chapter, we describe the language in [2] as the foundation upon which our work is built. We also describe the use of lattices for specifying secure flow policies, the type system for the language and its secure information flow properties.

We pick the language in [2] as our basic language to expand and enhance through the latter chapters because the language is simple, with a simple and elegant type system and proved secure information flow properties.

2.1 Security Lattice

In literature of secure information flow, lattices are widely used to describe flow policies. Here are graphical representations of two lattices that describe flow policies.

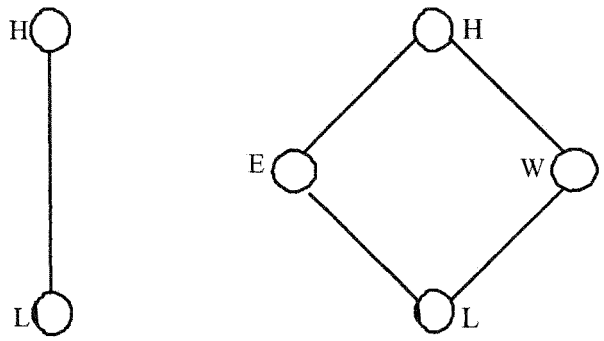


Figure 3: Two Sample Lattices

The left-hand side graph shows the simplest scenario of information flow. There are only two security levels in the lattice, L and H. Information is allowed to flow from L variables to H variables, but not vice versa. In the graph on the right-hand side, we have a diamond-shaped lattice with four security levels. Information is allowed to flow from lower security levels to higher security levels. Note that no information flow is allowed between E and W.

Formally, the definition of lattice is based on the the notion of partial order[16, page 2].

Definition 2.1.1 *Let P be a set. A partial order on P is a binary relation $\leq$ on P such that, for all x, y, z in P,*

1. $x \leq x$,
2. $x \leq y$ *and* $y \leq x$ *imply* $x = y$,
3. $x \leq y$ *and* $y \leq z$ *imply* $x \leq z$.

Conditions 1, 2 and 3 are referred to, respectively, as **reflexivity**, **anti-symmetry** and **transitivity**.

The ordering is partial rather than total, because there may exist elements x and y for which neither $x \leq y$ nor $y \leq x$. A set P equipped with a partial order is said to be an ordered set, written as $\langle P; \leq \rangle$.

A lattice is a partially ordered set in which all finite subsets have a least upper bound(join) and a greatest lower bound(meet). For example, for a subset of $\{x, y\}$, the join of x and y is denoted as $x \vee y$; the meet of x and y is denoted as $x \wedge y$. We use $\bot$ and $\top$ to denote the greatest lower bound and the least upper bound for a whole lattice.

The greatest lower bound and the least upper bound are defined as[16, page 28]:

Definition 2.1.2 *Let P be an ordered set and let $S \subseteq P$. An element $x \in P$ is an upper bound of S if $s \leq x$ for all $s \in S$. A lower bound is defined dually. The set of all upper bounds of S is denoted by S^u and the set of all lower bounds by S^l:*

$$S^u := \{x \in P | (\forall s \in S) s \leq x\} \text{ and } S^u := \{x \in P | (\forall s \in S) s \geq x\}.$$

We call x the **least upper bound** *of S if*

1. *x is an upper bound of S, and*

2. *$x \leq y$ for all upper bounds of y of S.*

Dually, if S^l has a largest element, then x is called the **greatest lower bound** *of S.*

Finally, a lattice is defined as[16, page 29]:

Definition 2.1.3 *Let P be a non-empty ordered set. If $x \vee y$ and $x \wedge y$ exist for all x and $y \in P$, then P is called a* **lattice**.

In our study, we assume all lattices are finite.

2.2 The Language and its Semantics

The syntax of the language is as follows:

$$
\begin{array}{lrcl}
(\textit{phrases}) & p & ::= & e \mid c \\
(\textit{expressions}) & e & ::= & x \mid n \mid \\
 & & & e_1/e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid \\
 & & & e_1 * e_2 \mid e_1 = e_2 \mid e_1 < e_2 \\
(\textit{commands}) & c & ::= & x := e \mid \\
 & & & \textbf{skip} \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & & \textbf{while } e \textbf{ do } c \mid \\
 & & & c_1 ; c_2
\end{array}
$$

Our starting language as in [2] is a simple language. It has integer literals(n), integer variables(x) and binary expressions. A language phrase can be either an expression or a command. Standard binary operators are included. Commands include assignment, **if**,

while, composition and **skip**. The **skip** command is used as a "no-op". The composition command is for composing programs.

Execution of commands is given by a standard structural operational semantics [17]:

(UPDATE) $$\frac{x \in dom(\mu)}{(x := e, \mu) \longrightarrow \mu[x := \mu(e)]}$$

(NO-OP) $$(\textbf{skip}, \mu) \longrightarrow \mu$$

(BRANCH) $$\frac{\mu(e) \neq 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_2, \mu)}$$

(LOOP) $$\frac{\mu(e) = 0}{(\textbf{while } e \textbf{ do } c, \mu) \longrightarrow \mu}$$

$$\frac{\mu(e) \neq 0}{(\textbf{while } e \textbf{ do } c, \mu) \longrightarrow (c; \textbf{while } e \textbf{ do } c, \mu)}$$

(SEQUENCE) $$\frac{(c_1, \mu) \longrightarrow \mu'}{(c_1; c_2, \mu) \longrightarrow (c_2, \mu')}$$

$$\frac{(c_1, \mu) \longrightarrow (c_1', \mu')}{(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu')}$$

The operational semantics is very similar to that of the C language. We use μ to denote the memory that maps identifiers to their values. We use $\mu(e)$ to denote the value of expression e in memory μ. For conditional statements **if** and **while**, integer 0 stands for the boolean value "false" while a non-zero integer stands for the boolean value "true". Since only integers are allowed, divisions are carried out according to the rule of the integer division.

2.3 Type System

One of the major contributions of [2] is the use of type systems as the static analysis method. A type system consists of a set of typing rules and axioms for deriving typing judgments. For our purpose, a judgment has the following form:

$$\gamma \vdash p : \rho$$

which says that under identifier typing γ, program phrase p has type ρ. identifier typing γ maps identifiers to their types. It gives all free variables in p their types. A typing rule usually consists of one or more judgments as hypotheses and one judgment as the conclusion. We say that the conclusion follows from the typing rules if all the hypotheses are met.

For example, consider a simple type system for integer expressions. We have the following typing rules:

(1) $\gamma \vdash n : int$

(2) $\gamma \vdash x : \tau,\ if\ \gamma(x) = \tau$

(3)
$$\frac{\gamma \vdash e : int \qquad \gamma \vdash e' : int}{\gamma \vdash e + e' : int}$$

Rule (1) is an axiom asserting that all integer literals have type *int*. Rule (2) is a judgment that says a variable as an expression will have whatever type the variable has. Rule (3) has two hypotheses and a conclusion that states that if sub-expression e and e' are both of type *int*, expression $e + e'$ is of type *int*.

To type expression $y + 1$, first 1 and y are typed as integer expressions by rules (1) and (2); then by rule (3), the whole expression is typed as integer.

Here are the types used by our type system:

$$\begin{array}{ll} (\textit{data types}) & \tau ::= l_1 \mid l_2 \mid \ldots \\ (\textit{phrase types}) & \rho ::= \tau \mid \tau\ \textit{var} \mid \tau\ \textit{cmd} \end{array}$$

Data types, denoted by τ, correspond to security levels in a security lattice. They are used as expression types. Variable types are denoted as $\tau\ \textit{var}$. A variable of type $\tau\ \textit{var}$ can be assigned to by an expression whose type is τ or below. Commands also have types denoted as $\tau\ \textit{cmd}$. Command types are used to track implicit flows. A command of type $\tau\ \textit{cmd}$ means that the command does not assign to any variable whose type is below τ. Here is a simple example that shows how to use the $\tau\ \textit{cmd}$ type to prevent illegal implicit flows. In the command **while** e **do** c, if e has type τ, then in command c, any assignment to variables of type below τ could cause an illegal implicit flow. However, if we make sure that c is typed as $\tau\ \textit{cmd}$, then we are sure that c does not assign to any variables whose type is below τ and therefore no illegal implicit flow will occur. The typing rules IF and WHILE show how to use command types to prevent illegal implicit flows. In typing rule WHILE the loop body c is required to have type $\tau\ \textit{cmd}$ or higher to prevent possible illegal flow from the τ type guard expression.

The $\leq$ relation of the lattice is carried over into the type system in terms of sub-typings, denoted as $\subseteq$. Subtyping among data types allows the normal upward flows in the program. More interestingly, the sub-typing rule for commands is *antimonotonic* (or *contravariant*), which means that if $\tau \subseteq \tau'$, then $\tau'\ \textit{cmd} \subseteq \tau\ \textit{cmd}$.

The typing rules and subtyping rules are given in Figures 4 and 5.

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

$$\text{(INT)}\quad \gamma \vdash n : \bot$$

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

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

$$\text{(SKIP)}\quad \gamma \vdash \textbf{skip} : \top\ cmd$$

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

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

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

Figure 4: Typing Rules for Our Basic Language

$$\text{(BASE)}\quad \bot \subseteq \top$$

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

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

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

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

Figure 5: Subtyping Rules

2.4 Security Properties

Basically, if we classify variables into various security levels, secure information flow is the problem of preventing information flow from variables of higher security levels into lower security levels. The notion is very direct and intuitive. It is desirable to formalize this notion into some provable properties of programs. Volpano and Smith [2] used *noninterference* to formally describe the property demonstrated by programs that satisfy secure information flow requirements. Noninterference says that for any security level τ, if we run a program under two memories that differ only in variables of type above τ, then after program termination, the two memories still agree on variable of type τ and below. In Chapter 3, we add arrays into our basic language and prove its secure information flow properties.

3 Arrays

In this chapter, we focus on the typing of array operations in a type system for secure information flow. Arrays are interesting because they play a major role in many nontrivial programs and because they can cause subtle information leaks, leading existing type systems to impose severe restrictions.

An example of a leak resulting from array indexing can be found in Denning's early work on secure information flow [1, page 509]. If array `a` is L and `secret` is H, then the assignment `a[secret] = 1;` is dangerous; if `a` is initially all zero, then after the assignment we can deduce the value of `secret` by searching `a` for a nonzero element:

```
a[secret] = 1;
i = 0;
while (i < a.length) {
  if (a[i] == 1)
    leak = i;
  i++;
}
```

Out-of-bounds array indices cause other problems. If array bounds checking is not performed (as in typical C implementations), then assignments to array elements can actually write outside the array, making it impossible to ensure any security properties whatsoever. But if out-of-bounds array indices lead to exceptions (as in Java), then statements sequentially following an array operation may not be reached, leading to possible information flows. For example, Figure 6 gives a Java program that leaks a 10-bit `secret` by turning on each

```
class Array {
  public static void main(String[] args) {
    int secret = Integer.parseInt(args[0]);
    int leak = 0;
    int [] a = new int[1];

    for (int bit = 0; bit < 10; bit++)
      try {
        a[1 - (secret >> bit) % 2)] = 1;
        leak |= (1 << bit);
      }
      catch (ArrayIndexOutOfBoundsException e) { }
    System.out.println("The secret is " + leak);
  }
}
```

Figure 6: A Leak Exploiting Out-of-bounds Array Indices

bit of `leak` following an array assignment that throws an exception if the corresponding bit of `secret` is 0.

Recent type systems for secure information flow have imposed a variety of restrictions to prevent leaks caused by array indexing. The simplest approach, adopted by Agat [18, page 45] (which aims to prevent *timing leaks*), is to require that all array indices and lengths be L. But this is of course very restrictive. In fact, just requiring that array indices be L already prevents something as basic as summing an array whose length is H:

```
sum = 0;

i = 0;

while (i < a.length) {

  sum = sum + a[i];

  i = i + 1;

}
```

Here, the `while` loop causes an *implicit flow* [1] from `a.length` to `i`, because the assignment `i = i + 1` is guarded by the condition `i < a.length`. Hence if `a.length` is H, then we must make `i` (and `sum`) be H as well, making `a[i]` illegal, if array indices must be L.

In Jif [12], a full-featured language for secure information flow, very complex rules are used to track information flows resulting from possible exceptions. In particular, subscripting an array with a H index causes the *program-counter label* pc to be raised to H, thereby preventing subsequent statements from assigning to L variables (until the potential `ArrayIndexOutOfBoundsException` is caught). It should be noted that the Jif type system has not, to our knowledge, been proved to ensure a noninterference property.

A similar strategy is described in Yocum's unpublished thesis [19]: an operation involving a H array index or an array of H length cannot be followed sequentially by any assignments to L variables, since those assignments will not be reached if there is an out-of-bounds index.

Because it is so disruptive to have to address the possibility of exceptions after every array operation involving a H index or length, we are led here to propose a lenient execution model in which programs *never* abort. The language *does* check for out-of-bounds indices, but

- an out-of-bounds array read simply yields 0, and
- an out-of-bounds array write is simply skipped.

This lenient execution model makes no difference on programs that are free of out-of-bounds array indices, though it does make debugging erroneous programs harder. (Of course, in this regard we are no worse off than in C!) But our focus here is on avoiding insecure information flows—we sacrifice exception reporting to L observers for the sake of a more permissive type system.

The lenient execution model can be used for other partial operations. For instance, we can say that division by 0 simply yields 0, thereby avoiding the need for restrictions like those proposed in [20]. This is also like Java's use of 32-bit two's compliment modular arithmetic, avoiding the need for integer overflow exceptions [21, page 156].

Because of our lenient execution model, we are able to use a simple and permissive type system. In our system, arrays are given types of the form τ_1 *arr* τ_2, where τ_1 is the security class of the array's contents and τ_2 is the security class of its length. Several combinations are useful: L *arr* L is a completely public array, H *arr* L is an array whose contents are private but whose length is public, and H *arr* H is a completely private array.

The rest of the chapter is organized as follows. In Section 3.1, we describe the simple sequential imperative language that we consider, and formally define its lenient semantics for array operations. In Section 3.2 we present the details of our type system, and in Section 3.3 we prove that it guarantees a noninterference property. In Section 4.5, we discuss the behavior of the type system on an example tax calculation program.

3.1 The Language and its Semantics

Programs are written in the simple imperative language [17], extended with one-dimensional integer arrays. The syntax of the language is as in Figure 7.

