**ISSN** 1746-7659, England, UK
**Journal of Information and Computing Science**
Vol. 1, No. 2, 2006, *pp*. 67-77

# Discretional Array Operations for Secure Information Flow

Jianbo Yao [+], Jianshi Li

Faculty of Computer Science and Engineering, Guizhou University, Guiyang, Guizhou, 550025, China

( *Received December 26,2006, accepted March 16, 2006* )

**Abstract.** Arrays exist in many nontrivial programs. Array operations can cause subtle information leaks. This paper allows array as first-class value and regards discretional array as array of array by alias array. Arrays are given types of the form $\tau_1$ alias $\tau_2$, where $\tau_1$ is the security class of the array and $\tau_2$ is the security class of the array's alias. To distinguish array from its alias, we propose a novel binary memory model $[\mu_1;\mu_2]$. The soundness of our type system is proved by noninterference property.

**Keywords:** alias array, discretional array , secure information flow, type system.

## 1. Introduction

The static analysis of secure information flow has been studied for many years [1]. The analysis of secure information flow is that checks whether the information flow of a given program is secure. Secure information flow ensures the noninterference, that is, observations of the initial and final values of low-security class $(L)$ program variables do not provide any information about the initial value of high-security class $(H)$ program variables.

For example, in the assignment statement $y := x$, there is an information flow from $x$ to $y$. If the security class of $x$ is $L$ or $H$ and the security class of $y$ is $H$, then the information flow $x$ to $y$ is secure. If $x$ is $H$ and $y$ is $L$, then the information flow $x$ to $y$ is insecure.

In some early work on secure information flow, Denning et al. [2,3,4] proposed a static certification method for verifying the secure flow of information through a program. Program variables are assigned a security class and security classes are assumed to form a lattice structure, ordered by $\leq$. Each variable has a static security class binding that can be determined from the declarations in the program.

Later, Vopano et al. [5,6] developed an elegant syntax-directed type system for annotating program variables, commands, and procedure parameters with security levels. They also proved that their type system ensures noninterference. Geoffrey Smith [7] extend the analysis to deal with a a multi-threaded imperative language. The type system is insufficient to ensure noninterference.

Benjamin C. Pierce [8] presents the definition of type system. "A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." Type system can be used in secrecy analysis.

Banerjee and Naumann [9] extend the type system of Volpano et al. to support a more realistic object oriented sequential language. They proved noninterference for a language with pointers, mutable state, private fields, class-based visibility, dynamic binding and inheritance, type casts, type tests, and mutually recursive classes and methods.

Amer Diwan et al. [10] use programming language types to disambiguate memory reference. They uses type compatibility to determine aliases.

Many type systems of secure information flow have been developed, of special mention Deng and Smith's type system focused on one-dimensional integer array operations for practical secure information

[+] Corresponding author. Tel.: +86-851-3627946; fax: +86-851-3627946.
*E-mail address*: jianbo-yao@21cn.com

flow [11]. Arrays are given types of the form $\tau_1$ 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. The type system ensures noninterference property. In their system, a multidimensional-array isn't regarded as array of array, for they do not treat arrays as first-class values for simplicity.

Arrays play a major role in many nontrivial programs, but array operations can cause information leaks. For example, the program

```
$i = 0;$
int $a_0[5] = \{0,1,2,3,4\};$
int $a_1[5] = \{5,6,7,8,9\};$
int $a[2];$
$while\ (i < 2)\{$
        $\&x_i = a_i[5];$
        $a[i] = x_i;$
            $i = i + 1;$
        $\}$
```

Here, $x$ denotes an alias of the one-dimensional array $a_i[5]$. There are some information flows [4] from $a_i[5]$ to the alias $x_i$. Hence if $a_i[5]$ is $H$, then we must make the alias $x_i$ be $H$ as well.

In our system, arrays are treated as first-class values and a discretional array may be regarded as array of array. arrays are given types of the form $\tau_1$ alias $\tau_2$; where $\tau_1$ is the security class of the array and $\tau_2$ is the security class of its alias. Several combinations are useful: $L$ alias $L$ is an array whose both contents and its alias are private, $H$ alias $H$ is an array whose both contents and its alias are public.

The remainder of the paper is organized as follows. In Section 2, we describe the simple sequential imperative language that we consider, and formally define its 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. Finally, Section 5 gives our conclusions.

