

# Lenient Array Operations for Practical Secure Information Flow

Zhenyue Deng      Geoffrey Smith  
School of Computer Science  
Florida International University  
Miami, Florida 33199, USA  
{zdeng01, smithg}@cs.fiu.edu

## Abstract

Our goal in this paper is to make secure information flow typing more practical. We propose simple and permissive typing rules for array operations in a simple sequential imperative language. Arrays are given types of the form  $\tau_1 \text{ arr } \tau_2$ , where  $\tau_1$  is the security class of the array's contents and  $\tau_2$  is the security class of the array's length. To keep the typing rules permissive, we propose a novel, lenient semantics for out-of-bounds array indices. We show that our type system ensures a noninterference property, and we present an example that suggests that it will not be too difficult in practice to write programs that satisfy the typing rules.

## 1. Introduction

The *secure information flow* problem is concerned with preventing information leaks from high-security ( $H$ ) to low-security ( $L$ ) program variables. It has received a great deal of attention in recent years, and many type systems have been developed that provably guarantee various *noninterference* properties; see [8] for a recent survey. But so far this research has had little practical impact. One major practical problem is that these type systems tend to be quite restrictive.

In this paper, 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 [4, page 509]. If array  $a$  is  $L$  and  $\text{secret}$  is  $H$ , then the assignment  $a[\text{secret}] = 1;$  is dangerous; if  $a$  is initially all zero, then after the assignment we can deduce the value of  $\text{secret}$  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 1 gives a Java program that leaks a 10-bit  $\text{secret}$  by turning on each bit of  $\text{leak}$  following an array assignment that throws an exception if the corresponding bit of  $\text{secret}$  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 [1, 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* [4] 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  $L$ .