Here metavariable x ranges over identifiers and n over integer literals. The expression x.**length** yields the length of array x, as in Java. The command **allocate** $x[e]$ allocates a 0-initialized block of memory for array x; the size of the array is given by e. Note that for simplicity we do not treat arrays as first-class values. (First-class arrays would lead to issues of aliasing, which have been considered by Banerjee and Naumann [13].)

$$
\begin{array}{lrll}
(\textit{phrases}) & p & ::= & e \mid c \\
(\textit{expressions}) & e & ::= & x \mid n \mid x[e] \mid x.\textbf{length} \mid \\
 & & & e_1/e_2 \mid e_1 + e_2 \mid \\
 & & & e_1 * e_2 \mid e_1 = e_2 \mid e_1 - e_2 \\
(\textit{commands}) & c & ::= & x := e \mid \\
 & & & x[e_1] := e_2 \mid \\
 & & & \textbf{allocate } x[e] \mid \\
 & & & \textbf{skip} \mid \\
 & & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\
 & & & \textbf{while } e \textbf{ do } c \mid \\
 & & & c_1 ; c_2
\end{array}
$$

Figure 7: Language Syntax

A program c is executed under a memory μ, which maps identifiers to values. A value is either an integer n or an array of integers $\langle n_0, n_1, n_2, \ldots, n_{k-1} \rangle$, where $k \geq 0$. (Note that this simple memory model is not sufficient for modeling array aliasing—in Java, for example, two identifiers a and b can point to the same block of memory.)

We assume that expressions are evaluated atomically, with $\mu(e)$ denoting the value of expression e in memory μ. The formal semantics of array expressions and division is given in Figure 8. Note that the rules specify that an array read with an out-of-bounds index yields 0, as does division by 0.

Execution of commands is given by a standard structural operational semantics [17]. In addition, we have new rules for array writes and array allocation; these are given in Figure 9. These rules define a transition relation $\longrightarrow$ on *configurations*. A configuration is either a pair (c, μ) or simply a memory μ. In the first case, c is the command yet to be executed; in the second case, the command has terminated, yielding final memory μ.

$$\text{(ARR-READ)} \qquad \frac{x \in dom(\mu) \quad \mu(x) = \langle n_0, \ldots n_{k-1} \rangle, \;\; \mu(e) = i, \;\; 0 \leq i < k}{\mu(x[e]) = n_i}$$

$$\frac{x \in dom(\mu) \quad \mu(x) = \langle n_0, \ldots n_{k-1} \rangle, \;\; \mu(e) = i, \;\; (i < 0 \vee i \geq k)}{\mu(x[e]) = 0}$$

$$\text{(GET-LENGTH)} \qquad \frac{x \in dom(\mu), \;\; \mu(x) = \langle n_0, \ldots n_{k-1} \rangle}{\mu(x.\textbf{length}) = k}$$

$$\text{(DIV)} \qquad \frac{\mu(e_1) = n_1, \;\; \mu(e_2) = n_2, \;\; n_2 \neq 0}{\mu(e_1/e_2) = \lfloor n_1/n_2 \rfloor}$$

$$\frac{\mu(e_1) = n, \;\; \mu(e_2) = 0}{\mu(e_1/e_2) = 0}$$

Figure 8: Semantics of Array Expressions and Division

We write $\longrightarrow^k$ for the k-fold self composition of $\longrightarrow$, and $\longrightarrow^*$ for the reflexive, transitive closure of $\longrightarrow$.

3.2 The Type System

In this section, we extend the typing rules in [2] with new rules for typing array operations. We type arrays using types of the form τ_1 *arr* τ_2; here τ_1 describes the contents of the array, and τ_2 describes its length.

Here are the types used by our type system:

$$\begin{array}{lll} (\textit{data types}) & \tau & ::= \; l_1 \mid l_2 \mid \ldots \\ (\textit{phrase types}) & \rho & ::= \; \tau \mid \tau \; \textit{var} \mid \tau \; \textit{cmd} \mid \tau_1 \; \textit{arr} \; \tau_2 \end{array}$$

The l_i's are security levels drawn from a security lattice.

$$\text{(UPDATE-ARR)}\quad \frac{x \in dom(\mu),\ \ \mu(x) = \langle n_0, \ldots n_{k-1}\rangle,\ \ \mu(e_1) = i,\ \ 0 \le i < k,\ \ \mu(e_2) = n}{(x[e_1] := e_2, \mu) \longrightarrow \mu[x := \langle n_0, \ldots, n_{i-1}, n, n_{i+1}, \ldots, n_{k-1}\rangle]}$$

$$\frac{x \in dom(\mu),\ \ \mu(x) = \langle n_0, \ldots n_{k-1}\rangle,\ \ \mu(e_1) = i,\ \ (i < 0 \vee i \ge k)}{(x[e_1] := e_2, \mu) \longrightarrow \mu}$$

$$\text{(CALLOC)}\quad \frac{x \in dom(\mu),\ \ \mu(e) \ge 0}{(\textbf{allocate}\ x[e], \mu) \longrightarrow \mu[x := \langle \underbrace{0, 0, \ldots, 0}_{\mu(e)\ \text{of these}} \rangle]}$$

$$\frac{x \in dom(\mu),\ \ \mu(e) < 0}{(\textbf{allocate}\ x[e], \mu) \longrightarrow \mu[x := \langle\rangle]}$$

Figure 9: Semantics of Array Commands

For the array typing, the contents type dominates the length type, because the contents of an array implicitly includes the array's length. It makes no sense to have an array whose security type of its length is higher than its security type of contents. We therefore adopt the following constraint globally:

Global Array Constraint: In any array type τ_1 *arr* τ_2, we require that $\tau_2 \subseteq \tau_1$.

We now can present our type system formally. It allows us to prove *typing judgments* of the form $\gamma \vdash p : \rho$ as well as *subtyping judgments* of the form $\rho_1 \subseteq \rho_2$. Here γ denotes an *identifier typing*, which maps identifiers to phrase types of the form τ *var* or τ_1 *arr* τ_2. The typing rules are given in Figure 10. The subtyping rules are the same as those in our basic language as in Figure 5.

We briefly discuss the array typing rules. In rule SUBSCR, the value of expression $x[e]$ depends on the length and contents of array x as well as on the subscript e. For example, if $x[e]$ is nonzero, then we know that e is in range; that is, $0 \le e < x.\textbf{length}$. So if $x : \tau_1$ *arr* τ_2

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

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

Figure 10: Typing Rules

and $e : \tau_3$, then we need $x[e] : \tau_1 \vee \tau_2 \vee \tau_3$, where $\vee$ denotes join in the security lattice. Given the Global Array Constraint, this simplifies to $\tau_1 \vee \tau_3$.

Rule ASSIGN-ARR addresses similar issues. One interesting property of this rule is that, for example, if $x : H\ arr\ L$, then the command $x[e_1] := e_2$ can be given type $H\ cmd$, which says intuitively that it only assigns to H variables. This is valid because it does not change the L length of x.

In contrast, the command **allocate** $x[e]$ does assign a length to x, and this length can later be read by x.**length**. Hence, for example, if $x : \tau\ arr\ L$, then rule ALLOCATE gives **allocate** $x[e]$ type $L\ cmd$, to indicate that it (in effect) assigns to a L variable.

3.3 Properties of the Type System

In this section, we use relatively standard techniques to prove that our type system guarantees noninterference.

The proofs of some of the lemmas below are complicated somewhat by subtyping. We therefore assume, without loss of generality, that all typing derivations end with a single (perhaps trivial) use of rule SUBSUMP.

Lemma 3.3.1 (Subject Reduction) *If* $\gamma \vdash c : \tau\ cmd$ *and* $(c, \mu) \longrightarrow (c', \mu')$, *then* $\gamma \vdash c' : \tau\ cmd$.

Proof. By induction on the structure of c. There are just three kinds of commands that can take more than one step to terminate:

1. Case **if** e **then** c_1 **else** c_2.

   By our assumption, the typing derivation for c must end with a use of rule IF followed by a use of SUBSUMP:

$$\frac{\dfrac{\gamma \vdash e : \tau' \quad \gamma \vdash c_1 : \tau'\ cmd \quad \gamma \vdash c_2 : \tau'\ cmd}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau'\ cmd} \quad \tau'\ cmd \subseteq \tau\ cmd}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd}$$

Hence, by rule CMD$^-$, we must have $\tau \subseteq \tau'$. So by SUBSUMP we have $\gamma \vdash c_1 : \tau\ cmd$ and $\gamma \vdash c_2 : \tau\ cmd$. By semantic rule BRANCH, c' can be either c_1 or c_2; therefore, we have $\gamma \vdash c' : \tau\ cmd$.

2. Case **while** e **do** c_1.

   By our assumption, the typing derivation for c must end with a use of rule WHILE followed by a use of SUBSUMP:

$$\frac{\dfrac{\gamma \vdash e : \tau' \quad \gamma \vdash c_1 : \tau'\ cmd}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c_1 : \tau'\ cmd} \quad \tau'\ cmd \subseteq \tau\ cmd}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c_1 : \tau\ cmd}$$

   Hence, by rule CMD$^-$, we must have $\tau \subseteq \tau'$. By semantic rule LOOP, c' must be c_1 ; **while** e **do** c. So by rule SUBSUMP we have $\gamma \vdash c_1 : \tau\ cmd$. By typing rule COMPOSE, we have $\gamma \vdash c_1$; **while** e **do** $c_1 : \tau\ cmd$. Therefore, we have $\gamma \vdash c' : \tau\ cmd$.

3. Case $c_1; c_2$.

   By our assumption, the typing derivation for c must end with a use of rule COMPOSE followed by a use of SUBSUMP:

$$\dfrac{\dfrac{\gamma \vdash c_1 : \tau' \ cmd \quad \gamma \vdash c_2 : \tau' \ cmd}{\gamma \vdash c_1; c_2 : \tau' \ cmd} \quad \tau' \ cmd \subseteq \tau \ cmd}{\gamma \vdash c_1; c_2 : \tau \ cmd}$$

   Hence, by rule CMD$^-$, we must have $\tau \subseteq \tau'$. By SUBSUMP, we get $\gamma \vdash c_1 : \tau \ cmd$ and $\gamma \vdash c_2 : \tau \ cmd$. By semantic rule SEQUENCE, c' is either c_2 (if c_1 terminates in one step) or else $c_1'; c_2$, where $(c_1, \mu) \longrightarrow (c_1', \mu')$. For the first case, we have $\gamma \vdash c_2 : \tau \ cmd$. For the second case, we have $\gamma \vdash c_1' : \tau \ cmd$ by induction; hence $\gamma \vdash c_1'; c_2 : \tau \ cmd$ by rule COMPOSE.

□

Definition 3.3.1 *Memory μ is consistent with identifier typing γ, written $\mu : \gamma$, if $dom(\mu) = dom(\gamma)$ and, for every x, $\mu(x)$ is an integer n if $\gamma(x) = \tau \ var$ and $\mu(x)$ is an array of integers $\langle n_0, \ldots, n_{k-1} \rangle$ if $\gamma(x) = \tau_1 \ arr \ \tau_2$.*

Lemma 3.3.2 (Total Expressions) *If $\gamma \vdash e : \tau$ and $\mu : \gamma$, then $\mu(e)$ is a well-defined integer.*

Proof. By induction on the structure of e.

1. Cases x, n, $e_1 + e_2$, $e_1 - e_2$, $e_1 = e_2$.

   By induction, e_1 and e_2 are both well-defined integers. For these expressions, we have standard semantic rules. It is obvious that $\mu(e)$ is a well-defined integer.

2. Case e_1/e_2.

   By induction, e_1 and e_2 are both well-defined integers. By semantic rule DIV, if e_2 is 0, $\mu(e_1/e_2) = 0$; if e_2 is non-zero, e_1/e_2 is carried out as integer division, yielding a well-defined integer.

3. Case x.**length**.

   By semantic rule GET-LENGTH, the actual size of the array is returned as an integer.

4. Case $x[e_1]$.

   By induction, e_1 is a well-defined integer. By semantic rule ARR-READ, if e_1 is out of bounds of array x, $\mu(x[e_1]) = 0$; otherwise, the corresponding array element is returned as $\mu(x[e_1])$.

□

Lemma 3.3.3 (Progress) *If $\gamma \vdash c : \tau$ cmd and $\mu : \gamma$, then there is a unique configuration C, of the form (c', μ') or just μ', such that $(c, \mu) \longrightarrow C$ and $\mu' : \gamma$.*

Proof. By induction on the structure of c. It follows from the semantic rules. □

From the Subject Reduction and Progress lemmas, it follows that if command c is well typed under γ and c is executed in a memory μ consistent with γ, then the execution either terminates successfully or else loops—it cannot get stuck.

We also need a lemma about the execution of a sequential composition:

Lemma 3.3.4 *If $(c_1; c_2, \mu) \longrightarrow^j \mu'$, then there exist k and μ'' such that $0 < k < j$, $(c_1, \mu) \longrightarrow^k \mu''$, and $(c_2, \mu'') \longrightarrow^{j-k} \mu'$.*

Proof. By induction on j. If the derivation begins with an application of the first SEQUENCE rule, then there exists μ'' such that $(c_1, \mu) \longrightarrow \mu''$ and

$$(c_1; c_2, \mu) \longrightarrow (c_2, \mu'') \longrightarrow^{j-1} \mu'.$$

So we can let $k = 1$. And, since $j - 1 \geq 1$, we have $k < j$.

If the derivation begins with an application of the second SEQUENCE rule, then there exists c_1' and μ_1 such that $(c_1, \mu) \longrightarrow (c_1', \mu_1)$ and

$$(c_1; c_2, \mu) \longrightarrow (c_1'; c_2, \mu_1) \longrightarrow^{j-1} \mu'.$$

By induction, there exists k and μ'' such that $0 < k < j - 1$, $(c_1', \mu_1) \longrightarrow^k \mu''$, and $(c_2, \mu'') \longrightarrow^{j-1-k} \mu'$. Hence $(c_1, \mu) \longrightarrow^{k+1} \mu''$ and $(c_2, \mu'') \longrightarrow^{j-(k+1)} \mu'$. And $0 < k + 1 < j$.

□

Now we are ready to show that our type system ensures a noninterference property for well-typed commands c. Noninterference says that for some security level τ, changing the initial values of variables of type τ and above, cannot affect the final values of variables of type below τ. (Note however that under typing rule WHILE, changing the initial values of variables of type τ and above *can* affect the termination of c.)