## 2. Syntax and Semantics

Our typed language is the simple imperative language[4] with arrays. The syntax of the language is as follows:

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

(expressions) $e ::= x \mid n \mid x[e] \mid \&x \mid e_1 \div e_2 \mid e_1 * e_2 \mid e_1 \vee e_2 \mid e_1 \wedge e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \cdots$

(commands) $c ::= x := e \mid x[e_1] := e_2 \mid allocate\ x[e] \mid \&x := x[e] \mid x[e_1] := \&x \mid skip \mid$
$if\ e\ then\ c_1\ do\ c_2 \mid while\ e\ do\ c \mid c_1; c_2$

Here, meta-variable $x$ ranges over identifiers and $n$ over integer literals. 0 stands for false and 1 for true. The expression $x[e]$ is one-dimensional array. The expression $\&x$ is an alias of an array. The command $allocate\ x[e]$ allocates a 0-initialized block of memory for one-dimensional array $x[e]$; the size of the array is given by $e$. The command $\&x := x[e]$ declares that $\&x$ is an alias of the array $x[e]$. The command $x[e_1] := \&x$ initializes the elements of an array by alias.

A program $c$ is executed under a memory, which maps identifiers to values. A value is either an integer

$n$ a multidimensional-array $[x_0, x_1, x_2, \ldots, x_{k-1}]$ ,where $k \geq 0$ , $x_0, x_1, x_2, \ldots, x_{k-1}$ are aliases of inferior-dimensional-array. An array and its alias have same address, so two identifiers a and b can point to the same block of memory. A binary memory $[\mu_1; \mu_2]$ is needed for distinguished them. A program $c$ is executed under the binary memory $[\mu_1; \mu_2]$. $\mu_1$ maps identifiers to values. A value is either an integer $n$ or a multidimensional-array $[x_0, x_1, x_2, \ldots, x_{k-1}]$ ,where $k \geq 0$ , $x_0, x_1, x_2, \ldots, x_{k-1}$ are aliases of inferior-dimensional-array. $\mu_2$ maps aliases identifiers to values.

The formal semantics of commands and expressions is given by a standard structural operational semantics. See Figure 1.

(UPDATE)
$$\frac{x \in dom(\mu_1)}{(x := e, [\mu_1; \mu_2]) \Rightarrow [\mu_1[x := \mu_1(e)]; \mu_2]}$$

(NO-OP)
$$(skip, [\mu_1; \mu_2]) \Rightarrow [\mu_1; \mu_2]$$

(BRANCH)
$$\frac{\mu_1(e) = 1}{(\text{if } e \text{ then } c_1 \text{ else } c_2, [\mu_1; \mu_2]) \Rightarrow (c_1, [\mu_1; \mu_2])}$$

$$\frac{\mu_1(e) = 0}{(\text{if } e \text{ then } c_1 \text{ else } c_2, [\mu_1; \mu_2]) \Rightarrow (c_2, [\mu_1; \mu_2])}$$

(LOOP)
$$\frac{\mu_1(e) = 0}{(\text{while } e \text{ do } c, [\mu_1; \mu_2]) \Rightarrow [\mu_1; \mu_2]}$$

$$\frac{\mu_1(e) = 1}{(\text{while } e \text{ do } c, [\mu_1; \mu_2]) \Rightarrow (c; \text{while } e \text{ do } c, [\mu_1; \mu_2])}$$

(SEQUENCE)
$$\frac{(c_1, [\mu_1; \mu_2]) \Rightarrow [\mu_1'; \mu_2]}{(c_1; c_2, [\mu_1; \mu_2]) \Rightarrow (c_2, [\mu_1'; \mu_2])}$$

$$\frac{(c_1, [\mu_1; \mu_2]) \Rightarrow (c_1', [\mu_1'; \mu_2])}{(c_1; c_2, [\mu_1; \mu_2]) \Rightarrow (c_1'; c_2, [\mu_1'; \mu_2])}$$

Figure1. The formal semantics of commands and expressions

The formal semantics of array commands and expressions are given in Figure 2. We prescribe an array indexes with an out-of-bounds indices yields 0 and an array initializes with an out-of-bounds indices is skiped.

(ARR-INDEX)
$$\frac{x \in dom(\mu_1), [\mu_1; \mu_2](x) = [[x_0, \ldots, x_{k-1}]; \mu_2], \mu_1(e) = i, 0 \leq i < k}{[\mu_1; \mu_2](x[e]) = [x_i; \mu_2]}$$

$$\frac{x \in dom(\mu_1), [\mu_1; \mu_2](x) = [[x_0, \ldots, x_{k-1}]; \mu_2], \mu_1(e) = i, i < 0 \vee i \geq k}{\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}(x[e]) = [0; \mu_2]}$$

(DIV)
$$\frac{\mu(e_1)=n,\mu(e_2)=0}{\mu(e_1 \div e_2)=0}$$

$$\frac{\mu(e_1)=n_1,\mu(e_2)=n_2,n\neq 0}{\mu(e_1 \div e_2)=[n_1 \div n_2]}$$

(UPDATE-ARR)
$$\frac{x\in dom(\mu_1),\ [\mu_1;\mu_2](x)=\big[[x_0,\ldots,x_{k-1}];\mu_2\big],\mu_1(e)=i,\ 0\le i<k}{\big(x[e]=e_1,[\mu_1;\mu_2]\big)\Rightarrow\big[\mu_1\big[x_i:=\mu_1(e_1)\big];\mu_2\big]}$$

$$\frac{x\in dom(\mu_1),[\mu_1;\mu_2](x)=\big[[x_0,\ldots,x_{k-1}];\mu_2\big],\mu_1(e)=i,\ i<0\vee i\ge k}{\big(x[e]=e_1,[\mu_1;\mu_2]\big)\Rightarrow[\mu_1;\mu_2]}$$