---

```

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 1. A leak exploiting out-of-bounds array indices**

---

In Jif [6], 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*  $\underline{pc}$  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.

The treatment of arrays in Flow Caml [9] seems to be similar, though we have not found a detailed description of its rules for arrays. A similar strategy is described in Yocum’s unpublished thesis [12]: 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 [10]. This is also like Java’s use of 32-bit two’s complement modular arithmetic, avoiding the need for integer overflow exceptions [2, 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  $\tau_1 \text{ arr } \tau_2$ , where  $\tau_1$  is the security class of the array’s contents and  $\tau_2$  is the security class of its length. Several combinations are useful:  $L \text{ arr } L$  is a completely public array,  $H \text{ arr } L$  is an array whose contents are private but whose length is public, and  $H \text{ arr } H$  is a completely private array.

The rest of the paper is organized as follows. In Section 2, we describe the simple sequential imperative language that we consider, and formally define its lenient semantics for array operations. In Section 3 we present the details of our type system, and in Section 4 we prove that it guarantees a noninterference property. In Section 5, we discuss the behavior of the type system on an example tax calculation program. Finally, Section 6 concludes.

## 2. The Language and its Semantics

Programs are written in the simple imperative language [5], extended with one-dimensional integer arrays. The syn-

tax of the language is as follows:

(phrases)  $p ::= e \mid c$

(expressions)  $e ::= x \mid n \mid x[e] \mid x.\mathbf{length} \mid e_1/e_2 \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 = e_2 \mid \dots$

(commands)  $c ::= x := e \mid x[x_1] := e_2 \mid \mathbf{allocate} \ x[x] \mid \mathbf{skip} \mid \mathbf{if} \ e \ \mathbf{then} \ c_1 \ \mathbf{else} \ c_2 \mid \mathbf{while} \ e \ \mathbf{do} \ c \mid c_1; c_2$

Here metavariable  $x$  ranges over identifiers and  $n$  over integer literals. The expression  $x.\mathbf{length}$  yields the length of array  $x$ , as in Java. The command  $\mathbf{allocate} \ x[x]$  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 [3].)

A program  $c$  is executed under a memory  $\mu$ , which maps identifiers to values. A value is either an integer  $n$  or an array of integers  $\langle n_0, n_1, n_2, \dots, n_{k-1} \rangle$ , 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  $\mu$ . The formal semantics of array expressions and division is given in Figure 2. 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 [5]:

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

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

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

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

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

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

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

$$\frac{(c_1, \mu) \longrightarrow (c'_1, \mu')}{(c_1; c_2, \mu) \longrightarrow (c'_1; c_2, \mu')}$$

In addition, we have new rules for array writes and array allocation; these are given in Figure 3. These rules define a transition relation  $\longrightarrow$  on *configurations*. A configuration is either a pair  $(c, \mu)$  or simply a memory  $\mu$ . In the first case,  $c$  is the command yet to be executed; in the second case, the command has terminated, yielding final memory  $\mu$ . We write  $\longrightarrow^k$  for the  $k$ -fold self composition of  $\longrightarrow$ , and  $\longrightarrow^*$  for the reflexive, transitive closure of  $\longrightarrow$ .

### 3. The Type System

In this section, we extend the typing rules in [11] with new rules for typing array operations. We type arrays using types of the form  $\tau_1 \text{ arr } \tau_2$ ; here  $\tau_1$  describes the contents of the array, and  $\tau_2$  describes its length.

Here are the types used by our type system:

(data types)  $\tau ::= L \mid H$

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

For simplicity, we limit the security classes here to just  $L$  and  $H$ ; it is possible to generalize to an arbitrary lattice of security classes.

Out of the four possible array types ( $H \text{ arr } H$ ,  $H \text{ arr } L$ ,  $L \text{ arr } H$ , and  $L \text{ arr } L$ ),  $L \text{ arr } H$  does not really make sense—the contents of an array implicitly includes the array’s length. So if an array’s contents are  $L$ , then the array’s length must also be  $L$ . We therefore adopt the following constraint globally:

**Global Array Constraint:** In any array type  $\tau_1 \text{ arr } \tau_2$ , 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  $\gamma$  denotes an *identifier typing*, which maps identifiers to phrase types of the form  $\tau \text{ var}$  or  $\tau_1 \text{ arr } \tau_2$ . The typing rules are given in Figure 4 and the subtyping rules 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,  $0 \leq e < x.\mathbf{length}$ . So if  $x : \tau_1 \text{ arr } \tau_2$  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 if  $x : H \text{ arr } L$ , then the command  $x[x_1] := e_2$  can be given type  $H \text{ cmd}$ , which says intuitively that it only assigns to  $H$  variables. This is valid because it does not change the length of  $x$ .

---

(ARR-READ)	$\frac{x \in \text{dom}(\mu), \mu(x) = \langle n_0, \dots, n_{k-1} \rangle, \mu(e) = i, 0 \leq i < k}{\mu(x[e]) = n_i}$
	$\frac{x \in \text{dom}(\mu), \mu(x) = \langle n_0, \dots, n_{k-1} \rangle, \mu(e) = i, (i < 0 \vee i \geq k)}{\mu(x[e]) = 0}$
(GET-LENGTH)	$\frac{x \in \text{dom}(\mu), \mu(x) = \langle n_0, \dots, n_{k-1} \rangle}{\mu(x.\text{length}) = k}$
(DIV)	$\frac{\mu(e_1) = n_1, \mu(e_2) = n_2, n_2 \neq 0}{\mu(e_1/e_2) = \lfloor n_1/n_2 \rfloor}$
	$\frac{\mu(e_1) = n, \mu(e_2) = 0}{\mu(e_1/e_2) = 0}$

---

**Figure 2. Semantics of array expressions and division**

---



---

(UPDATE-ARR)	$\frac{x \in \text{dom}(\mu), \mu(x) = \langle n_0, \dots, n_{k-1} \rangle, \mu(e_1) = i, 0 \leq i < k, \mu(e_2) = n}{(x[e_1] := e_2, \mu) \longrightarrow \mu[x := \langle n_0, \dots, n_{i-1}, n, n_{i+1}, \dots, n_{k-1} \rangle]}$
	$\frac{x \in \text{dom}(\mu), \mu(x) = \langle n_0, \dots, n_{k-1} \rangle, \mu(e_1) = i, (i < 0 \vee i \geq k)}{(x[e_1] := e_2, \mu) \longrightarrow \mu}$
(CALLOC)	$\frac{x \in \text{dom}(\mu), \mu(e) \geq 0}{(\text{allocate } x[e], \mu) \longrightarrow \mu[x := \underbrace{\langle 0, 0, \dots, 0 \rangle}_{\mu(e) \text{ of these}}]}$
	$\frac{x \in \text{dom}(\mu), \mu(e) < 0}{(\text{allocate } x[e], \mu) \longrightarrow \mu[x := \langle \rangle]}$

---

**Figure 3. Semantics of array commands**

---

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

## 4. 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 4.1 (Subject Reduction)** *If  $\gamma \vdash c : \tau \text{ cmd}$  and  $(c, \mu) \longrightarrow (c', \mu')$ , then  $\gamma \vdash c' : \tau \text{ cmd}$ .*

*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:

$$\frac{\begin{array}{l} \gamma \vdash e : \tau' \\ \gamma \vdash c_1 : \tau' \text{ cmd} \\ \gamma \vdash c_2 : \tau' \text{ cmd} \end{array}}{\gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \tau' \text{ cmd}} \quad \frac{\tau' \text{ cmd} \subseteq \tau \text{ cmd}}{\gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \tau \text{ cmd}}$$

Hence, by rule CMD<sup>-</sup>, we must have  $\tau \subseteq \tau'$ . So by SUBSUMP we have  $\gamma \vdash c_1 : \tau \text{ cmd}$  and  $\gamma \vdash c_2 : \tau \text{ cmd}$ . By rule BRANCH,  $c'$  can be either  $c_1$  or  $c_2$ ; therefore, we have  $\gamma \vdash c' : \tau \text{ cmd}$ .

2. Case **while**  $e$  **do**  $c$ . By an argument similar to that used in the previous case, we have  $\gamma \vdash c : \tau \text{ cmd}$ . By rule LOOP,  $c'$  is  $c$ ; **while**  $e$  **do**  $c$ . Hence  $\gamma \vdash c' : \tau \text{ cmd}$  by rule COMPOSE.
3. Case  $c_1; c_2$ . As above, we get  $\gamma \vdash c_1 : \tau \text{ cmd}$  and  $\gamma \vdash c_2 : \tau \text{ cmd}$ . By rule SEQUENCE,  $c'$  is either

---