Definition 3.3.2 *Memories μ and ν are equivalent up to τ, written $\mu \sim_\tau \nu$, if*

- *μ and ν are both consistent with γ,*
- *μ and ν agree on all variables of type τ_1 var, where $\tau_1 \leq \tau$,*
- *μ and ν agree on all arrays of type τ_1 arr τ_2, where $\tau_1 \leq \tau$ and $\tau_2 \leq \tau$, and*

- μ *and* ν *agree on the length (but not necessarily on the contents) of all arrays of type* τ_1 *arr* τ_2, *where* $\tau_1 \not\leq \tau$ *and* $\tau_2 \leq \tau$.

Lemma 3.3.5 (Simple Security) *If* $\gamma \vdash e : \tau$ *and* $\mu \sim_\tau \nu$, *then* $\mu(e) = \nu(e)$.

Proof. By induction on the structure of e:

1. Case x.

   By our assumption, the type derivation for x must end with a use of rule R-VAL followed by a use of SUBSUMP.

$$\frac{\dfrac{\gamma(x) = \tau' \ var}{\gamma \vdash x : \tau'} \qquad \tau' \subseteq \tau}{\gamma \vdash x : \tau}$$

   Because $\tau' \subseteq \tau$ and $\mu \sim_\tau \nu$, we have $\mu(x) = \nu(x)$.

2. Case $x[e_1]$.

   By our assumption, the type derivation for $x[e]$ must end with a use of rule SUBSCR followed by a use of SUBSUMP.

$$\frac{\dfrac{\gamma(x) = \tau_1 \ arr \ \tau_2, \quad \gamma \vdash e_1 : \tau_1,}{\gamma \vdash x[e] : \tau_1} \qquad \tau_1 \subseteq \tau}{\gamma \vdash x[e_1] : \tau}$$

   $\tau_1 \subseteq \tau$, $\tau_2 \subseteq \tau$, $\mu \sim_\tau \nu$, therefore by induction, $\mu(e) = \nu(e)$, and $\mu(x) = \nu(x)$.

3. Case x.**length**.

   By our assumption, the type derivation for x.**length** must end with a use of rule LENGTH followed by a use of SUBSUMP.

$$\dfrac{\dfrac{\gamma(x) = \tau_1 \; arr \; \tau_2}{\gamma \vdash x.\mathbf{length} : \tau_2} \quad \tau_2 \subseteq \tau}{\gamma \vdash x.\mathbf{length} : \tau}$$

4. Case e_1/e_2.

   By our assumption, the type derivation for e_1/e_2 must end with a use of rule QUOTIENT followed by a use of SUBSUMP.

$$\dfrac{\dfrac{\gamma \vdash e_1 : \tau', \;\; \gamma \vdash e_2 : \tau'}{\gamma \vdash e_1/e_2 : \tau'} \quad \tau' \subseteq \tau}{\gamma \vdash e_1/e_2 : \tau}$$

   By typing rule SUBSUMP, we have $\gamma \vdash e_1 : \tau$ and $\gamma \vdash e_2 : \tau$. By induction, we have $\mu(e_1) = \nu(e_1)$ and $\mu(e_2) = \nu(e_2)$. Then, by semantic rule DIV, we have $\mu(e_1/e_2) = \nu(e_1/e_2)$.

5. Cases $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, $e_1 = e_2$ are similar.

□

Lemma 3.3.6 (Confinement) *If* $\gamma \vdash c : \tau'$ *cmd and* $(c, \mu) \longrightarrow (c', \mu')$ *(or* $(c, \mu) \longrightarrow \mu'$*), then for any* τ *such that* $\tau' \not\subseteq \tau$, $\mu \sim_\tau \mu'$.

Proof. By induction on the structure of c:

1. Case $x := e$.

   By our assumption, the typing derivation for c must end with a use of rule UPDATE followed by a use of SUBSUMP:

$$\dfrac{\dfrac{\gamma(x) = \tau''\ var \quad \gamma \vdash e : \tau''}{\gamma \vdash x := e : \tau''\ cmd} \quad \tau''\ cmd \subseteq \tau'\ cmd}{\gamma \vdash x := e : \tau'\ cmd}$$

   Hence, by rule CMD$^-$, we must have $\tau' \subseteq \tau''$. Since $\tau' \not\subseteq \tau$, we have $\tau'' \not\subseteq \tau$. By semantic rule UPDATE, x is the only variable being assigned to in command c and $\gamma(x) = \tau''\ var$, so $\mu \sim_\tau \mu'$.

2. Case $x[e_1] := e_2$.

   By our assumption, the typing derivation for c must end with a use of rule ASSIGN-ARR followed by a use of SUBSUMP:

$$\dfrac{\dfrac{\gamma(x) = \tau_1\ arr\ \tau_2, \quad \gamma \vdash e_1 : \tau_1, \quad \gamma \vdash e_2 : \tau_1}{\gamma \vdash x[e_1] := e_2 : \tau_1\ cmd} \quad \tau_1\ cmd \subseteq \tau'\ cmd}{\gamma \vdash x[e_1] := e_2 : \tau'\ cmd}$$

   Hence, by rule CMD$^-$, we must have $\tau' \subseteq \tau_1$. Since $\tau' \not\subseteq \tau$, we have $\tau_1 \not\subseteq \tau$. By semantic rule UPDATE-ARR, command c may only assign to the contents of array x, whose type is $\tau_1\ arr\ \tau_2$, so $\mu \sim_\tau \mu'$.

3. Case **allocate** $x[e]$.

   By our assumption, the typing derivation for c must end with a use of rule ALLOCATE followed by a use of SUBSUMP:

$$\dfrac{\dfrac{\gamma(x) = \tau_1 \ arr \ \tau_2, \ \ \gamma \vdash e : \tau_2}{\gamma \vdash \textbf{allocate} \ x[e] : \tau_2 \ cmd} \qquad \tau_2 \ cmd \subseteq \tau' \ cmd}{\gamma \vdash \textbf{allocate} \ x[e] : \tau' \ cmd}$$

   Hence, by rule CMD$^-$, we must have $\tau' \subseteq \tau_2$. Since $\tau' \not\subseteq \tau$, we have $\tau_2 \not\subseteq \tau$. By semantic rule CALLOC, command c may only assign to the length of array x, whose type is $\tau_1 \ arr \ \tau_2$, so $\mu \sim_\tau \mu'$.

4. Cases **skip**, **if** e **then** c_1 **else** c_2, and **while** e **do** c.

   These cases are trivial, because $\mu = \mu'$.

5. Case $c_1; c_2$.

   By our assumption, the typing derivation for c must end with a use of rule COMPOSE followed by a use of SUBSUMP:

$$\dfrac{\dfrac{\gamma \vdash c_1 : \tau'' \ cmd \qquad \gamma \vdash c_2 : \tau'' \ cmd}{\gamma \vdash c_1; c_2 \ : \ \tau'' \ cmd} \qquad \tau'' \ cmd \subseteq \tau' \ cmd}{\gamma \vdash c_1; c_2 : \tau' \ cmd}$$

Hence, by rule CMD$^-$, we must have $\tau' \subseteq \tau''$. By SUBSUMP, $\gamma \vdash c_1 : \tau'\ cmd$. By semantic rule SEQUENCE, $(c_1, \mu) \longrightarrow (c_1', \mu')$ (or $(c_1, \mu) \longrightarrow \mu'$). So, by induction, we have $\mu \sim_\tau \mu'$. Therefore, either $(c_1; c_2, \mu) \longrightarrow (c_1'; c2, \mu')$ or $(c_1; c_2, \mu) \longrightarrow (c_2; \mu')$, we have $\mu \sim_\tau \mu'$.

□

Corollary 3.3.7 *If $\gamma \vdash c : \tau'\ cmd$ and $(c, \mu) \longrightarrow^* \mu'$, then for any τ such that $\tau' \not\subseteq \tau$, $\mu \sim_\tau \mu'$.*

Proof. By induction on the length of the execution of $(c; \mu) \longrightarrow^* \mu'$. We consider the difference forms of c :

1. Cases $x := e$, **skip**, $c_1; c_2$, x.**length**, $x[e_1] := e_2$, **allocate** $x[e]$.

   Follows immediately from Lemma 3.3.6(Confinement).

2. Case **while** e **do** c_1.

   By semantic rule LOOP, if $\mu(e) \neq 0$ then execution have the form:

   $$(\textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow (c_1; \textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow^* \mu'$$

   By induction, $\mu \sim_\tau \mu'$.

   If $\mu(e) = 0$ then execution have the form:

   $$(\textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow \mu$$

   $\mu = \mu'$, so $\mu \sim_\tau \mu'$.

3. Case **if** e **then** c_1 **else** c_2.

   By semantic rule BRANCH, if $\mu(e) \neq 0$ then execution have the form:

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu) \longrightarrow^* \mu'$$

   By induction, $\mu \sim_\tau \mu'$. Similarly, if $\mu(e) = 0$ then execution have the form:

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_2, \mu) \longrightarrow^* \mu'$$

   By induction, $\mu \sim_\tau \mu'$.

□

Theorem 3.3.8 (Noninterference) *Suppose that command c is well typed under γ and memories $\mu \sim_\tau \nu$. If $(c, \mu) \longrightarrow^* \mu'$ and $(c, \nu) \longrightarrow^* \nu'$, then $\mu' \sim_\tau \nu'$.*

Proof. By induction on the length of the execution $(c, \mu) \longrightarrow^* \mu'$. We consider the different forms of c:

1. Case $x := e$.

   Command c is well typed, so there must exist τ' and by typing rule ASSIGN :

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

So if $\tau' \not\subseteq \tau$, since $\gamma \vdash x := e \ : \tau' \ cmd$ and $(c, \mu) \longrightarrow \mu'$, by Lemma 3.3.6(Confinement), we have $\mu \sim_\tau \mu'$. Similarly, since $\gamma \vdash x := e \ : \tau' \ cmd$ and $(c, \nu) \longrightarrow \nu'$, by Lemma 3.3.6(Confinement), we have $\nu \sim_\tau \nu'$. Therefore, $\mu' \sim_\tau \nu'$.

Or else $\tau' \subseteq \tau$, since $\gamma \vdash e : \tau$ and $\mu \sim_\tau \nu$, by Lemma 3.3.5(Simple Security), we have $\mu(e) = \nu(e)$. Therefore, by semantic rule UPDATE, $\mu[x := \mu(e)] \sim_\tau \nu[x := \nu(e)]$.

2. Case **skip**.

   The result follows immediately from semantic rule NO-OP.

3. Case $c_1; c_2$.

   If $(c_1; c_2, \mu) \longrightarrow^j \mu'$ then by Lemma 3.3.4 there exist k and μ'' such that $0 < k < j$, $(c_1, \mu) \longrightarrow^k \mu''$ and $(c_2, \mu'') \longrightarrow^{j-k} \mu'$.

   Similarly, if $(c_1; c_2, \nu) \longrightarrow^{j'} \nu'$ then there exist k' and ν'' such that $0 < k' < j'$, $(c_1, \nu) \longrightarrow^{k'} \nu''$ and $(c_2, \nu'') \longrightarrow^{j'-k'} \nu'$.

   By induction, $\mu'' \sim_\tau \nu''$. So by induction again, $\mu' \sim_\tau \nu'$.

4. Case **if** e **then** c_1 **else** c_2.

   Command c is well typed, so there must exist τ' and by typing rule IF :

$$\frac{\gamma \vdash e : \tau' \qquad \gamma \vdash c_1 : \tau' \ cmd \qquad \gamma \vdash c_2 : \tau' \ cmd}{\gamma \vdash \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \ : \tau' \ cmd}$$

   So if $\tau' \not\subseteq \tau$, since $\gamma \vdash c \ : \tau' \ cmd$ and $(c, \mu) \longrightarrow^* \mu'$, by Corollary 3.3.7, we have $\mu \sim_\tau \mu'$. Similarly, since $\gamma \vdash c \ : \tau' \ cmd$ and $(c, \nu) \longrightarrow^* \nu'$, by Corollary 3.3.7, we have $\nu \sim_\tau \nu'$.

Therefore, $\mu \sim_\tau \mu'$. Or else $\tau' \subseteq \tau$, since $\gamma \vdash e : \tau$ and $\mu \sim_\tau \nu$, by Lemma 3.3.5(Simple Security), we have $\mu(e) = v(e)$. By semantic rule BRANCH, if $\mu(e) \neq 0$ then the two execution have the form:

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_1, \mu) \longrightarrow^* \mu'$$

and

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \nu) \longrightarrow (c_1, \nu) \longrightarrow^* \nu'.$$

By induction, $\mu' \sim_\tau \nu'$.

If $\mu(e) = 0$ then the two execution have the form:

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \mu) \longrightarrow (c_2, \mu) \longrightarrow^* \mu'$$

and

$$(\textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2, \nu) \longrightarrow (c_2, \nu) \longrightarrow^* \nu'.$$

By induction, $\mu' \sim_\tau \nu'$.

5. Case **while** e **do** c_1.

Command c is well typed, so there must exist τ' and by typing rule WHILE :

$$\frac{\gamma \vdash e : \tau' \qquad \gamma \vdash c_1 : \tau'\ cmd}{\gamma \vdash \textbf{while } e \textbf{ do } c_1 \ : \tau'\ cmd}$$

So if $\tau' \not\subseteq \tau$, since $\gamma \vdash c \ : \tau'\ cmd$ and $(c, \mu) \longrightarrow^* \mu'$, by Corollary 3.3.7, we have $\mu \sim_\tau \mu'$.

Similarly, since $\gamma \vdash c : \tau' \ cmd$ and $(c, \nu) \longrightarrow^* \nu'$, by Corollary 3.3.7, we have $\nu \sim_\tau \nu'$. Therefore, $\mu \sim_\tau \mu'$.

Or else $\tau' \subseteq \tau$, since $\gamma \vdash e : \tau$ and $\mu \sim_\tau \nu$, by Lemma 3.3.5(Simple Security), we have $\mu(e) = v(e)$. By semantic rule LOOP, if $\mu(e) \neq 0$ then the two execution have the form:

$$(\textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow (c_1; \textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow^* \mu'$$

and

$$(\textbf{while } e \textbf{ do } c_1, \nu) \longrightarrow (c_1; \textbf{while } e \textbf{ do } c_1, \nu) \longrightarrow^* \nu'.$$

By induction, $\mu' \sim_\tau \nu'$.

If $\mu(e) = 0$ then the two execution have the form:

$$(\textbf{while } e \textbf{ do } c_1, \mu) \longrightarrow \mu$$

and

$$(\textbf{while } e \textbf{ do } c_1, \nu) \longrightarrow \nu$$

Since $\mu' = \mu$ and $\nu' = \nu$, we have $\mu' \sim_\tau \nu'$.