(CALLOC)
$$\frac{x\in dom(\mu_1),\mu_1(e)\ge 0}{\big(\text{allocat}\,x[e],[\mu_1;\mu_2]\big)\Rightarrow\left[\mu_1\left\langle x:=\underbrace{[0,0,\ldots,0]}_{\mu_1(e)\text{ of these}}\right\rangle;\mu_2\right]}$$

$$\frac{x\in dom(\mu_1),\mu_1(e)<0}{\big(\text{allocat}\,x[e],[\mu_1;\mu_2]\big)\Rightarrow\big[\mu_1\langle x:=0\rangle;\mu_2\big]}$$

(CALIAS)
$$\frac{x\in dom(\mu_1),[\mu_1;\mu_2](x)=\big[[x_0,\ldots,x_{k-1}];\mu_2\big]}{\big(\&x:=x[e],[\mu_1;\mu_2]\big)\Rightarrow\big[[x_0,\ldots,x_{k-1}];[x_0,\ldots,x_{k-1}]\big]}$$

$$\frac{\begin{array}{l}x\in dom(\mu_1),\&x\in dom(\mu_1),[\mu_1;\mu_2](x)=\big[[x_0,\ldots,x_{k-1}];\mu_2\big],\\ [\mu_1;\mu_2](\&x)=\big[\mu_1;[x_0,\ldots,x_{k-1}]\big],\\ \mu_1(e)=i,0\le i\le k\end{array}}{\big(x[e]:=\&x,[\mu_1;\mu_2]\big)\Rightarrow\big[\mu_1\langle x_i:=x\rangle;[x_0,\ldots,x_{k-1}]\big]}$$

$$\frac{\begin{array}{l}x\in dom(\mu_1),\&x\in dom(\mu_2),[\mu_1;\mu_2](x)=\big[[x_0,\ldots,x_{k-1}];\mu_2\big],\\ [\mu_1;\mu_2](\&x)=\big[\mu_1;[x_0,\ldots,x_{k-1}]\big],\mu_1(e)=i,0\le i\le k\end{array}}{\big(x[e]:=\&x,[\mu_1;\mu_2]\big)\Rightarrow\big[\mu_1\langle x_i:=x\rangle;[x_0,\ldots,x_{k-1}]\big]}$$

Figure2. The formal semantics of array commands and expressions

## 3. The Type System

In this section, we extend the typing rules in [5] with new rules for typing array operations. We type array using a type of the form $\tau_1$ *alias* $\tau_2$; where $\tau_1$ is the security class of the array and $\tau_2$ is the security class of its alias.

Here are the types used by our system:

| | |
|---|---|
| (data types) | $\tau ::= L \mid H$ |
| ( phrase types ) | $\rho ::= \tau \mid \tau_{\text{var}} \mid \tau_{cmd} \mid \tau_1$ *alias* $\tau_2$ |

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. Type $\tau_{\text{var}}$ is the type of a variable and $\tau_{cmd}$ is the type of a command.

All of the four possible array types ($H$ *alias* $H$, $H$ *alias* $L$, $L$ *alias* $H$, $L$ *alias* $L$), $L$ *alias* $H$ do not really make sense. Although an $L$-array can be aliased by $L$-variable and $H$-variable, a type $L$-array never is aliased an $H$-variable[9]. So if an array is of type $L$, then its alias must be $L$. $H$ *alias* $L$ do not really make sense too. There is an information flow [4] from an array to its alias. Hence if an array is of type $H$, then its alias must be $H$. We therefore adopt the following constraint globally:

**Global Array Constraint: In any array type $\tau_1$ *alias* $\tau_2$, we require that $\tau_2 = \tau_1$.**

We can now present our type system formally. It allows us to prove typing judgments of the form $\gamma|-p:\rho$ as well as subtyping judgments of the form $\rho \subseteq \rho'$. Here $\gamma$ denotes an identifier typing, which maps identifiers to phrase types of the form $\tau_{\text{var}}$ or $\tau_1$ *alias* $\tau_2$. The typing judgment $\gamma|-p:\rho$ means that phrase $p$ has type $\rho$, assuming $\gamma$ prescribes types for any free identifiers in $p$. The subtype relation is antimonotonic in the types of commands, meaning that if $\tau \subseteq \tau'$ then $\tau'_{cmd} \subseteq \tau_{cmd}$. As usual, there is a type coercion rule that allows a phrase of type $\rho$ to be assigned a type $\rho'$ whenever $\rho \subseteq \rho'$.