(R-VAL)	$\frac{\gamma(x) = \tau \text{ var}}{\gamma \vdash x : \tau}$
(SUBSCR)	$\frac{\gamma(x) = \tau_1 \text{ arr } \tau_2, \gamma \vdash e : \tau_3,}{\gamma \vdash x[e] : \tau_1 \vee \tau_3}$
(INT)	$\gamma \vdash n : L$
(QUOTIENT)	$\frac{\gamma \vdash e_1 : \tau, \gamma \vdash e_2 : \tau}{\gamma \vdash e_1/e_2 : \tau}$
(ASSIGN)	$\frac{\gamma(x) = \tau \text{ var}, \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau \text{ cmd}}$
(ASSIGN-ARR)	$\frac{\gamma(x) = \tau_1 \text{ arr } \tau_2, \gamma \vdash e_1 : \tau_1, \gamma \vdash e_2 : \tau_1}{\gamma \vdash x[e_1] := e_2 : \tau_1 \text{ cmd}}$
(LENGTH)	$\frac{\gamma(x) = \tau_1 \text{ arr } \tau_2}{\gamma \vdash x.\text{length} : \tau_2}$
(ALLOCATE)	$\frac{\gamma(x) = \tau_1 \text{ arr } \tau_2, \gamma \vdash e : \tau_2}{\gamma \vdash \text{allocate } x[e] : \tau_2 \text{ cmd}}$
(SKIP)	$\gamma \vdash \text{skip} : H \text{ cmd}$
(IF)	$\frac{\gamma \vdash e : \tau, \gamma \vdash c_1 : \tau \text{ cmd}, \gamma \vdash c_2 : \tau \text{ cmd}}{\gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : \tau \text{ cmd}}$
(WHILE)	$\frac{\gamma \vdash e : \tau, \gamma \vdash c : \tau \text{ cmd}}{\gamma \vdash \text{while } e \text{ do } c : \tau \text{ cmd}}$
(COMPOSE)	$\frac{\gamma \vdash c_1 : \tau \text{ cmd}, \gamma \vdash c_2 : \tau \text{ cmd}}{\gamma \vdash c_1; c_2 : \tau \text{ cmd}}$

**Figure 4. Typing rules**

---

(BASE)	$L \subseteq H$
(CMD <sup>-</sup> )	$\frac{\tau' \subseteq \tau}{\tau \text{ cmd} \subseteq \tau' \text{ cmd}}$
(REFLEX)	$\rho \subseteq \rho$
(TRANS)	$\frac{\rho_1 \subseteq \rho_2, \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3}$
(SUBSUMP)	$\frac{\gamma \vdash p : \rho_1, \rho_1 \subseteq \rho_2}{\gamma \vdash p : \rho_2}$

**Figure 5. Subtyping rules**

---

$c_2$  (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  $\gamma \vdash c_2 : \tau \text{ cmd}$ . For the second case, we have  $\gamma \vdash c'_1 : \tau \text{ cmd}$  by induction; hence  $\gamma \vdash c'_1; c_2 : \tau \text{ cmd}$  by rule COMPOSE.

□

**Definition 4.1** *Memory  $\mu$  is consistent with identifier typing  $\gamma$ , written  $\mu : \gamma$ , if  $\text{dom}(\mu) = \text{dom}(\gamma)$  and, for every  $x$ ,  $\mu(x)$  is an integer  $n$  if  $\gamma(x) = \tau \text{ var}$  and  $\mu(x)$  is an array of integers  $\langle n_0, \dots, n_{k-1} \rangle$  if  $\gamma(x) = \tau_1 \text{ arr } \tau_2$ .*

**Lemma 4.2 (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  $e$ . □

**Lemma 4.3 (Progress)** *If  $\gamma \vdash c : \tau \text{ cmd}$  and  $\mu : \gamma$ , then there is a unique configuration  $C$ , of the form  $(c', \mu')$  or just  $\mu'$ , such that  $(c, \mu) \longrightarrow C$  and  $\mu' : \gamma$ .*

*Proof.* By induction on the structure of  $c$ . □

From the Subject Reduction and Progress lemmas, it follows that if command  $c$  is well typed under  $\gamma$  and  $c$  is executed in a memory  $\mu$  consistent with  $\gamma$ , 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 4.4** *If  $(c_1; c_2, \mu) \longrightarrow^j \mu'$ , then there exist  $k$  and  $\mu''$  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  $\mu''$  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  $\mu_1$  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  $\mu''$  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 changing the initial values of  $H$  variables cannot affect the final values of  $L$  variables. (Note however that under typing rule WHILE, changing the initial values of  $H$  variables *can* affect the termination of  $c$ .)

**Definition 4.2** *Memories  $\mu$  and  $\nu$  are equivalent with respect to  $\gamma$ , written  $\mu \sim_\gamma \nu$ , if*

- $\mu$  and  $\nu$  are both consistent with  $\gamma$ ,
- $\mu$  and  $\nu$  agree on all  $L$  variables,
- $\mu$  and  $\nu$  agree on all arrays of type  $L \text{ arr } L$ , and
- $\mu$  and  $\nu$  agree on the length (but not necessarily on the contents) of all arrays of type  $H \text{ arr } L$ .

**Lemma 4.5 (Simple Security)** *If  $\gamma \vdash e : L$  and  $\mu \sim_\gamma \nu$ , then  $\mu(e) = \nu(e)$ .*

*Proof.* By induction on the structure of  $e$ :