6. Case **allocate** $x[e]$.

Command c is well typed, so there must exist τ_2 and by typing rule ALLOCATE :

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

So if $\tau_2 \not\subseteq \tau$, since $\gamma \vdash c : \tau_2\ cmd$ and $(c, \mu) \longrightarrow \mu'$, by Lemma 3.3.6(Confinement), we have $\mu \sim_\tau \mu'$. Similarly, since $\gamma \vdash c : \tau_2\ cmd$ and $(c, \nu) \longrightarrow \nu'$, by Lemma 3.3.6(Confinement), we have $\nu \sim_\tau \nu'$. Therefore, $\mu' \sim_\tau \nu'$.

Or else $\tau_2 \subseteq \tau$, since $\gamma \vdash e : \tau$ and $\mu \sim_\tau \nu$, by Lemma 3.3.5(Simple Security), we have $\mu(e) = \nu(e)$. By semantic rule CALLOC, if $\mu(e) \geq 0$, $\mu(e)$ memory cells are allocated to array x, all initialized to 0, in both μ and ν. By Global Constraint for arrays, if $\tau_2 \subseteq \tau_1 \subseteq \tau$, arrays $\mu(x)$ and $\nu(x)$ agree on both the τ contents and the τ lengths, therefore, $\mu' \sim_\tau \nu'$. If $\tau_1 \not\subseteq \tau$, two arrays $\mu(x)$ and $\nu(x)$ agree on the τ lengths, therefore, $\mu' \sim_\tau \nu'$.

If $\mu(e) < 0$, by semantic rule CALLOC, arrays $\mu(x)$ and $\nu(x)$ are reset to zero length in both μ' and ν', By global constraint for arrays, if $\tau_2 \subseteq \tau_1 \subseteq \tau$, arrays $\mu(x)$ and $\nu(x)$ agree on both the τ contents (vacuously) and the τ lengths, therefore, $\mu' \sim_\tau \nu'$. If $\tau_1 \not\subseteq \tau$, two arrays $\mu'(x)$ and $\nu'(x)$ agree on the τ lengths, therefore, $\mu' \sim_\tau \nu'$.

7. Case $x[e_1] := e_2$.

Command c is well typed, so there must exist τ_1 and by typing rule ASSIGN-ARR:

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

So if $\tau_1 \not\subseteq \tau$, since $\gamma \vdash c : \tau_1\ cmd$ and $(c, \mu) \longrightarrow \mu'$, by Lemma 3.3.6(Confinement), we have $\mu \sim_\tau \mu'$. Similarly, since $\gamma \vdash c : \tau_1\ cmd$ and $(c, \nu) \longrightarrow \nu'$, by Lemma 3.3.6(Confinement), we have $\nu \sim_\tau \nu'$. Therefore, $\mu' \sim_\tau \nu'$.

Or else $\tau_1 \subseteq \tau$. Since $\gamma \vdash e_1 : \tau$, $\gamma \vdash e_2 : \tau$ and $\mu \sim_\tau \nu$, by Lemma 3.3.5(Simple Security), we have $\mu(e_1) = \nu(e_1)$ and $\mu(e_2) = \nu(e_2)$.

By Global Constraint, arrays $\mu(x)$ and $\nu(x)$ agree on both the contents and lengths. By semantic rule UPDATE-ARR, if $\mu(e_1)$ is within the bounds of array x, arrays $\mu'(x)$ and $\nu'(x)$ agree on both the contents and lengths. So $\mu' \sim_\tau \nu'$.

If $\mu(e_1)$ is out of the bounds of array x, no change will be made to memories. $\mu = \mu'$, $\nu = \nu'$, therefore, $\mu' \sim_\tau \nu'$.

□

3.4 An Example Tax Calculation Program

We now try to get a sense of the practicality of our type system by considering its behavior on an example tax calculation program. Suppose that we are calculating income taxes, using a tax table like the following:

Taxable income		Income tax	
At least	Less than	Single	Married
...	...	...	...
25,200	25,250	3,434	3,084
25,250	25,300	3,441	3,091
25,300	25,350	3,449	3,099
...	...	...	...

In a richer language, we would likely represent the tax table as an array of records; here we use three parallel arrays instead:

`brackets`

...	25,200	25,250	25,300	...

`singleTaxTable`

...	3,434	3,441	3,449	...

`marriedTaxTable`

...	3,084	3,091	3,099	...

Given these tables, we can calculate the income tax for taxable income t by using binary search to find an index b such that

$$\texttt{brackets}[b] \leq t < \texttt{brackets}[b+1]$$

and then returning either $\texttt{singleTaxTable}[b]$ or $\texttt{marriedTaxTable}[b]$, depending on the marital status.

With respect to our type system, we want the typings $\texttt{brackets} : L\ arr\ L$, $\texttt{singleTaxTable} : L\ arr\ L$, and $\texttt{marriedTaxTable} : L\ arr\ L$, since the tax table is public information.

Let us further specify that we wish to calculate the income taxes for many tax returns. We represent the tax returns using two parallel arrays, `taxableIncome` and `maritalStatus`, using 0 to represent "single" and 1 to represent "married". We choose the typings $\texttt{taxableIncome} : H\ arr\ L$ and $\texttt{maritalStatus} : L\ arr\ L$ to indicate that taxable income is private, marital status is public, and the number of tax returns to be processed is public.

Our goal is to fill in an array `incomeTax` with the tax owed for each tax return. We also wish to compute `singleReturns` and `marriedReturns`, which count the number of single tax returns and married tax returns, respectively. The typings that we want for our outputs are `incomeTax` : H *arr* L, `singleReturns` : L, and `marriedReturns` : L.

Given this specification, we would naturally write a program like the one in Figure 11. Notice that the program makes use of four auxiliary variables: `i`, `lo`, `hi`, and `mid`.

Now we wish to see whether the program is accepted by our type system. To do this, we must figure out whether there are any acceptable types for the auxiliary variables. We begin by observing that the `if` command within the binary search has a H guard, since `taxableIncome[i]` is H. Hence the branches must not assign to L variables. This implies that `hi` : H and `lo` : H. And then the assignment `mid := (lo + hi) / 2` implies that `mid` : H as well. Finally, the last `if` command assigns to the L variables `singleReturns` and `marriedReturns`. As a result, its guard must be L, which implies that `i` : L.

With these typings for the auxiliary variables, it is straightforward to verify that the tax calculation program is well typed under our type system. We find it rather encouraging that we are able to write this program in a natural way and still have it accepted by the type system.

In contrast, we can observe that other approaches to typing arrays for secure information flow would run into trouble on this program. If we follow Agat's approach [18] and require that array indices be L, then the program seems hopeless, because it uses the H variables `lo` and `mid` as indices. If we follow Jif's approach [12] and disallow assignments to L variables after array operations that might fail due to H variables, then we cannot follow the reference to `brackets[mid]` with assignments to the L variables `singleReturns` and

```
// Tax calculation program.
//
// Inputs:
//    taxableIncome : H arr L
//    maritalStatus : L arr L
//    brackets : L arr L
//    singleTaxTable : L arr L
//    marriedTaxTable : L arr L
// Outputs:
//    incomeTax : H arr L
//    singleReturns : L
//    marriedReturns : L
// Auxiliary variables:
//    i, lo, hi, mid


allocate incomeTax[taxableIncome.length];
singleReturns := 0;
marriedReturns := 0;
i := 0;

while i < taxableIncome.length do (
  lo := 0;
  hi := brackets.length;
  while lo+1 < hi do (
    mid := (lo + hi) / 2;
    if taxableIncome[i] < brackets[mid] then
      hi := mid
    else
      lo := mid
  );

  if maritalStatus[i] = 0 then (
    incomeTax[i] := singleTaxTable[lo];
    singleReturns := singleReturns + 1
  )
  else (
    incomeTax[i] := marriedTaxTable[lo];
    marriedReturns := marriedReturns + 1
  )

  i := i+1
)
```

Figure 11: Tax Calculation Program

`marriedReturns`. It does seem that we could rewrite the program to satisfy Jif's typing rules—it appears that we could move the calculation of the L variables `singleReturns` and `marriedReturns` to the beginning of the program, before the dangerous array operations. This has the disadvantage of requiring some duplication of work; for example, it would require two passes through the `maritalStatus` array. More seriously, it is unclear whether this sort of transformation would always be possible, especially if the lattice of security classes is not a total order.

Alternatively, it seems that we could satisfy Jif's typing rules by wrapping each array operation in a tight `try-catch` block as shown below:

```
try {
  if (taxableIncome[i] < brackets[mid])
    hi = mid;
  else
    lo = mid;
}
catch (ArrayIndexOutOfBoundsException e){}
```

This technique allows us to achieve an approximation to our lenient semantics within Java, though at the cost of some syntactic clumsiness.

3.5 Conclusion

Because of our lenient execution model and our array types of the form τ_1 *arr* τ_2, we are able to do secure information flow analysis on interesting programs, using simple and permissive typing rules. The simplicity of our rules makes it straightforward to prove

that our type system ensures noninterference. Our tax calculation example suggests that interesting programs satisfying our typing rules can be written in a simple and natural way.

An earlier version of the work in this chapter was published in the proceedings of the 17th IEEE Security Foundations Workshop [22]. There we assumed a simple lattice of H and L instead of the general lattices used in this chapter. At the time of the publication, we had not found detailed information on how arrays are handled in Flow Caml. Afterwards, we received email correspondence from Vincent Simonet, the main author of Flow Caml. He noted that in their effort to overcome restrictions associated with array operations, they adopted a semantics that specifies that out-of-bounds array operations cause the program to abort. This is similar in effect to our lenient semantics.

In our tax calculation program, we see that it takes some pretty hard reasoning to figure out the types for auxiliary variable *lo*, *hi* and *mid* to have the whole program well typed. The example is just a relatively short program. It would become impossible to type a large program with many auxiliary variables by this simple reasoning and try-and-error approach. Therefore, it is very helpful to have all the types for auxiliary variables inferred automatically. We present a type inference algorithm for our language in the next chapter.

4 Type Inference and Error Reporting

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 [23, 12, 24]. 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 [25].

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 [26].

These challenges are evident in both Flow Caml [27] and Jif [28], currently the two most sophisticated implementations of languages with secure information flow. While Flow Caml performs very sophisticated polymorphic type inference, it currently makes no effort to report the source of type errors. If a program fails to type check, Flow Caml simply reports a message like

```
This expression generates the following information flow:
  !high < !low
which is not legal.
```

giving no indication of the location of the error. Jif is more ambitious in its error reporting, trying to find which constraint to "blame" if a system of constraints is unsolvable, but it remains difficult for users to track down the source of errors.

For example,

> Jif suffers quite heavily from the problem of confusing error messages that plague many languages that perform constraint solving as part of type checking. The problem is that as the compiler solves constraints, a failure may occur due to a constraint that was introduced in a context distant from where the error is actually reported. The constraint failure could also be the result of a number of interacting annotations, making the cause of the problem difficult to pin down. [29, p. 12]

In this work, we propose a novel, non-constraint-based type inference approach. We develop our approach for the simple imperative language with arrays that is considered in [22], but we expect that similar languages could be accommodated without difficulty. Our approach 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.

4.1 Discovery of the B Algorithm

To search for an inference algorithm that is closely based upon the type system, we assume that all the possible types of an inferable variable form an interval and study how typing rules affect the intervals for the inferable variables. For example, we consider an assignment to an inferable variable:

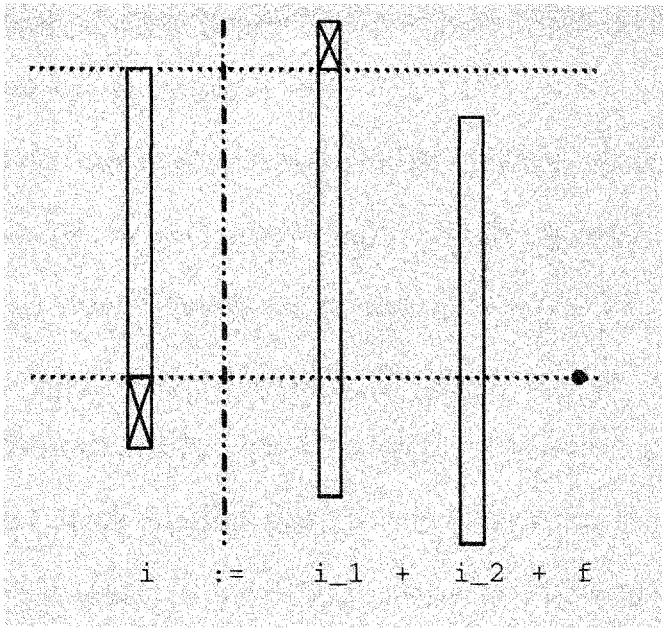


Figure 12: Inference Diagram

In the assignment, we have an inferable variable i on the left-hand side and inferable variables i_1, i_2 and fixed variable f on the right-hand side. The adjustments to the intervals for the inferable variables are shown as the crossed-out areas. According the typing rule for assignments, the bottom portion of the interval for i that is below f is "cut off" because i's security level need to be equal to f's or higher. Also, the top portion of the interval for i_1 is"cut off" because the typing rule for assignments requires that i_1's

security level be less than or equal to i's. After these conservative "cut-offs", we observe that the bottoms of the adjusted intervals guarantee the satisfaction of the typing rule for assignments. Of course this is just one of the many scenarios for assignments, for example, we could have fixed variable on the left-hand side with all inferable variables on the right-hand side, or have all inferable variables on both sides. We are able to make similar adjustments and arguments for all the scenarios for assignments.

Encouraged by the results, we studied all cases of our language commands and found out that in each case, after minimum adjustments made according to the typing rule, the bottoms of the intervals satisfy the typing rule.

Notice that when the interval of an inferable variable changes, other inferable variables may be affected. Therefore, we need to iteratively adjust the intervals until a stable state is reached, that is, no more adjustments can be made according to the typing rules.

Will the iteration end up in an infinite loop? The answer is no because we have shown that in each case, the intervals only "shrink" in height. Therefore, at some point, either the iteration fails, that is, the required minimum adjustments cannot be made (after adjustments, the interval is empty), or the iteration reaches the stable state.

The fact that upon success, we only need to report the bottoms of the intervals for inferable variables leads to a great optimization of our inference algorithm: we only need to keep track of the bottoms of the intervals instead of the whole intervals.

Finally, we reached a surprisingly simple approach for type inference that is closely based on the type system:

We have fixed variables and inferable variables in the program. Initially, all security levels for inferable variables are initialized to $\perp$, the lowest level in the security lattice. In

the process of type checking, if a typing rule is not satisfied, but raising the level of an inferable variable can satisfy it, then we raise the level of that variable by the minimum amount necessary. We succeed if a stable state is reached and report all the current types for inferable variables as an instance of successful typing.