The typing rules are given in Figure 3 and the subtyping rules in Figure 4.

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

(ARRAY) $$\frac{\gamma(x) = \tau_1 \ alias \ \tau_2}{\gamma|-x[e]:\tau_1}$$

(INT) $$\gamma|-n:L$$

(QUOTIENT) $$\frac{\gamma|-e_1:\tau, \gamma|-e_2:\tau}{\gamma|-e_1 \div e_2:\tau}$$

(ADDITION) $$\frac{\gamma|-e_1:\tau, \gamma|-e_2:\tau}{\gamma|-e_1 + e_2:\tau}$$

(ASSIGN) $$\frac{\gamma(x) = \tau_{\text{var}}, \gamma|-e:\tau}{\gamma|-x := e:\tau_{cmd}}$$

(ASSIGN-ARR) $$\frac{\gamma(x) = \tau_1 \ alias \ \tau_2 \ , \gamma|-e_1:\tau_1, \gamma|-e_2:\tau_1}{\gamma|-x[e_1] := e_2:\tau_{1cmd}}$$

(ALLOCATE) $$\frac{\gamma(x) = \tau_1 \ alias \ \tau_2 \ , \gamma|-e:\tau_1}{\gamma|-allocate \ x[e]:\tau_{1cmd}}$$

(ALIAS) $$\frac{\gamma(x) = \tau_1 \ alias \ \tau_2}{\gamma|-\&x:\tau_2}$$

$$\frac{\gamma(x) = \tau_1 \ alias \ \tau_2}{\gamma|-\&x := x[e]:\tau_{2cmd}}$$

$$\frac{\gamma(x) = \tau_1 \ alias \ \tau_2}{\gamma|-x[e_1] := \&x:\tau_{1cmd}}$$

(SKIP) $$\gamma|-skip:H_{cmd}$$

(IF) $$\frac{\gamma|-e:\tau, \gamma|-c_1:\tau_{cmd}, \quad \gamma|-c_2:\tau_{cmd}}{\gamma|-\text{if } e \text{ then } c_1 \text{ else } c_2:\tau_{cmd}}$$

(WHILE) $$\frac{\gamma|-e:\tau, \gamma|-c:\tau_{cmd}}{\gamma|-\text{while } e \text{ do } c:\tau_{cmd}}$$

(COMPOSE)
$$\frac{\gamma|-c_1 : \tau_{cmd},\quad \gamma|-c_2 : \tau_{cmd}}{\gamma|-c_1; c_2 : \tau_{cmd}}$$

Figure 3. Typing Rules

(BASE) $L \subseteq H$

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

(REFLEX) $\rho \subseteq \rho$

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

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

Figure 4. Subtyping Rules

Now, we discuss the array typing rules. In rule ARRAY, the security class of expression $x[e]$ depends on the contents of the array $x$ as well as on the alias of the array. Given the Global Array Constraint, this simplifies to $\tau_1$.

For rule ASSIGN-ARR, if $x : H\ alias\ L$, then the command $x[e_1] := e_2$ can be given type $H_{cmd}$, which intuitively says that it only assigns to $H$ variables. If $x : L\ alias\ L$, then the command $x[e_1] := e_2$ can be given type $L_{cmd}$, which intuitively says that it only assigns to $L$ variables. These are valid because they do change the alias of the array $x[e]$.

For rule ALLOCATE, the command allocate $x[e]$ do assign a length of index to the array $x$, and every element of the array $x[e]$ can later be initialized by alias $\&x$.

For rule ALIAS, $\&x$ is an alias of $x$; If $x : \tau_1\ alias\ \tau_2$, then $\&x := x[e] : \tau_{2cmd}$. If $x : \tau_1\ alias\ \tau_2$, then $x[e_1] := \&x : \tau_{1cmd}$.

## 4. Type Soundness as Noninterference

In this section, we establish the semantics soundness of our type system by proving a noninterference theorem.

**Definition 4.1:** Given an identifier typing $\gamma$, if $[\mu_1;\mu_2],[\nu_1;\nu_2]$ and $\gamma$ have the same domain and $[\mu_1;\mu_2]$ and $[\nu_1;\nu_2]$ agree on all $L$ identifiers, then memories $[\mu_1;\mu_2]$ and $[\nu_1;\nu_2]$ are equivalent. This is written $[\mu_1;\mu_2] \sim_\gamma [\nu_1;\nu_2]$.

**Lemma 4.1** (Simple Security): If $\gamma|-e : L$ and $[\mu_1;\mu_2] \sim_\gamma [\nu_1;\nu_2]$, then $[\mu_1;\mu_2](e) = [\nu_1;\nu_2](e)$.

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

1. The case $x$. By typing rule R-VAL and $\gamma|-x : L$, $\gamma(x) = L_{\text{var}}$. We have $[\mu_1;\mu_2](x) = [\nu_1;\nu_2](x)$.