1. Case  $x$ . By typing rule R-VAL and  $\gamma \vdash x : L$ ,  $\gamma(x) = L \text{ var}$ . Therefore,  $\mu(x) = \nu(x)$ .
2. Case  $x[e]$ . By typing rule SUBSCR and  $\gamma \vdash x[e] : L$ , we have  $\gamma(x) = L \text{ arr } L$  and  $\gamma \vdash e : L$ . By the definition of memory equivalence, we have  $\mu(x) = \nu(x)$ . By induction, we have  $\mu(e) = \nu(e)$ . Then, by semantic rule ARR-READ, we have  $\mu(x[e]) = \nu(x[e])$ .
3. Case  $x.\text{length}$ . By typing rule LENGTH and  $\gamma \vdash x.\text{length} : L$ , we have  $\gamma(x) = \tau \text{ arr } L$ , for some  $\tau$ . By memory equivalence, we have  $\mu(x.\text{length}) = \nu(x.\text{length})$ .
4. Case  $e_1/e_2$ . By typing rule QUOTIENT, we have  $\gamma \vdash e_1 : L$  and  $\gamma \vdash e_2 : L$ . 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. Other cases are similar.

□

**Lemma 4.6 (Confinement)** *If  $\gamma \vdash c : H \text{ cmd}$  and  $(c, \mu) \longrightarrow (c', \mu')$  (or  $(c, \mu) \longrightarrow \mu'$ ), then  $\mu \sim_\gamma \mu'$ .*

*Proof.* By induction on the structure of  $c$ :

1. Case  $x := e$ . Here we have  $\gamma(x) = H \text{ var}$ , so  $\mu \sim_\gamma \mu[x := \mu(e)]$ .
2. Case  $x[e_1] := e_2$ . Here we have  $\gamma(x) = H \text{ arr } H$  or  $\gamma(x) = H \text{ arr } L$ . Hence by rule UPDATE-ARR we have  $\mu \sim_\gamma \mu'$ .
3. Case **allocate**  $x[e]$ . Here we have  $\gamma(x) = H \text{ arr } H$ , so by rule CALLOC,  $\mu \sim_\gamma \mu'$ .
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  $c_1; c_2$ . Follows by induction.

□

**Corollary 4.7** *If  $\gamma \vdash c : H \text{ cmd}$  and  $(c, \mu) \longrightarrow^* \mu'$ , then  $\mu \sim_\gamma \mu'$ .*

*Proof.* Follows inductively from Subject Reduction and Confinement. □

**Theorem 4.8 (Noninterference)** *Suppose that  $\gamma \vdash c : \tau \text{ cmd}$ ,  $\mu \sim_{\gamma} \nu$ ,  $\mu : \gamma$ , and  $\nu : \gamma$ . If  $(c, \mu) \longrightarrow^* \mu'$  and  $(c, \nu) \longrightarrow^* \nu'$ , then  $\mu' \sim_{\gamma} \nu'$ .*

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

1. Case  $x := e$ . By rule UPDATE, we have  $\mu' = \mu[x := \mu(e)]$  and  $\nu' = \nu[x := \nu(e)]$ . Since  $c$  is well typed, we have  $x \in \text{dom}(\gamma)$ . If  $\gamma(x) = L \text{ var}$  then by rule ASSIGN, we have  $\gamma \vdash e : L$ . So by Simple Security,  $\mu(e) = \nu(e)$ , and so  $\mu[x := \mu(e)] \sim_{\gamma} \nu[x := \nu(e)]$ . If, instead,  $\gamma(x) = H \text{ var}$  then trivially  $\mu[x := \mu(e)] \sim_{\gamma} \nu[x := \nu(e)]$ .
2. Case **skip**. The result follows immediately from rule NO-OP.
3. Case  $c_1; c_2$ . If  $(c_1; c_2, \mu) \longrightarrow^j \mu'$  then by Lemma 4.4 there exist  $k$  and  $\mu''$  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  $\nu''$  such that  $0 < k' < j'$ ,  $(c_1, \nu) \longrightarrow^{k'} \nu''$  and  $(c_2, \nu'') \longrightarrow^{j'-k'} \nu'$ . By induction,  $\mu'' \sim_{\gamma} \nu''$ . So by induction again,  $\mu' \sim_{\gamma} \nu'$ .
4. Case **if  $e$  then  $c_1$  else  $c_2$** . If  $\gamma \vdash e : L$  then  $\mu(e) = \nu(e)$  by Simple Security. If  $\mu(e) \neq 0$  then the two executions have the form

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

and

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

By induction,  $\mu' \sim_{\gamma} \nu'$ . The case when  $\mu(e) = 0$  is similar.

If instead  $\gamma \not\vdash e : L$  then by rule IF, and the fact that  $c$  is well typed, we have  $\gamma \vdash \text{if } e \text{ then } c_1 \text{ else } c_2 : H \text{ cmd}$ . Then by the corollary to Confinement,  $\mu \sim_{\gamma} \mu'$  and  $\nu \sim_{\gamma} \nu'$ . So  $\mu' \sim_{\gamma} \nu'$ .

5. Case **while  $e$  do  $c_1$** . Similar to the **if** case.
6. Case **allocate  $x[e]$** . We consider the possible types of  $x$ .
  - If  $\gamma(x) = \tau \text{ arr } L$ , then by rule ALLOCATE we have  $\gamma \vdash e : L$ . By Simple Security, we have  $\mu(e) = \nu(e)$ . So, by rule CALLOC,  $\mu'(x) = \nu'(x)$ . So,  $\mu' \sim_{\gamma} \nu'$ .