The rest of the chapter is organized as follows. We present our inference algorithm in Section 4.2 and prove its soundness and completeness in Section 4.3. In Section 4.4, we describe our techniques for error reporting. In Section 4.5, we illustrate the capabilities of our approach on an example program that processes some medical information. Finally, Section 4.6 discusses related work.

4.2 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 4.2.1 FIXED *is the set of variables whose declarations are of the form* τ *var.* FIXED-CONTENTS *is the set of array variables whose declarations are of the form* τ *arr ? or* τ_1 *arr* τ_2. FIXED-LENGTH *is the set of array variables whose declarations are of the form ? arr* τ*, or* τ_1 *arr* τ_2. *(Note that a variable declared* τ_1 *arr* τ_2 *is in both* FIXED-CONTENTS *and* FIXED-LENGTH.*)*

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 4.2.2 *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* τ_2, $\gamma'(x) = \tau_1'$ *arr* τ_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* $\tau_2 = \tau_2'$.

Note that that $\leq$ relation between identifier typings is a partial order, that is, $\leq$ is reflexive, anti-symmetric and transitive.

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 raised as necessary to get a well-typed program. Algorithm B is given in Figure 13. It makes use of an auxiliary function $Lev(\gamma, e)$ that returns the minimal type of e under γ. *Lev* follows straightforwardly from the typing rules for expressions.

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

$B(\gamma, pc, x := e)$ 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 $Lev(\gamma, e)$.

$B(\gamma, pc, c) =$ case c of

$c_1; c_2$:

let $\gamma_1 = B(\gamma, pc, c_1)$

$\gamma_2 = B(\gamma_1, pc, c_2)$

in if $\gamma_2 = \gamma_1$ then γ_2 else $B(\gamma_2, pc, c_1; c_2)$

$x := e$:

let $\tau\ var = \gamma(x)$

in if $x \in$ FIXED and $pc \vee Lev(\gamma, e) \not\leq \tau$ then *fail*;

$\gamma[x := (\tau \vee pc \vee Lev(\gamma, e))\ var]$

while e **do** c :

if $B(\gamma, pc \vee Lev(\gamma, e), c) = \gamma'$ and $\gamma = \gamma'$ then γ'

else $B(\gamma', pc,$ **while** e **do** $c)$

if e **then** c_1 **else** c_2 :

if $B(\gamma, pc \vee Lev(\gamma, e), c_1; c_2) = \gamma'$ and $\gamma = \gamma'$ then γ'

else $B(\gamma', pc,$ **if** e **then** c_1 **else** $c_2)$

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*;

$\gamma[x := (\tau_1 \vee \tau_2 \vee pc \vee Lev(\gamma, e))\ arr\ (\tau_2 \vee pc \vee Lev(\gamma, e))]$

$x[e_1] := e_2$:

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*;

$\gamma[x := (\tau_1 \vee \tau_2 \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2))\ arr\ \tau_2]$

skip:

γ

Figure 13: Type Inference Algorithm B

$Lev(\gamma, e)$ = 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 14: Helper Function *Lev*

In $B(\gamma, pc, \textbf{while}\ e\ \textbf{do}\ c)$, we simply call B on c with the join of pc and $Lev(\gamma, e)$ as the new pc. If $B(\gamma, pc, \textbf{while}\ e\ \textbf{do}\ c)$ succeeds and returns γ', where $\gamma' = \gamma$, then B terminates and returns γ. Otherwise, we recursively call B with the newly returned γ'.

$B(\gamma, pc, \textbf{allocate}\ x[e])$ is very similar to $B(\gamma, pc, x := e)$ 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 $B(\gamma, pc, c_1; c_2)$, B is first run on c_1 with pc and γ, returning γ_1, which is used in turn to run B on c_2, returning γ_2. If γ_2 is the same as γ_1, B terminates and returns γ_2. Otherwise, the whole process repeats with γ_2 as the new identifier typing.

4.3 Properties of the Inference Algorithm

In this section, we establish the soundness and completeness of algorithm B with respect to the typing rules.

Lemma 4.3.1 (Termination) *$B(\gamma, pc, c)$ either fails or terminates and returns γ' such that $\gamma \leq \gamma'$.*

Proof. By induction on the structure of c.

1. Case $x := e$.

   According to B, if $x \in$ FIXED, B either fails or returns γ' such that $\gamma = \gamma'$.

   If $x \notin$ FIXED, B always terminates and returns γ' such that $\gamma \leq \gamma'$.

2. Case $x[e_1] := e_2$.

   According to B, if $x \in$ FIXED-CONTENTS, B either fails or returns γ' such that $\gamma = \gamma'$.

   If $x \notin$ FIXED-CONTENTS, B always terminates and returns γ' such that $\gamma \leq \gamma'$.

3. Case **allocate** $x[e]$.

   According to B, if $x \in$ FIXED-CONTENTS and $x \in$ FIXED-LENGTH, B either fails or returns γ' such that $\gamma = \gamma'$.

   If $x \in$ FIXED-CONTENTS and $x \notin$ FIXED-LENGTH, B either fails or returns γ' such that $\gamma \leq \gamma'$, where $\gamma(x) = \tau_1\ arr\ \tau_2$, $\gamma'(x) = \tau_1\ arr\ \tau_2'$, and $\tau_2 \subseteq \tau_2' \subseteq \tau_1$.

   If $x \notin$ FIXED-CONTENTS and $x \in$ FIXED-LENGTH, B either fails or returns γ' such that $\gamma = \gamma'$, where $\gamma(x) = \gamma'(x) = \tau_1\ arr\ \tau_2$.

   If $x \notin$ FIXED-CONTENTS and $x \notin$ FIXED-LENGTH, B always terminates and returns γ' such that $\gamma \leq \gamma'$.

4. Case **skip**.

   According to B, $B(\gamma, pc, \textbf{skip}) = \gamma$.

5. Case **while** e **do** c.

   According to B, we have $B(\gamma, pc \vee Lev(\gamma, e), c)$. Since command c is simpler in structure than **while** e **do** c, by induction, $B(\gamma, pc \vee Lev(\gamma, e), c)$ either fails or terminates and return γ' such that $\gamma \leq \gamma'$.

   If $B(\gamma, pc \vee Lev(\gamma, e), c)$ returns γ' and $\gamma \neq \gamma'$, then according to B, we have $B(\gamma', pc \vee Lev(\gamma', e), c)$, where $\gamma' > \gamma$. Under a finite lattice, we know that eventually, $B(\gamma', pc \vee Lev(\gamma', e), c)$ either fails or returns some γ'', where $B(\gamma'', pc \vee Lev(\gamma'', e), c) = \gamma''$.

   Therefore, $B(\gamma, pc,$ **while** e **do** $c)$ either fails ro terminates and return γ' such that $\gamma \leq \gamma'$.

6. Case **if** e **then** c_1 **else** c_2.

   According to B, we have $B(\gamma, pc \vee Lev(\gamma, e), c_1; c_2)$. Since command $c_1; c_2$ is simpler in structure than **if** e **then** c_1 **else** c_2, by induction, $B(\gamma, pc \vee Lev(\gamma, e), c_1; c_2)$ either fails or terminates and return γ' such that $\gamma \leq \gamma'$.

   If $B(\gamma, pc \vee Lev(\gamma, e), c_1; c_2)$ returns γ' and $\gamma \neq \gamma'$, then according to B, we have $B(\gamma', pc \vee Lev(\gamma', e), c_1; c_2)$, where $\gamma' > \gamma$. Under a finite lattice, we know that eventually, $B(\gamma', pc \vee Lev(\gamma', e), c_1; c_2)$ either fails or returns some γ'', where $B(\gamma'', pc \vee Lev(\gamma'', e), c_1; c_2) = \gamma''$.

   Therefore, $B(\gamma, pc,$ **if** e **then** c_1 **else** $c_2)$ either fails or terminates and return γ' such that $\gamma \leq \gamma'$.

7. Case $c_1; c_2$.

   By induction, $B(\gamma, pc, c_1)$ either fails or returns γ' such that $\gamma \leq \gamma'$. Also by induction, $B(\gamma', pc, c_2)$ either fails or returns γ'' such that $\gamma' \leq \gamma''$.

If either $B(\gamma, pc, c_1)$ or $B(\gamma', pc, c_2)$ fails, $B(\gamma, pc, c_1; c_2)$ fails.

If $B(\gamma, pc, c_1)$ and $B(\gamma', pc, c_2)$ both terminate and $\gamma' = \gamma''$, case proved.

If $B(\gamma, pc, c_1)$ and $B(\gamma', pc, c_2)$ both terminate and $\gamma' \neq \gamma''$, then according to B, recursively run $B(\gamma'', pc, c_1; c_2)$ again. B will not loop indefinitely, because with each recursive call, some inferable variables in the starting γ end up being raised higher in γ''. Under the assumption that we only have a finite number of variables in a program and a finite number of security levels in a lattice, ultimately, B either fails or terminates with γ'' such that $\gamma' = \gamma''$ and $\gamma \leq \gamma'$.

□

Lemma 4.3.2 *Suppose identifier typings γ and γ' are identical except for x, then*

1. *If $\gamma(x) = \tau$ var, $\gamma'(x) = (\tau \vee \tau_a)$ var $= \tau'$ var, and x appears in expression e, then $Lev(\gamma', e) = \tau' \vee Lev(\gamma, e)$.*

2. *If $\gamma(x) = \tau_1$ arr τ_2, $\gamma'(x) = (\tau_1 \vee \tau_a)$ arr $\tau_2 = \tau'$ arr τ_2, and $x[\,]$ appears in expression e, then $Lev(\gamma', e) = \tau' \vee Lev(\gamma, e)$.*

3. *If $\gamma(x) = \tau_1$ arr τ_2, $\gamma'(x) = \tau_1$ arr $(\tau_2 \vee \tau_a) = \tau_1$ arr τ', and x.**length** appears in expression e, then $Lev(\gamma', e) = \tau' \vee Lev(\gamma, e)$.*

Proof. For case 1, by induction on the structure of e:

a Case x,

$Lev(\gamma', x) = \tau_a \vee \tau \vee \tau = \tau' \vee Lev(\gamma, x)$.

b Case x_1.**length**.

Since x does not appear in e, it holds vacuously.

c Case $x_1[e_1]$.

Variable x must appear in e_1. By induction, $Lev(\gamma', e_1) = \tau' \vee Lev(\gamma, e_1)$. Let $\gamma(x_1) = \tau_1\ arr\ \tau_2$, we have

$$\begin{aligned} Lev(\gamma', x_1[e_1]) &= \tau_1 \vee Lev(\gamma', e_1) \\ &= \tau' \vee \tau_1 \vee Lev(\gamma, e_1) \\ &= \tau' \vee Lev(\gamma, x_1[e_1]). \end{aligned}$$

d Case $e_1 + e_2$.

We prove the case where x appears in both e_1 and e_2. Other cases are similar and simpler.

By induction, $Lev(\gamma', e_1) = \tau' \vee Lev(\gamma, e_1)$. By induction, $Lev(\gamma', e_2) = \tau' \vee Lev(\gamma, e_2)$.

So,

$$\begin{aligned} Lev(\gamma', e_1 + e_2) &= Lev(\gamma', e_1) \vee Lev(\gamma', e_2) \\ &= \tau' \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2) \\ &= \tau' \vee Lev(\gamma, e_1 + e_2). \end{aligned}$$

For case 2, by induction on the structure of e:

a Case x_1,

Since $x[]$ does not appear in e, it holds vacuously.

b Case x_1.**length**.

Since $x[]$ does not appear in e, it holds vacuously.

c Case $x_1[e_1]$.

$x[]$ must appear in e_1. By induction, $Lev(\gamma', e_1) = \tau' \vee Lev(\gamma, e_1)$. Let $\gamma(x_1) = \gamma'(x_1) = \tau_1''\ arr\ \tau_2''$, we have

$$\begin{aligned} Lev(\gamma', x_1[e_1]) &= \tau_1'' \vee Lev(\gamma', e_1) \\ &= \tau' \vee \tau_1'' \vee Lev(\gamma, e_1) \\ &= \tau' \vee Lev(\gamma, x_1[e_1]). \end{aligned}$$

d Case $e_1 + e_2$.

We prove the case where x appears in both e_1 and e_2. Other cases are similar and simpler.

By induction, $Lev(\gamma', e_1) = \tau' \vee Lev(\gamma, e_1)$. By induction, $Lev(\gamma', e_2) = \tau' \vee Lev(\gamma, e_2)$.

So,

$$\begin{aligned} Lev(\gamma', e_1 + e_2) &= Lev(\gamma', e_1) \vee Lev(\gamma', e_2) \\ &= \tau' \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2) \\ &= \tau' \vee Lev(\gamma, e_1 + e_2). \end{aligned}$$

For case 3, by induction on the structure of e:

a Case x_1,

Since x.**length** does not appear in e, it holds vacuously.

b Case x.**length**.

$Lev(\gamma', e) = \tau_2 \vee \tau_a \vee \tau_2 = \tau' \vee \tau_2 = \tau' \vee Lev(\gamma, e)$.

c Case $x_1[e_1]$.

x.length must appear in e_1. By induction, $Lev(\gamma', e_1) = \tau' \vee Lev(\gamma, e_1)$. Let $\gamma(x_1) = \gamma'(x_1) = \tau_1''$ *arr* τ_2'', we have

$$\begin{aligned} Lev(\gamma', x_1[e_1]) &= \tau_1'' \vee Lev(\gamma', e_1) \\ &= \tau' \vee \tau_1'' \vee Lev(\gamma, e_1) \\ &= \tau' \vee Lev(\gamma, x_1[e_1]). \end{aligned}$$

d Case $e_1 + e_2$.

We prove the case where x appears in both e_1 and e_2. Other cases are similar and simpler.

By induction, $Lev(\gamma', e_1) = \tau' \vee Lev(\gamma, e_1)$. By induction, $Lev(\gamma', e_2) = \tau' \vee Lev(\gamma, e_2)$.

So,

$$\begin{aligned} Lev(\gamma', e_1 + e_2) &= Lev(\gamma', e_1) \vee Lev(\gamma', e_2) \\ &= \tau' \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2) \\ &= \tau' \vee Lev(\gamma, e_1 + e_2). \end{aligned}$$

□

Lemma 4.3.3 (Fixed Point) *If* $B(\gamma, pc, c) = \gamma'$, *then* $B(\gamma', pc, c) = \gamma'$.