2. The case $x[e]$. By typing rule ARRAY and $\gamma|-x[e] : L$, we have $\gamma(x) = L\ alias\ L$, $\gamma|-e : L$, for some $\tau$. By the definition of memory equivalence, we have $[\mu_1;\mu_2](x[e]) = [\nu_1;\nu_2](x[e])$.

3. The case $\&x$. By typing rule ALIAS and $\gamma|-\&x:L$, we have $\gamma(x)=\tau\ alias\ L$, for some $\tau$. By memory equivalence, we have $[\mu_1;\mu_2](\&x)=[\nu_1;\nu_2](\&x)$.

4. The case $e_1 \div e_2$. By typing rule QUOTIENT and $\gamma|-e_1:L$, $\gamma|-e_2:L$, we have $\gamma|-e_1\div e_2:L$, $[\mu_1;\mu_2](e_1)=[\nu_1;\nu_2](e_1)$ and $[\mu_1;\mu_2](e_2)=[\nu_1;\nu_2](e_2)$, then

$$[\mu_1;\mu_2](e_1\div e_2)=[\nu_1;\nu_2](e_1\div e_2).$$

5. The case $e_1+e_2$. By typing rule ADDITION and $\gamma|-e_1:L$, $\gamma|-e_2:L$, we have $\gamma|-e_1+e_2:L$, $[\mu_1;\mu_2](e_1)=[\nu_1;\nu_2](e_1)$ and $[\mu_1;\mu_2](e_2)=[\nu_1;\nu_2](e_2)$, then

$$[\mu_1;\mu_2](e_1+e_2)=[\nu_1;\nu_2](e_1+e_2)..$$

6. The other cases $e_1\times e_2, e_1\vee e_2$, $e_1\wedge e_2, e_1=e_2$, $e_1\neq e_2, e_1<e_2$.

Each one of them is similar to The case $e_1\div e_2$ and The case $e_1+e_2$.

**Lemma 4.2** (Confinement)**:** If $\gamma|-c:H_{cmd}$ and $(c,[\mu_1;\mu_2])\rightarrow(c',[\mu_1';\mu_2'])$ or $(c,[\mu_1;\mu_2])\rightarrow[\mu_1';\mu_2']$, then $[\mu_1;\mu_2]\sim_\gamma[\mu_1';\mu_2']$.

Proof. This proof is by induction on the structure of the commands $c$:

1. The case $x:=e$. Here we have $\gamma(x)=H_{\text{var}}$, so $[\mu_1;\mu_2]\sim_\gamma[\mu_1[x:=\mu(e)];\mu_2]$.

2. The case $x[e_1]:=e_2$. Here we have $\gamma(x)=H\ alias\ H$. Hence by rule UPDATE-ARR we have $[\mu_1;\mu_2]\sim_\gamma[\mu_1';\mu_2']$.

3. The case $allocate\ x[e]$.

Here we have $\gamma(x)=H\ alias\ H$, so by rule CALLOCATE, we have $[\mu_1;\mu_2]\sim_\gamma[\mu_1';\mu_2']$.

4. The case $\&x:=x[e]$. Here we have $\gamma(x)=H\ alias\ H$. Hence by rule CALIAS, we have $[\mu_1;\mu_2]\sim_\gamma[\mu_1';\mu_2']$.

5. The case $x[e_1]:=\&x$. Here we have $\gamma(x)=H\ alias\ H$. Hence by rule CALIAS, we have $[\mu_1;\mu_2]\sim_\gamma[\mu_1';\mu_2']$.

6. The case $skip$, if $e$ then $c_1$ else $c_2$, and while $e$ do $c$. These cases are trivial, because $[\mu_1;\mu_2]=[\mu_1';\mu_2']$.

7. The case $c_1;c_2$.

The command $c_1;c_2$ is the composition of two commands $c_1$ and $c_2$. The lemma follows directly by induction.

**Lemma 4.3** (Subject Reduction)**:** If $\gamma|-c:\tau_{cmd}$ and $(c,[\mu_1;\mu_2])\rightarrow(c',[\mu_1';\mu_2'])$, then $\gamma|-c:\tau_{cmd}'$.

**Proof.** This proof is by induction on the structure of the commands $c$:

1. The case $c_1;c_2$.

If $c$ is of the form $c_1;c_2$, then it follows that $\gamma|-c_1:\tau_{cmd}$ and $\gamma|-c_2:\tau_{cmd}$. By rule SEQUENCE, $c'$ is either $c_2$ or else $c_1';c_2$, where $(c_1,[\mu_1;\mu_2])\rightarrow(c_1',[\mu_1';\mu_2'])$. For the first case, we have $\gamma|-c_2:\tau_{cmd}$. For the second case, we have $\gamma \vdash c_1 : \tau_{cmd}$ hence by induction we have $\gamma \vdash c_1^{'}; c_2 : \tau_{cmd}$ by rule COMPOSE.