  - If  $\gamma(x) = H \text{ arr } H$ , by rule ALLOCATE we have  $\gamma \vdash \text{allocate } x[e] : H \text{ cmd}$ . By the corollary to Confinement, we have  $\mu' \sim_{\gamma} \nu'$ .
7. Case  $x[e_1] := e_2$ . We consider the possible types of  $x$ .
  - If  $\gamma(x) = L \text{ arr } \tau$ , then by rule ASSIGN-ARR we have  $\gamma \vdash e_1 : L$  and  $\gamma \vdash e_2 : L$ . By Simple Security, we have  $\mu(e_1) = \nu(e_1)$  and  $\mu(e_2) = \nu(e_2)$ . So, by rule UPDATE-ARR,  $\mu'(x) = \nu'(x)$ . So  $\mu' \sim_{\gamma} \nu'$ .

If  $\gamma(x) = H \text{ arr } H$ , then by rule ASSIGN-ARR we have  $\gamma \vdash x[e_1] := e_2 : H \text{ cmd}$ . By the corollary to Confinement, we have  $\mu' \sim_{\gamma} \nu'$ .

□

## 5. 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

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

and then returning either `singleTaxTable[b]` or `marriedTaxTable[b]`, depending on the marital status.

With respect to our type system, we want the typings `brackets : L arr L`, `singleTaxTable : L arr L`, and `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 `taxableIncome : H arr L` and `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 6. 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 [1] 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 [6] 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 `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 in Figure 7. This technique allows us to achieve an approximation to our lenient semantics within Java, though at the cost of some syntactic clumsiness.

## 6. Conclusion

Because of our lenient execution model and our array types of the form  $\tau_1 \text{ arr } \tau_2$ , 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 the future, it would be good to develop techniques for automatically inferring the classes of auxiliary variables, as in the tax calculation example. More generally, more effort should be devoted to making secure information flow analysis practical. Especially important is to better understand how to account for *controlled declassification*, which is needed in situations where noninterference is too strong; the approach of Myers, Sabelfeld, and Zdancewic [7] seems promising.

## 7. Acknowledgments

This work was partially supported by the National Science Foundation under grants CCR-9900951 and HRD-0317692.

## References

- [1] J. Agat. Transforming out timing leaks. In *Proceedings 27th Symposium on Principles of Programming Languages*, pages 40–53, Boston, MA, Jan. 2000.
- [2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language, Third Edition*. Addison-Wesley, 2000.
- [3] 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*, pages 253–267, Cape Breton, Nova Scotia, Canada, June 2002.
- [4] D. Denning and P. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
- [5] C. A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1992.
- [6] A. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings 26th Symposium on Principles of Programming Languages*, pages 228–241, San Antonio, TX, Jan. 1999.
- [7] A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. In *Proceedings 17th IEEE Computer Security Foundations Workshop*, Pacific Grove, California, June 2004.
- [8] A. Sabelfeld and A. C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

---

```

// 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 6. Tax calculation program**

---

```

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

```

**Figure 7. Approximating lenient semantics using try-catch**

---

- [9] V. Simonet. The Flow Caml system (version 1.00): Documentation and user's manual. Technical report, Institut National de Recherche en Informatique et en Automatique, July 2003. Available at <http://crystal.inria.fr/~simonet/soft/flowcaml/manual/index.html>.
- [10] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proceedings 10th IEEE Computer Security Foundations Workshop*, pages 156–168, June 1997.
- [11] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2,3):167–187, 1996.
- [12] R. Yocum. Type checking for secure information flow in a multi-threaded language. Master's thesis, Florida International University, 2002.