Proof. By induction on the structure of c:

1. Case $x := e$.

   If $\gamma = \gamma'$, case proved.

   Otherwise, from Lemma 4.3.1, $\gamma' \not\leq \gamma$. By algorithm B, we must have $\gamma'(x) \not\leq \gamma(x)$. Therefore, x must be inferable, and B always terminates. x is the only variable having its type raised.

   According to $B(\gamma, pc, c := e) = \gamma'$, we have $\gamma'(x) = \tau'\ var$ and $\tau' = \tau \vee pc \vee Lev(\gamma, e)$. Therefore, according to $B(\gamma', pc, c := e) = \gamma''$, $\gamma''(x) = \tau''\ var$ and $\tau'' = \tau' \vee pc \vee Lev(\gamma', e)$.

If x does not appear in e, then

$$\begin{aligned}\tau'' &= \tau' \vee pc \vee Lev(\gamma, e) \\ &= (\tau \vee pc \vee Lev(\gamma, e)) \vee pc \vee Lev(\gamma, e) \\ &= \tau \vee pc \vee Lev(\gamma, e) \\ &= \tau'\end{aligned}$$

Therefore, $\tau''(x) = \tau'(x)$, case proved.

If x does appear in e, by Lemma 4.3.4,

$$\begin{aligned}\tau'' &= \tau' \vee pc \vee Lev(\gamma', e) \\ &= \tau' \vee pc \vee \tau' \vee Lev(\gamma, e) \\ &= \tau'\end{aligned}$$

Therefore, $\tau''(x) = \tau'(x)$, case proved.

2. Case **while** e **do** c.

   According to B, if $B(\gamma, pc, \textbf{while } e \textbf{ do } c) = \gamma'$, then $B(\gamma', pc, \textbf{while } e \textbf{ do } c) = \gamma'$.

   Case proved.

3. Case **if** e **then** c_1 **else** c_2.

   According to B, if $B(\gamma, pc, \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) = \gamma'$, then $B(\gamma', pc, \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) = \gamma'$. Case proved.

4. Case **allocate** $x[e]$.

   Since $B(\gamma, pc \vee Lev(\gamma, e), \textbf{allocate } x[e]) = \gamma'$, let $\gamma(x) = \tau_1 \ arr \ \tau_2$, $\gamma'(x) = \tau_1' \ arr \ \tau_2'$, we have $\tau_2' = \tau_2 \vee pc \vee Lev(\gamma, e) = \tau_2 \vee \tau'$.

If $\tau \in$ FIXED-CONTENTS, τ_2 must be the only type raised, and $\tau_2' \leq \tau_1$.

$$\begin{aligned}
\tau_2'' &= \tau_2' \vee pc \vee Lev(\gamma', e) \\
&= \tau_2' \vee pc \vee \tau_2 \vee Lev(\gamma, e) \\
&= \tau_2' \vee pc \vee Lev(\gamma, e) \\
&= \tau_2'
\end{aligned}$$

If τ_1 and τ_2 both got raised, and $x[]$ and $x.\textbf{length}$ both appear in e and e_1, we have $\tau_1' = \tau_2' = \tau_1 \vee \tau_2 \vee pc \vee Lev(\gamma, e)$.

$$\begin{aligned}
\tau_2'' &= \tau_2' \vee pc \vee Lev(\gamma', e) \\
&= \tau_2' \vee pc \vee (pc \vee \tau_2 \vee \tau_1 \vee Lev(\gamma, e)) \\
&= \tau_2' \vee \tau_1 \vee pc \vee Lev(\gamma, e) \\
&= \tau_2'
\end{aligned}$$

Similarly, we have $\tau_1'' = \tau_1'$. Case proved.

5. Case $x[e_1] := e_2$.

   If $\gamma = \gamma'$, case proved.

   Otherwise, from Lemma 4.3.1, $\gamma' \not\leq \gamma$. By algorithm B, we must have $\gamma'(x) \not\leq \gamma(x)$. Therefore, x must be inferable, and B always terminates. x is the only variable having its type raised.

   According to $B(\gamma, pc, x[e_1] := e_2) = \gamma'$, we have $\gamma'(x) = \tau'\ arr\ \tau_2$ and $\tau' = \tau \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2)$. Therefore, according to $B(\gamma', pc, c := e) = \gamma''$, $\gamma''(x) = \tau''\ var$ and $\tau'' = \tau' \vee pc \vee Lev(\gamma', e_1) \vee Lev(\gamma', e_2)$.

If x does not appear in e, then

$$\begin{aligned}\tau'' &= \tau' \vee pc \vee Lev(\gamma', e_1) \vee Lev(\gamma', e_2) \\ &= \tau' \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma', e_2) \\ &= \tau \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_2) \\ &= \tau'\end{aligned}$$

Therefore, $\tau''(x) = \tau'(x)$, case proved.

If $x[]$ does appear in e_1 and e_2, by Lemma 4.3.4,

$$\begin{aligned}\tau'' &= \tau' \vee pc \vee Lev(\gamma', e_1) \vee Lev(\gamma', e_2) \\ &= \tau' \vee pc \vee \tau \vee Lev(\gamma, e_1) \vee \tau \vee Lev(\gamma, e_2) \\ &= \tau'\end{aligned}$$

Therefore, $\tau''(x) = \tau'(x)$, case proved.

6. Case $c_1; c_2$.

   According to B, when $\gamma_2 = \gamma_1 = \gamma'$, we have $B(\gamma', pc, c_2) = \gamma'$, by induction, $B(\gamma', pc, c_2) = \gamma'$.

   According to B, we have $B(\gamma_?, pc, c_1) = \gamma'$, by induction, $B(\gamma', pc, c_1) = \gamma'$.

   Therefore, $B(\gamma', pc, c_1; c_2) = \gamma'$.

7. Case **skip**.

   Since $B(\gamma, pc, \mathbf{skip}) = \gamma$, case proved.

□

Lemma 4.3.4 $\gamma \vdash e : Lev(\gamma, e)$ *holds. And if* $\gamma \vdash e : \tau$, *then* $Lev(\gamma, e) \leq \tau$.

Proof. By looking at $Lev(\gamma, e)$ in algorithm B. $Lev(\gamma, e)$ is in fact the minimal type e can have under identifier typing γ.

Therefore the lemma holds. □

Lemma 4.3.5 *If* $\gamma \leq \gamma'$, *then* $Lev(\gamma, e) \leq Lev(\gamma', e)$.

Proof. Proof by induction on the structure of e. □

Theorem 4.3.6 (Soundness) *If* $B(\gamma, pc, c)$ *succeeds and returns* γ', *then* $\gamma' \vdash c : pc\ cmd$.

Proof. By induction on the structure of c:

1. Case $x := e$.

   By Lemma 4.3.3, we have $\gamma'(x) = \tau'\ var$, where $\tau' = \tau' \vee pc \vee Lev(\gamma', e)$, and $\gamma' \vdash e : Lev(\gamma', e)$. By typing rule ASSIGN, $\gamma' \vdash x := e : \tau'\ cmd$. Since $pc \leq \tau'$, we have $\gamma' \vdash x := e : pc\ cmd$.

2. Case **while** e **do** c.

   By Lemma 4.3.3, we have $B(\gamma', pc \vee Lev(\gamma', e), c) = \gamma'$, so by induction, we have $\gamma' \vdash c : (pc \vee Lev(\gamma', e))\ cmd$. Also, by Lemma 4.3.4, we have $\gamma' \vdash e : pc \vee Lev(\gamma', e)$.

   Therefore, by WHILE, $\gamma' \vdash$ **while** e **do** $c : (pc \vee Lev(\gamma', e))\ cmd$.

   By sub-typing rule CMD$^-$, $\gamma' \vdash$ **while** e **do** $c : pc\ cmd$.

3. Case **if** e **then** c_1 **else** c_2.

   By Lemma 4.3.3, we have $B(\gamma', pc \vee Lev(\gamma', e), c_1; c_2) = \gamma'$. So by induction, we have $\gamma' \vdash c_1; c_2 : (pc \vee Lev(\gamma', e))\ cmd$.

By typing rule COMPOSE, we have $\gamma' \vdash c_1 : (pc \vee Lev(\gamma', e))\ cmd$ and $\gamma' \vdash c_2 : (pc \vee Lev(\gamma', e))\ cmd$. Also, by Lemma 4.3.4, we have $\gamma' \vdash e : pc \vee Lev(\gamma', e)$.

Therefore, by typing rule IF, $\gamma' \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : (pc \vee Lev(\gamma', e))\ cmd$.

By sub-typing rule CMD$^-$, $\gamma' \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : pc\ cmd$.

4. Case **allocate** $x[e]$.

   Let $\gamma'(x) = \tau_1'\ arr\ \tau_2'$, by Lemma 4.3.3, we have $\tau_1' = \tau_1' \vee \tau_2' \vee pc \vee Lev(\gamma', e)$, $\tau_2' = \tau_2' \vee pc \vee Lev(\gamma', e)$, and $\gamma' \vdash e : Lev(\gamma', e)$. Therefore, by the Global Constraint and ALLOCATE, we have $\gamma' \vdash \textbf{allocate}\ x[e] : (\tau_2' \vee pc \vee Lev(\gamma', e))\ cmd$.

   By sub-typing rule CMD$^-$, $\gamma' \vdash \textbf{allocate}\ x[e] : pc\ cmd$.

5. Case $x[e_1] := e_2$.

   By Lemma 4.3.3, we have $\gamma'(x) = \tau_1\ arr\ \tau_2$, where $\tau_1 = \tau_1 \vee \tau_2 \vee pc \vee Lev(\gamma', e_1) \vee \vee Lev(\gamma', e_2)$. By Lemma 4.3.5, $\gamma' \vdash e_1 : Lev(\gamma', e_1)$ and $\gamma' \vdash e_2 : Lev(\gamma', e_2)$. By typing rule ASSIGN-ARR, $\gamma' \vdash x[e_1] := e_2 : \tau_1\ cmd$. Since $pc \leq \tau_1$, by subtyping rule CMD$^-$, we have $\gamma' \vdash x[e_1] := e_2 : pc\ cmd$.

6. Case $c_1; c_2$.

   By induction, we have $\gamma' \vdash c_1 : pc\ cmd$ and $\gamma' \vdash c_2 : pc\ cmd$. By typing rule COMPOSE, $\gamma' \vdash c_1; c_2 : pc\ cmd$.

7. Case **skip**.

   Since $\gamma' \vdash \textbf{skip} : \top\ cmd$ and $pc \leq \top$, by subtyping rule CMD$^-$, we have $\gamma' \vdash \textbf{skip} : pc\ cmd$.

□

Theorem 4.3.7 (Completeness) *If* $\gamma' \vdash c : \tau\ cmd$ *and* $\gamma \leq \gamma'$, *then* $B(\gamma, \tau, c)$ *succeeds and returns* γ'' *such that* $\gamma'' \leq \gamma'$.

Proof. By induction on the structure of c.

1. Case $x := e$.

   According to the typing rules, there is τ_1, such that $\tau \leq \tau_1$, and $\gamma' \vdash e : \tau_1$, $\gamma'(x) = \tau_1\ var$, and $\gamma' \vdash x := e : \tau_1\ cmd$.

   According to $B(\gamma, \tau, x := e)$, if $x \in$ FIXED, then $\gamma(x) = \gamma'(x) = \tau_1\ var$. By Lemma 4.3.4 and 4.3.5, $lev(\gamma, e) \leq lev(\gamma', e) \leq \tau_1$, so $\tau \vee lev(\gamma, e) \leq \tau_1$, B succeeds and $\gamma'' \leq \gamma'$. If $x \notin$ FIXED, then let $\tau_2\ var = \gamma(x)$, since $\gamma \leq \gamma'$, $\tau_2 \leq \tau_1$. So, $\gamma''(x) = (\tau_2 \vee \tau \vee Lev(\gamma, e))\ var$ and $\tau_2 \vee \tau \vee Lev(\gamma, e) \leq \tau_1$, therefore, B succeeds and $\gamma'' \leq \gamma'$.

2. Case **while** e **do** c.

   According to the typing rules, there is τ_1, such that $\tau \leq \tau_1$, $\gamma' \vdash e : \tau_1$, and $\gamma' \vdash c : \tau_1\ cmd$.

   By Lemma 4.3.4 and 4.3.5, $Lev(\gamma, e) \vee \tau \leq Lev(\gamma', e) \vee \tau \leq \tau_1$. By induction, $B(\gamma, Lev(\gamma, e) \vee \tau, c) = \gamma_1$ succeeds and $\gamma_1 \leq \gamma'$.

   If $\gamma_1 \neq \gamma'$, then by assumption and CMD$^-$, we have $\gamma' \vdash c : \gamma_1(e) \vee \tau\ cmd$, and $\gamma_1 \leq \gamma$, so $B(\gamma_1, Lev(\gamma_1, e) \vee \tau, c) = \gamma_2$ succeeds and $\gamma_2 \leq \gamma'$. If $\gamma_1 \neq \gamma_2$, we repeat the process until we reach a stable type mapping γ_i under a finite lattice, where $\gamma_i \leq \gamma'$.

   Therefore, according to B, $B(\gamma, \tau,$ **while** e **do** $c) = \gamma''$ and $\gamma'' \leq \gamma'$.

3. **if** e **then** c_1 **else** c_2.

   According to the typing rules, there is τ_1, such that $\tau \leq \tau_1$, $\gamma' \vdash e : \tau_1$, and $\gamma' \vdash c_1 : \tau_1\ cmd$, $\gamma' \vdash c_2 : \tau_1\ cmd$..

   By Lemma 4.3.4 and 4.3.5, $Lev(\gamma, e) \vee \tau \leq Lev(\gamma', e) \vee \tau \leq \tau_1$. By induction, $B(\gamma, Lev(\gamma, e) \vee \tau, c_1; c_2) = \gamma_1$ succeeds and $\gamma_1 \leq \gamma'$.

   If $\gamma_1 \neq \gamma'$, then by assumption and CMD$^-$, we have $\gamma' \vdash c : \gamma_1(e) \vee \tau\ cmd$, and $\gamma_1 \leq \gamma$, so $B(\gamma_1, Lev(\gamma_1, e) \vee \tau, c) = \gamma_2$ succeeds and $\gamma_2 \leq \gamma'$. If $\gamma_1 \neq \gamma_2$, we repeat the process until we reach a stable type mapping γ_i under a finite lattice, where $\gamma_i \leq \gamma'$.

   Therefore, according to B, $B(\gamma, \tau, \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2) = \tau''$ and $\gamma'' \leq \gamma'$.