2. The case $\text{while } e \text{ do } c$

If $c$ is the form of $\text{while } e \text{ do } c$, then $\tau$ must be $L$, we must have $\gamma \vdash c : L_{cmd}$, and so $\gamma \vdash c; \text{while } e \text{ do } c : L_{cmd}$.

3. The case of $\text{if } e \text{ then } c_1 \text{ else } c_2$.

The case of $\text{if } e \text{ then } c_1 \text{ else } c_2$ is similar to the case $\text{while } e \text{ do } c$..

**Lemma 4.4:** If $(c_1; c_2, [\mu_1; \mu_2]) \to^k [\mu_1^{'}; \mu_2^{'}]$, then there exist $0 < j < k$ and $[\mu_1^{''}; \mu_2^{''}]$ such that $(c_1, [\mu_1; \mu_2]) \to^j [\mu_1^{''}; \mu_2^{''}]$, and $(c_2, [\mu_1^{''}; \mu_2^{''}]) \to^{k-j} [\mu_1^{'}; \mu_2^{'}]$. $\to^k$ denotes $k - fold$ self composition of $\to$.

**Proof.** By induction on $k$. If the derivation begins with an application of the first SEQUENCE rule, then there exists $[\mu_1^{''}; \mu_2^{''}]$ such that

$$(c_1, [\mu_1; \mu_2]) \to [\mu_1^{''}; \mu_2^{''}]$$

and

$$(c_1; c_2, [\mu_1; \mu_2]) \to [\mu_1^{''}; \mu_2^{''}] \to^{k-1} [\mu_1^{'}; \mu_2^{'}].$$

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

If the derivation begins with an application of the second SEQUENCE rule, then there exists $c_1^{'}$ and $[\mu_{11}; \mu_{21}]$ such that

$$(c_1, [\mu_1; \mu_2]) \to (c_1^{'}, [\mu_{11}; \mu_{21}])$$

and

$$(c_1; c_2, [\mu_1; \mu_2]) \to (c_1^{'}; c_2, [\mu_{11}; \mu_{21}]) \to^{k-1} [\mu_1^{'}; \mu_2^{'}].$$

By induction, there exists $j$ and $[\mu_1^{''}; \mu_2^{''}]$ such that

$$0 < j < k-1,\ (c_1^{'}, [\mu_{11}; \mu_{21}]) \to^j [\mu_1^{''}; \mu_2^{''}],$$

and

$$(c_2, [\mu_1^{''}; \mu_2^{''}]) \to^{k-1-j} [\mu_1^{'}; \mu_2^{'}].$$

Hence

$$(c_1, [\mu_1; \mu_2]) \to^{j+1} [\mu_1^{''}; \mu_2^{''}]$$

and

$$(c_2, [\mu_1^{''}; \mu_2^{''}]) \to^{k-(j+1)} [\mu_1^{'}; \mu_2^{'}].$$

And $0 < j+1 < k$.

**Theorem 4.6** (Noninterference)**:** Suppose that

$$\gamma \vdash c : \tau_{cmd}\text{, and } [\mu_1; \mu_2] \sim_\gamma [\nu_1; \nu_2] \ .$$

If

$$(c, [\mu_1; \mu_2]) \to^* [\mu_1^{'}; \mu_2^{'}] \text{ and } (c, [\nu_1; \nu_2]) \to^* [\nu_1^{'}; \nu_2^{'}],$$

then

$$\left[\mu_1';\mu_2'\right]\sim_\gamma\left[\nu_1';\nu_2'\right] \ .$$

**Proof.** By induction on the length of the execution $\left(c,[\mu_1;\mu_2]\right)\to^*\left[\mu_1';\mu_2'\right]$. We consider the different forms of $c$:

1. The case $x:=e$

By rule UPDATE, we have $\left[\mu_1';\mu_2'\right]=\left[\mu_1\left[x:=\mu(e)\right];\mu_2\right]$ and $\left[\nu_1';\nu_2'\right]=\left[\nu_1\left[x:=\nu(e)\right];\nu_2\right]$. If $\gamma(x)=L_{\text{var}}$, then by rule ASSIGN, we have $\gamma|\!-e:L$. So by Simple Security, we have

$$[\mu_1;\mu_2](e)=[\nu_1;\nu_2](e),$$

So

$$[\mu_1;\mu_2]\left[x:=\mu_1(e)\right]\sim_\gamma[\nu_1;\nu_2]\left[x:=\nu_1(e)\right].$$

If $\gamma(x)=H_{\text{var}}$ ,then trivially $[\mu_1;\mu_2]\left[x:=\mu_1(e)\right]\sim_\gamma[\nu_1;\nu_2]\left[x:=\nu_1(e)\right]$.

2. The case *skip* .

The result follows immediately from rule NO-OP.

3. The case $c_1;c_2$.

If $\left(c_1;c_2,[\mu_1;\mu_2]\right)\to^k\left[\mu_1';\mu_2'\right]$ then by Lemma 4.4 there exist $0<j<k$ and $\left[\mu_1'';\mu_2''\right]$such that,

$$\left(c_1,[\mu_1;\mu_2]\right)\to^j\left[\mu_1'';\mu_2''\right],$$

and

$$\left(c_2,\left[\mu_1'';\mu_2''\right]\right)\to^{k-j}\left[\mu_1';\mu_2'\right].$$

Similarly, If $\left(c_1;c_2,[\nu_1;\nu_2]\right)\to^{k'}\left[\nu_1';\nu_2'\right]$, then there exist $0<j'<k'$ and $\left[\nu_1'';\nu_2''\right]$, such that

$$\left(c_1,[\nu_1;\nu_2]\right)\to^{j'}\left[\nu_1'';\nu_2''\right],$$

and

$$\left(c_2,\left[\nu_1'';\nu_2''\right]\right)\to^{k'-j'}\left[\nu_1';\nu_2'\right].$$

By induction, we have $\left[\mu_1'';\mu_2''\right]\sim_\gamma\left[\nu_1'';\nu_2''\right]$. So by induction again, we have $\left[\mu_1';\mu_2'\right]\sim_\gamma\left[\nu_1';\nu_2'\right]$.

4. The case if $e$ then $c_1$ else $c_2$ .

If $\gamma|\!-e:L$ then $\mu_1(e)=\nu_1(e)$ by Simple Security. If $\mu_1(e)\neq 0$ then have the form

$$\left(\text{if } e \text{ then } c_1 \text{ else } c_2,[\mu_1;\mu_2]\right)\to\left(c_1,[\mu_1;\mu_2]\right)\to^*\left[\mu_1';\mu_2'\right]$$

and

$$\left(\text{if } e \text{ then } c_1 \text{ else } c_2,[\nu_1;\nu_2]\right)\to\left(c_1,[\nu_1;\nu_2]\right)\to^*\left[\nu_1';\nu_2'\right]$$

By induction, we have $\left[\mu_1';\mu_2'\right]\sim_\gamma\left[\nu_1';\nu_2'\right]$. The case when $\mu_1(e)=0$ is similar.

If $\gamma|\!\not- e:L$, we have $\gamma|\!-\text{if } e \text{ then } c_1 \text{ else } c_2 : H_{cmd}$. Then by Confinement, we have

$$[\mu_1;\mu_2]\sim_\gamma\left[\mu_1';\mu_2'\right]\text{and } [\nu_1;\nu_2]\sim_\gamma\left[\nu_1';\nu_2'\right].$$

So $\left[\mu_1';\mu_2'\right]\sim_\gamma\left[\nu_1';\nu_2'\right]$.

5. The case while $e$ do $c$ .

Similar to the case $\text{if } e \text{ then } c_1 \text{ else } c_2$.

6. The case $\text{allocate } x[e]$.

We consider the possible types of $x$. If $\gamma(x) = L\ alias\ L$. By Simple Security, we have $[\mu_1;\mu_2](x[e]) = [\nu_1;\nu_2](x[e])$. So $[\mu_1';\mu_2'] \sim_\gamma [\nu_1';\nu_2']$.

If $\gamma(x) = H\ alias\ H$, by rule ALLOCATE we have $\gamma \vdash \text{allocate } x[e] : H_{cmd}$. By the Confinement, we have $[\mu_1';\mu_2'] \sim_\gamma [\nu_1';\nu_2']$.

7. The case $x[e_1] := e_2$.

We consider the possible types of $x$.

If $\gamma(x) = L\ alias\ L$, then by rule ASSIGN-ARR we have $\gamma \vdash x[e_1] : L$, $\gamma \vdash e_2 : L$. By Simple Security, we have $[\mu_1;\mu_2](x[e_1]) = [\nu_1;\nu_2](x[e_1])$, $[\mu_1;\mu_2](e_2) = [\nu_1;\nu_2](e_2)$.

So by rule UPDATE-ARR, we have $[\mu_1';\mu_2'](x) = [\nu_1';\nu_2'](x)$, So, $[\mu_1';\mu_2'] \sim_\gamma [\nu_1';\nu_2']$.

If $\gamma(x) = H\ alias\ H$, then by rule ASSIGN-ARR we have $\gamma \vdash x[e_1] := e_2 : H_{cmd}$. By Confinement, we have $[\mu_1';\mu_2'] \sim_\gamma [\nu_1';\nu_2']$.

8. The case $\&x := x[e]$.

We consider the possible types of $x$.

If $\gamma(x) = H\ alias\ H$, then by rule CALIAS we have $\gamma \vdash \&x := x[e] : H_{cmd}$, By Confinement, we have $[\mu_1';\mu_2'] \sim_\gamma [\nu_1';\nu_2']$.