4. $x[e_1] := e_2$.

   According to the typing rules, there is τ_1, such that $\tau \leq \tau_1$, and $\gamma'(x) = \tau_1'\ arr\ \tau_2'$, $\gamma' \vdash e_1 : \tau_1'$ and $\gamma' \vdash e_2 : \tau_1'$.

   Let $\gamma(x) = \tau_1\ arr\ \tau_2$. According to $B(\gamma, \tau, x[e_1] := e_2)$, if $x \in$ FIXED-CONTENTS, then $\tau_2 \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_1) \leq \tau_1 = \tau_1'$, $x[e_1] := e_2$ succeeds and $\gamma = \gamma'$. If $x \notin$ FIXED-CONTENTS, then $\tau_1 \vee \tau_2 \vee pc \vee Lev(\gamma, e_1) \vee Lev(\gamma, e_1) \leq \tau_1'$, therefore, $\gamma'' \leq \gamma'$.

5. Case **allocate** $x[e]$.

   According to the typing rules, there is τ_1, such that $\tau \leq \tau_1$, and $\gamma'(x) = \tau_1\ arr\ \tau_2$, $\gamma' \vdash e_1 : \tau_1$ and $\gamma' \vdash$ **allocate** $x[e] : \tau_1\ cmd$.

   According to B, $B(\gamma, \tau, \textbf{allocate } x[e]) = \gamma''$ succeeds, and $\gamma'' \leq \gamma'$.

6. Case $c_1; c_2$.

   According to the typing rules, we have $\gamma' \vdash c_1 : \tau\ cmd$ and $\gamma' \vdash c_2 : \tau\ cmd$.

By induction, we have $B(\gamma, \tau, c_1) = \gamma_1$, $\gamma_1 \leq \gamma'$; $B(\gamma_1, \tau, c_2) = \gamma_2$, $\gamma_1 \leq \gamma_2 \leq \gamma'$. According to B, if $\gamma_1 = \gamma_2$, B succeeds and returns γ_2 as γ''. If $\gamma_1 \neq \gamma_2$, we run $B(\gamma_2, \tau, c_1; c_2)$ recursively. By Lemma 4.3.1, γ_1 and γ_2 are approaching γ under a finite lattice, therefore, at some point, B succeeds and $\gamma'' = \gamma_1 = \gamma_2 \leq \gamma'$.

7. Case **skip**.

   Since $B(\gamma, pc, \textbf{skip}) = \gamma$, case proved.

□

We now make a few observations about algorithm B. First, the security levels for inferable variables only increase as B executes. Second, whenever B fails, it must be the result of trying to raise 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.

We can calculate the running time of algorithm B as follows. If the length of c is n and the height of the lattice is h, then it is easy to see that the worst-case running time of B is $O(n^2h)$, since there are $O(n)$ inferable variable, each of which can be raised at most h times.

4.4 Error Reporting

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

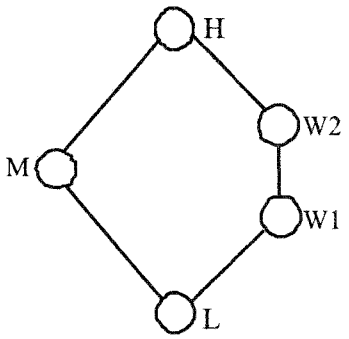
As discussed above, algorithm B begins by setting all inferable security levels to $\perp$. As it runs, it raises these levels as necessary. It fails whenever it finds that a fixed security level needs to be raised. When such a failure occurs, it is important to be able to explain

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 have the following lattice:



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.

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. Arrays declared as ? *arr* ? actually have two security level histories. 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. Further on, for each principal variable of e, we in turn look at its security level history for the corresponding set of principal variables that are responsible for its type. We repeat this process and generate the whole error report cascadingly.

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/example1:17.6: illegal implicit flow
     l := 7;
     ^
--left-hand side, l, has type L by declaration
--pc has type H because
  tests/example1:15.8: guard of control flow statement
  while (m + l > 0) do {
         ^
  has type M because m has type M by declaration
and
  tests/example1:16.15: guard of control flow statement
    while (w1 + w2 > 0) do {
                ^
  has type W2 because w2 has type W2 by declaration
and
  the join of M and W2 is H
```

Following 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.

Next, we show an example of clean, manageable error reporting via the use of principal variables and security level histories. Using the same security lattice as before, and again with $l : L$, $m : M$, $w1 : W1$, and $w2 : W2$, suppose that n, p, q, j, and k are inferable variables. Consider the following program:

```
p := m;

q := p;

j := w1;

j := w2;

k := j;

l := l + q + k + n
```

On this program, error report from the type checker gives a detailed explanation of why the expression $l + q + k + n$ has type H; this requires explaining the types of the inferable variables q, p, k, and j:

```
tests/example:29.1: illegal explicit flow
  l := l + q + k + n
  ^       ^   ^
--left-hand side, l, has type L by declaration
--right-hand side, l + q + k + n, has type H because
    q has type M because
       tests/example:23.6: explicit flow
       q := p;
       ^    ^
       and p has type M because
           tests/example:22.6: explicit flow
           p := m;
           ^    ^
           and m has type M by declaration
and
    k has type W2 because
       tests/example:26.6: explicit flow
       k := j;
       ^    ^
       and j has type W2 because
           tests/example:25.6: explicit flow
           j := w2;
           ^    ^
           and w2 has type W2 by declaration
```

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.

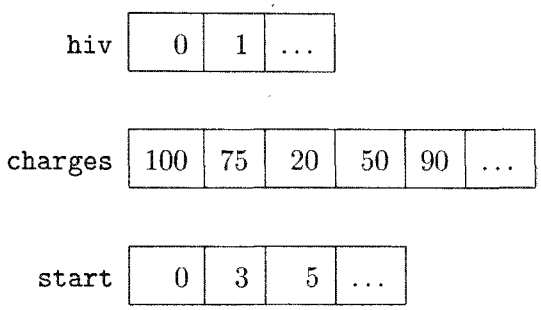
4.5 A Larger Example

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:



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 15.

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.

```
// 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
    };
    // Select median and store it into hivmedians:
    hivmedians[k] := temp[n/2];
    k := k+1
  }
  else { skip };

  lk := temp.length;
  i := i+1
}
```

Figure 15: 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, temp.length, has type H because
    tests/hiv:61.6: guard of control flow statement
       if hiv[i] then {
          ^
    has type H because hiv[] has type H
  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 16: Output for HIV Program

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 16. The first message reports that the assignment to `lk` is illegal, because 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.

4.6 Related Work

Constraint-based type inference approaches have been used in a number of previous works on secure information flow.

Volpano and Smith [23] describe sound and complete type inference for a simple sequential language with procedures. Procedures are polymorphic with respect to security types.

In Jif [12], a Java-like language, type inference is used to assign labels automatically to the methods of a class. The project is maintained by a research group at Cornell University. Jif uses the algorithm in [30] to resolve constraints. As for error reporting, Jif's compiler `jifc` has a command line argument for more detailed error explanations. However, the error reporting is not always accurate. Sometimes the constraint reported may not be the culprit that prevents the program's compiling. To our knowledge, there is no formal proofs of the security properties of Jif or typing with type inference.

Flow Caml [31, 24] 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 [32] has been implemented as a separate module for general use. However, as discussed in the Introduction, it does not attempt to localize type errors.

Sun, Banerjee, and Naumann [33] 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.

5 Language Extensions and Application Development Framework

In Chapter 3, arrays are added to the basic language. In Chapter 4, a novel type inference algorithm and error reporting methods are presented. Although our language and its type system are simple and manageable as desired in our goals, it needs more language features for practical use. In this chapter, we discuss the addition of more data types, references and functions.

There is currently little research work on how to develop applications using secure information flow. We propose a novel application development framework that covers all aspects from application specification to deployment. We identify the application scenario as multi-party computations and define core computations for secure information flow. We give procedural directions for developing core computations and the structural scheme for putting together core computations and supporting modules. Certification and authentication of code modules are discussed for deployment purpose.

5.1 Adding More Language Features

5.1.1 Adding More Data Types

Our language so far is limited in data types. Only integer literals and variables are allowed. However, it is no trouble at all to add data types like booleans, float point numbers, and strings into our language because the data types and security types are orthogonal to each other. Adding more data types would have no effect on security typing rules. Data typings and security typings do not interfere with each other.

5.1.2 Adding References

We add references in a similar way as in Flow Caml. Because a reference can be accessed through both reading and writing, it is both covariant and contravariant, i.e., it is invariant.

So a reference type does not have any associated subtyping rules. We denote reference types by $\tau\, ref$.

5.1.3 Adding Functions

Our type inference algorithm is amenable to the addition of functions in our language. One straightforward approach is to treat functions as we are currently treating a program. We initialize the parameters of a function with the security types of arguments. Then run the type inference algorithm on the function. If successful, the type of the return variable is used as the function's return type. Inside the function body, if another function call is made, we repeat the same process.

For recursive functions, if the parameters maintain the same security levels across recursive calls, then the return security level is constant. They can be handled just like non-recursive functions.

In practice, we can use caching to get better performance. When we need to do type inference on a function call, we first check the cache for an entry with the same argument types. If an entry is located, the return type in the entry is used as the function return type. If no cache entry is found, type inference is carried out on the function call and upon success, a new entry will be added to the cache.

Our straightforward approach for handling functions also accommodate global variables visible inside the function body. If a global variable used within the function body is raised by the type inference algorithm, we record it.

However, it is much more complicated to use cache for functions that make use of global variables. There might be calls to other functions, which in turn uses global variables. So building the cache index is more complicated.

5.2 Application Development Framework

5.2.1 Application Scenario and Core Computations

We identify multi-party computations as our application scenario, where input/output data of multiple parties are involved for a computational purpose. For example, in our tax e-filing example in Chapter 1, the application involves input/output data of multiple parties - the user, TaxCom, and the IRS, mainly for the purpose of calculating taxes and tax preparation fees.

Writing a useful application usually involves much complexity, such as platform API's, I/O, and networking. Enforcing secure information flow on the whole application is unnecessary and unrealistic. We would like to limit the enforcement of secure information flow to a code module in the application that only deals with the computation of input/output data from multiple parties.

A core computation communicates through well-defined interfaces with supporting code modules to set and send its input/output variables. Each supporting code module represents a party in the application. A party is a person or organization that participates in the application, such as the IRS and TaxCom in our tax e-filing example. Each party has its own security specification that shows what data is sent to the core computation as input and how the data can be shared with other parties, and what data a party gets as output from a core computation.

Note that supporting code modules do not make use of secure information flow and are responsible for the common services such as I/O, user interface, encryption and networking.

The structural scheme is shown in Figure 17. There we have supporting modules(A, B, etc.) representing parties(A, B, etc.). The modules implement corresponding interfaces(IA,

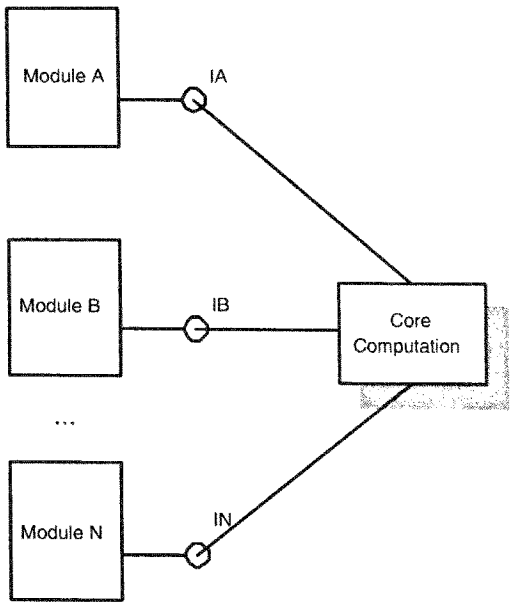


Figure 17: Core Computation and its Environment

IB, etc.), which specify input/output data for the respective modules. The core computation communicates with the supporting modules through their well-defined interfaces.

Next, we describe how to develop and type check a core computation.

5.2.2 Development of Core Computations

Suppose initially we only have a group of parties interested in creating a secure information flow application. We provide a 3-step procedure to develop the core computation.

1. To identify input/output variables.

   In this step, the problem that involves multiple parties is identified. Data of input and output of each party are also identified. For example, we write $a : A_{in}$; $b : A_{out}$; to denote that variable a holds some data as input from party A, while b some data as output to party A.

2. To develop the solution without regard to secure information flow.

   With the problem well defined and input/output variables identified, programmers should develop the core computation without regard to secure information flow. Standard software engineering analysis, development, and testing techniques should be used to make sure that the core computation is algorithmically correct.

3. To incorporate and adjust secure information flow policies.

   Basically, we need to inject security typing into the core computation and type check it. We demonstrate how to develop the security lattice from the security specification and how to adjust the security specification and the lattice in the event of a type checking error.

   First, we use the symbol • to indicate a security level for output variables, the ∘ symbol for a security level for input variables and inferable variables. When a security level is defined for both input and output variables, we overlap the two symbols as a shorthand. For example, for parties A, B and C, initially, we have this security lattice:

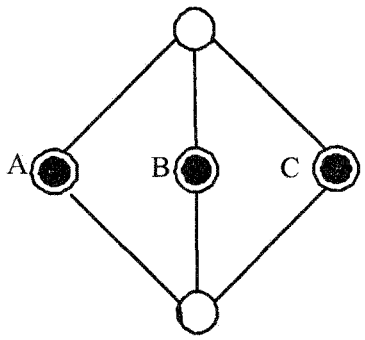


Figure 18: The Initial Lattice

where it says that all input and output variables are of type A, B, or C, and no input variables are to be shared among parties.

Now, we develop the security lattice according to the initial security specification. For example, in

$$i : A_{in}$$

If party A allows information in i to flow to party B, that is, i can be shared between A and B, we would give i a new security level denoted as J_{AB}, and write as:

$$i : A_{in} :: J_{AB}$$

which says that variable i is an input variable from party A and can be shared with party B.

We would also reflect the above policy in the security lattice by creating a new level for J_{AB}. We create a new node under A and B, with lines connected to A and B, as shown below:

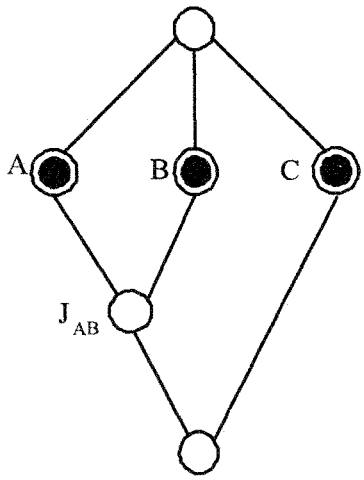


Figure 19: Lattice After Downgrading

After this process, input variables get refined security levels and a new lattice is generated. Now we run the type checker/inference on the annotated core computation.

If type checking/inference finishes successfully, we are finished with core computation. If, however, the type checking/inference returns an error, we still have options to deal with it. Chapter 4, we know the error message must be of this form:

```
fixed_1 := ...