If $\gamma(x) = L\ alias\ L$, then by rule ALIAS we have $\gamma \vdash \&x : L$, $\gamma \vdash x[e] : L$. By Simple Security, we have $[\mu_1;\mu_2](x[e]) = [\nu_1;\nu_2](x[e])$, $[\mu_1;\mu_2](\&x) = [\nu_1;\nu_2](\&x)$. So by rule CALIAS, we have $[\mu_1';\mu_2'](\&x) = [\nu_1';\nu_2'](\&x)$, So, $[\mu_1';\mu_2'] \sim_\gamma [\nu_1';\nu_2']$.

9. The case Case $x[e_1] := \&x$.

We consider the possible types of $x$.

If $\gamma(x) = H\ alias\ H$, then by rule CALIAS we have $\gamma \vdash x[e_1] := \&x : H_{cmd}$, By Confinement, we have $[\mu_1;\mu_2] \sim_\gamma [\mu_1';\mu_2']$.

If $\gamma(x) = L\ alias\ L$, then by rule ALIAS we have $\gamma \vdash \&x : L$, $\gamma \vdash x[e_1] : L$. By Simple Security, we have $[\mu_1;\mu_2](x[e_1]) = [\nu_1;\nu_2](x[e_1])$, $[\mu_1;\mu_2](\&x) = [\nu_1;\nu_2](\&x)$. So by rule CALIAS, we have $[\mu_1';\mu_2'](x) = [\nu_1';\nu_2'](x)$, So, $[\mu_1';\mu_2'] \sim_\gamma [\nu_1';\nu_2']$.

## 5. Conclusion

This paper allows arrays as first-class value, we can deal with array of arbitrary values. Our array types are of the form $\tau_1$ alias $\tau_2$, This allow us to regard freely discretional array as array of array by alias array. The soundness of our type system is proved by noninterference.

For the example program in Section 1. If array $a_i$ have types $H$ alias $H$ and $L$ alias $L$, the information flow from the $a_i$ to alias $x_i$ are secure.

In the future, we expect new type systems to support for secure information flow in more expressive languages. More generally, we expect to make secure information flow type system practical. The works of Stephen Chong, A.C.Myers [15] and Peng Li, Steve Zdancewic [16] are promising.

## 6. References

[1] A.Sabelfeld and A. C. Myers, Language-based information-flow security. IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security, 21(2003)1, 5-19.

[2] D. E. Denning and P. J. Denning, Secure Information Flow in Computer Systems. Purdue University Ph.D. Thesis, 1975.

[3] D. E. Denning and P. J. Denning, A Lattice Model of Secure Information Flow. Communications of the ACM, 19(1976)5, 236-242.

[4] D. E. Denning and P. J. Denning, Certification of programs for secure information flow. Communications of the ACM, 20(1977)7, 504-513.

[5] D. Volpano, G. Smith, C. Irvine, A Sound Type System for Secure Flow Analysis. Journal of computer security, 4(1996)3, 167-187.

[6] D. Volpano, G. Smith, A type-based approach to program security. in Proc. TAPSOFT'97, vol.1214 of LNCS, Springer-Verlag, Apr. 1997, pp. 607-621.

[7] G. Smith, D. Volpano, Secure information flow in a multi-threaded imperative language. in Proc. ACM Symp. on Principles of Programming Languages, Jan.1998, pp. 355-364.

[8] B. C. Pierce. Types and Programming languages, The MIT Press, Cambridge, Massachusetts, London, England. QA 76.7.P54, 2002

[9] A. Banerjee and D. A. Naumann, Secure information flow and pointer confinement in a Java-like language. in Proc. IEEE Computer Security Foundations Workshop, June 2002, pp. 253-267.

[10] A. Diwan, S. M. Kinley, B.Moss, Type-based alias analysis. In the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1998, pp. 106-107.

[11] Z. Deng and G. Smith, Lenient Array Operations for Practical Secure Information Flow. 17th IEEE Computer Security Foundations Workshop, Pacific Grove, California, June 2004, pp. 115-124.

[12] S. Zdancewic and A. C. Myers, Secure Information Flow via Linear Continuations. Higher Order and Symbolic Computation, 5(2002)2-3, 209-234.

[13] Francois Pottier and Vincent Simonet, Information Flow Inference for ML. in Proc. ACM Symp. on Principles of Programming Languages, Jan.2002, pp. 319-330.

[14] Hanne Riis Nielson and Flemming Nielson, Semantics with Applications a Formal Introduction. July, 1999.

[15] S. Chong and A.C.Myers, Security Policies for Downgrading. Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS), Washington, DC, USA, October 2004, pp. 189-209.

[16] P. Li and S. Zdancewic, Downgrading Policies and Relaxed Noninterference. In Proc. 32nd ACM Symp. On Principles of Programming Languages (POPL), January 2005, pp. 158-170.