......

inf := fixed_2
```

where we have one fixed variable at one end of the trace, followed by some inferable variables in the middle, and one or more fixed variable at the other end of the trace. Typing errors can be resolved either by eliminating the unintended implicit flows or by relaxing a party's security specification.

The first thing we can do is to examine the error trace and the program to eliminate possible unnecessary implicit flows that contributed to raises of inferable variables.

If the type checking/inference still fails, we have to relax some security policies from some party. For example, if we have variable $a : A_{in} :: J_{AB}$ and $c : C_{in}$ at either end of the trace, where a is a variable from party A and shared with B, while c is a variable from C and not shared:

```
a := ...

......

inf := c
```

From the error report trace, we know that variable c, which is intended only to be seen by party C now needs to be shared with party A and B in computation. Now, if

party C agrees to downgrade variable c from "C only" to "shared between A, B and C", then declaration for c would be:

$$c : A_{in} :: J_{ABC}$$

In this case, we don't need to create a new node for J_{ABC} in the lattice. The $\perp$ node is ideal for the purpose.

As a general rule, whenever an error occurs, we can solve it by asking some parties to relax their security policies and downgrade some input variables. If necessary, new nodes corresponding to the relaxed input security levels should be created in the lattice.

One remarkable thing is that downgrading has a clear meaning with respect to security, that is, the data through downgrading can be shared with more parties, and no new errors are introduced by downgrading through the more relaxed security policies.

5.2.3 Development of Environment for Core Computations

Now we develop the environment where the core computation lives. We require that core computations only communicate with modules via well-defined interfaces. All input/output variables are defined as read-only/write-only properties in the interfaces to make sure that a party can only access its own input/output data.

Properties are a shorthand for a member function call that either reads or sets a public data member of a class. In the following code sample, a property X is defined for read and write in class $MyClass$.

```
class MyClass{
    private int x;
    public  int X {
        get { return x; }
        set { x = value; }  }
}
```

To use properties, here is an example:

```
MyClass myc = new MyClass();
myc.X = 10;
myc.X = myc.X + 10;
```

We see that properties can be used just like a public variable of a class. The difference is that properties are in fact function calls and can be defined with fine-grained read/write attributes with respect to data access. Since properties are function calls, we can use them to trigger actions that come along with a data read or write. For example, a simple write operation of a property in the core computation can cause the data to be encrypted and sent over the Internet in the supporting code module.

Properties can also be declared on interfaces. For example:

```
interface IMyInterface{
    public int X {   get ;   set ;  }
    public int Y {   set ;    }
}
```

In the example, property X is for read/write, while property Y is for write only.

Now, according to the security specification, an interface is created for each party that have all the party's input variables designated as read-only properties while all the party's output variables as write-only properties.

Each party will implement a code module that implements the party's interface. Aside from abiding by the interface definition, the code module is free to do whatever the party desires. For example, a party's code module can choose to encrypt the output data it got from the core computation and send it over the Internet to its home website.

And core computations now only communicate with the code modules from various parties through well-defined interfaces.

5.2.4 Certification and Authentication

To ensure authenticity and integrity, a core computation is signed by a special authority to certify its enforcement of secure information flow, while various supporting code modules will be signed by their owners.

Back to the tax example. It involves three parties, TaxCom, the IRS, and the end user. In this special case, the end user does not participate in the development process. TaxCom will act for the end user with regard to the end user's input/output data to the core computation and security policies. Since the end user only needs the services of the user interface for input and output, a standard module will be provided for the purpose.

When the end user actually runs the tax e-filing application, its security policies will be displayed up-front. If the end user agrees with the way his or her personal financial data and tax preparation payment data flow, he or she may continue using the application and have the absolute assurance of his information privacy. If the user does not like the way the program handles his or her information, the user can decline to use the application. The user can try out applications from other vendors that satisfy his or her privacy concerns.

5.3 Related Work

There has been very little research work in the area of application of secure information flow. One such work is Jif/Split in [34]. The basic idea for Jif/Split is that a secure information flow program can be split into sub-programs and run on hosts of various levels of trust. However, how to develop an application of secure information flow and achieve the initial program is not discussed in the paper.

6 Conclusions and Future Work

Although the problem of secure information flow has been actively researched for nearly ten years, there has been virtually no practical impact. We believe that one reason for the lack of application is that current type systems are overly complicated and restrictive, making practical programming hard or impossible. Also, type inference is essential for practical secure information flow because a user is usually only concerned with the security types for the input/output variables of a program. An inference program should try to automatically type all the auxiliary variables and report to the user if the whole program is typable or not. Currently, few languages feature type inference [12, 27]. And they invariably rely on constraint-based type inference approaches, where constraints in the form of $\alpha \leq \beta$ (α and β are type terms) are gathered, and then usually a well-studies constraint-solving algorithm is applied to the constraint set to check its satisfiability. While well-studied and efficient, the constraint-solving algorithms do badly when something is not right, that is, when the whole constraint set is not satisfiable. Usually, the algorithms will stop at the first unsatisfiable constraint and try to explain the error in terms of the constraint-solving process, not the original program and type system. As a result, all existing secure information flow languages do poorly in terms of error reporting, making the inherently more difficult type system harder to use. Another major reason for lack of practical use is that little work has been done to identify and study scenarios most suitable for use of secure information flow. Some researchers aim to use secure information flow to achieve "end-to-end" security properties of systems, that is, to apply secure information flow to the whole system to achieve data confidentiality and integrity. This is a very ambitious goal. Aside from its complexity, one inevitable situation is that some downward information flow

must be allowed in the application. And the problem is what security properties can be guaranteed if such downward information flows are allowed.

With the major hurdles to practical use identified, we adopt a minimalist approach to developing our language and type system. We start out with a simple language [2] and as we expand the language, we only add language features that we deem most important for practical use. In this way, we manage to keep our language and type system simple and manageable for regular programmers.

The first language feature we decided to add is arrays. Arrays are indispensable for writing useful programs. Due to the various leaking channels associated with array operations, arrays have received complicated and restrictive typing rules in current secure information flow languages. We propose lenient array operations to overcome restrictions. Because of our lenient execution model and our array types of the form τ_1 *arr* τ_2, we are able to do secure information flow analysis on interesting programs, using simple and permissive typing rules. The simplicity of our rules makes it straightforward to prove that our type system ensures noninterference. Our tax calculation example suggests that interesting programs satisfying our typing rules can be written in a simple and natural way. In Chapter 5, we discuss how to add more language features, such as data types, references and functions.

We propose a non-constraint-based approach for type inference, which leads to simple proofs, theorems, and accurate, effective error reporting. In the future, we would like to study how our type inference approach works in other larger languages for secure information flow.

In the future, under the condition of keeping the language manageable and provable, we will add more language features to enrich the language's expressiveness. One possible direction is to make our language object oriented by adding classes, objects and inheritance.

In the future, we need to carry on with the implementation of the language and the framework. When a program passes security type checking/inference, the source code should be translated into a commonly used language. A translation module should be developed that accommodate our lenient array operations. As to the target language, C# [35] seems to be the language of choice. C# has assemblies as deployment modules which can be digitally signed and ready support for properties.

In the future, we need to investigate whether it is necessary to develop standardized ways for the core computation to access libraries. We also need to investigate whether we need to standardize user interface, etc.

Our research so far has been driven by examples and real world scenarios. In the future, we need to discover and solve new problems in the development of full applications.

In this dissertation, we identify user information privacy as the scenario in which to apply secure information flow analysis. In the future, we would like to investigate other practical scenarios for possible application.

List of References

[1] D. Denning and P. Denning, "Certification of programs for secure information flow," *Communications ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[2] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 2,3, pp. 167–187, 1996.

[3] A. Sabelfeld and A. C. Myers, "Language-based information flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, pp. 5–19, Jan. 2003.

[4] N. Heintze and J. G. Riecke, "The SLam calculus: programming with secrecy and integrity," in *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998* (ACM, ed.), (New York, NY, USA), pp. 365–377, ACM Press, 1998.

[5] S. Zdancewic and A. Myers, "Secure information flow via linear continuations," 2002.

[6] S. Zdancewic and A. C. Myers, "Secure information flow and CPS," *Lecture Notes in Computer Science*, vol. 2028, pp. 46–61, 2001.

[7] V. Simonet, "Fine-grained information flow analysis for a lambda-calculus with sum types."

[8] K. Honda, V. Vasconcelos, and N. Yoshida, "Secure information flow as typed process behaviour," in *Proceedings 9th European Symposium on Programming*, vol. 1782 of *Lecture Notes in Computer Science*, pp. 180–199, Apr. 2000.

[9] K. Honda and N. Yoshida, "A uniform type structure for secure information flow," in *Proceedings 29th Symposium on Principles of Programming Languages*, (Portland, Oregon), pp. 81–92, Jan. 2002.

[10] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," *Journal of Computer Security*, vol. 7, no. 2,3, pp. 231–253, 1999.

[11] G. Smith, "A new type system for secure information flow," in *Proceedings 14th IEEE Computer Security Foundations Workshop*, (Cape Breton, Nova Scotia, Canada), pp. 115–125, June 2001.

[12] A. Myers, "JFlow: Practical mostly-static information flow control," in *Proceedings 26th Symposium on Principles of Programming Languages*, (San Antonio, TX), pp. 228–241, Jan. 1999.

[13] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a Java-like language," in *Proceedings 15th IEEE Computer Security Foundations Workshop*, (Cape Breton, Nova Scotia, Canada), pp. 253–267, June 2002.

[14] "Irs e-file available for extension filers through aug. 15 and beyond", Internal Revenue Services. `http://www.irs.gov/newsroom/article/0,,id=141319,00.html`.

[15] "e-file using a computer", Internal Revenue Services. `http://www.irs.gov/efile/article/0,,id=98294,00.html?source=ttcom4home1`.

[16] B. Davey and H. Priestley, *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[17] C. A. Gunter, *Semantics of Programming Languages*. The MIT Press, 1992.

[18] J. Agat, "Transforming out timing leaks," in *Proceedings 27th Symposium on Principles of Programming Languages*, (Boston, MA), pp. 40–53, Jan. 2000.

[19] R. Yocum, "Type checking for secure information flow in a multi-threaded language," Master's thesis, Florida International University, 2002.

[20] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Proceedings 10th IEEE Computer Security Foundations Workshop*, pp. 156–168, June 1997.

[21] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.

[22] Z. Deng and G. Smith, "Lenient array operations for practical secure information flow," in *Proceedings 17th IEEE Computer Security Foundations Workshop*, (Pacific Grove, California), pp. 115–124, June 2004.

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

[24] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM Transactions on Programming Languages and Systems*, vol. 25, pp. 117–158, Jan. 2003.

[25] F. Henglein and J. Rehof, "The complexity of subtype entailment for simple types," in *Logic in Computer Science*, pp. 352–361, 1997.

[26] 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.

[27] V. Simonet, "The Flow Caml system (version 1.00): Documentation and user's manual," tech. rep., Institut National de Recherche en Informatique et en Automatique, July 2003. Available at `http:// cristal.inria.fr/ ~simonet/ soft/ flowcaml/ manual/ index.html`.

[28] 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/`.

[29] S. Tse and G. Washburn, "Cryptographic programming in jif," tech. rep., University of Pennsylvania, 2003. Available at `http://www.cis.upenn.edu/~stse/bank/main.pdf`.

[30] J. Rehof and T. A. Mogensen, "Tractable constraints in finite semilattices," *Science of Computer Programming*, vol. 35, no. 2–3, pp. 191–221, 1999.

[31] V. Simonet, "Flow Caml in a nutshell," in *Proceedings of the first APPSEM-II workshop* (G. Hutton, ed.), (Nottingham, United Kingdom), pp. 152–165, Mar. 2003.

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

[33] 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.

[34] S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers, "Untrusted hosts and confidentiality: Secure program partitioning," 2001.

[35] "MSDN Library", Microsoft Corp. `http://msdn.microsoft.com`.

Appendix A Language Specification

program	$\longrightarrow$	(lattice) decs cmd
lattice	$\longrightarrow$	lattice entry
	$\longrightarrow$	entry
entry	$\longrightarrow$	id = (+ | set) (− | set)
set	$\longrightarrow$	set2
	$\longrightarrow$	
set2	$\longrightarrow$	id
	$\longrightarrow$	id , id
decs	$\longrightarrow$	decs dec
	$\longrightarrow$	dec
dec	$\longrightarrow$	id id
	$\longrightarrow$	id **arr** id id
exp	$\longrightarrow$	id
	$\longrightarrow$	*int*
	$\longrightarrow$	id.**length**
	$\longrightarrow$	id [exp]
	$\longrightarrow$	(exp)
	$\longrightarrow$	exp + exp
	$\longrightarrow$	exp − exp
	$\longrightarrow$	exp * exp
	$\longrightarrow$	exp / exp
	$\longrightarrow$	exp = exp
	$\longrightarrow$	exp < exp
cmd	$\longrightarrow$	id := exp
	$\longrightarrow$	id[exp] := exp
	$\longrightarrow$	**skip**
	$\longrightarrow$	**if** exp **then** (cmd) **else** (cmd)
	$\longrightarrow$	**while** exp **do** (cmd)
	$\longrightarrow$	**allocate** id [exp]
	$\longrightarrow$	cmd ; cmd
	$\longrightarrow$	(cmd)

VITA

ZHENYUE DENG

1974	Born, Inner Mongolia, China
1992-1996	Bachelor of Science Department of Computer Science Xidian University Xi'an, China
1996-2000	Software Engineer Multimedia Institute Xidian University Xi'an, China
2000-2003	Master of Science School of Computer Science Florida International University Miami, Florida

PUBLICATIONS AND PRESENTATIONS

Zhenyue Deng and Geoffrey Smith: "Lenient Array Operations for Practical Secure Information Flow", 17th IEEE Computer Security Foundations Workshop, 2004.

Yanli Sun, Zhenyue Deng, Kalai Mathee and Giri Narasimhan: "Training Set Design for Pattern Discovery with Applications to Protein Mortif Detection", The Computational Systems Bioinformatics Conference, 2004.