An Accurate Type System for Information Flow in Presence of Arrays

Séverine Fratani, Jean-Marc Talbot

An Accurate Type System for Information Flow in Presence of Arrays

Séverine Fratani and Jean-Marc Talbot

Laboratoire d'Informatique Fondamentale de Marseille (LIF)
UMR6166 CNRS - Université de la Méditerranée - Université de Provence

Abstract. Secure information flow analysis aims to check that the execution of a program does not reveal information about secret data manipulated by this program. In this paper, we consider programs dealing with arrays; unlike most of existing works, we will not assume that arrays are homogeneous in terms of security levels. Some part of an array can be declared as secret whereas another part is public. Based on a pre-computed approximation of integer variables (serving as indices for arrays), we devise a type system such that typed programs do not leak unauthorized information. Soundness of our type system is proved by a non-interference theorem.

1 Introduction

Information flow analysis aims to check that data propagation within applications conforms some security requirements; the goal is to avoid that programs leak confidential information during their executions: observable/public outputs of a program must not disclose information about secret/confidential values manipulated by this program.

Data are labelled with security levels, usually H for secret/confidential/high security level values and L for observable/public/low security level variables (or more generally, by elements of a lattice of security levels [6]). The absence of illicit information flow is proved by the non-interference property: if two inputs of the program coincide on their public (observable) part then it is so for the outputs. Information flows arise from assignments (direct flow) or from the control structure of a program (implicit flow). For example, the code `l = h` generates a direct flow from `h` to `l`, while `if(h) then l=1 else l=0` generates an implicit flow. If `h` has security level H and `l` L, then the two examples are insecure and generate an illicit information flow, as confidential data can be deduced by the reader of `l`. Non-interference can be checked through typing or static analysis, leading to the automatic rejection of insecure programs. A considerable part of works on information flow control, based on static analysis, has been achieved in the last decades, *eg* [7,8,12,16,15].

Although some tools based on these works have been developed [12,14], it seems that they are not mature enough to properly address real-world problems. As advocated in [1], we believe that one of the bottlenecks is the current unability of the proposed works on information flow to address real-world programming languages, that is, languages including in particular, built-ins data structures (arrays, hash tables, ..). Another issue is the "real-world" programming style that is frequently used, in particular in the area of embedded software, to save resources (time, space, energy, ..).

In this paper, we focus on arrays, the most casual built-ins data structure. Two works [5,17] have already addressed this topic. Both are based on the Volpano, Smith and Irvine'approach (Volpano, Smith and Irvine proposed in their seminal paper a type system to guarantee the absence of illicit information flows [16]) and share the same feature: the security type of an array is uniform for the whole array; an array is either totally public or totally private. Unfortunately, this may be insufficient for real-world programs.

Let us consider the file given below on the left and the program next to it:

```
Doe
John
john.doe@yahoo.fr
Martin
Jean
jean.martin@mail.com
```

```
i=0
while (not(eof(f)))
  T[i] ::= read(f)
  i :=i+1
j :=0
while (j <i)
  if ((j mod 3) <> 2) then
    print(T[j])
  j :=j+1
```

The structure of the file is simple, one piece of information per line, and one after the other, a last name, a first name and an email. Suppose a policy security stating that first and last names are public but emails have to remain confidential. One may argue that this program is written by a programmer who is unware of security issues and should be rejected because of its programming style. But on the one hand, similar programs are rather frequent in embedded systems and on the other hand, this program does not disclose any confidential information and is thus correct wrt to secure information flows.

This example illustrates that considering access to arrays in fine-grained manner is crucial to obtain relevant results concerning security issues; this fact has motivated the work of Amtoft, Hatcliff and Rodríguez [1]: the authors proposed there an Hoare-style logic to reason about information flows, this logic being able to express properties about single array cells. Our approach is different and composed of two parts: first, an accurate information about the values of integer variables used as indices of arrays is considered. We will assume that these informations on integer variables are known (pre-computed or given by the programmer). Then, based on that, we define a type system such that typed programs do not leak unauthorized information. We prove the soundness of our approach by proving a non-interference theorem.

The paper is organized as follows: in Section 2, we describe the small programming language we consider within this paper: this language is a simple imperative language allowing to manipulate one-dimensional array. However, array aliases are not supported. As discussed earlier, accurate typing requires information about values taken by the integer variables during the execution of the program. These information allow us to consider arrays not only as a single piece of data but also quite sophisticated parts of them. Thus, Section 3 aims to described a framework for expressing approximation of integer variables. We also specify in which sense this framework is sound. As an example, we chose Presburger formulas as they form a quite expressive and well-known theory. Section 4 is devoted to our type system. This type systems relies on the approximation given in the previous section. In that section, we also give some examples of typed programs. Finally, in Section 5, we prove that the devised type system is correct: we show first that it satisfies the subject reduction property. As intermediate steps

to soundness, we prove the "classical" *simple security* property of expressions and the *confinement* property of commands. Finally, we prove the main result of the paper as a non-interference theorem: for inputs data that can not be distinguished on the lower level part, a well-typed program produces outputs coinciding on the lower level part.

2 Language and Semantics

The language we consider is a simple imperative language with arrays. We assume to be fixed a finite set of integer identifier $\mathcal{I} = \{x_1, \ldots, x_n\}$, a finite set of array identifiers $\mathcal{T} = \{T_1, T_2, \ldots\}$ and a finite set of labels $\mathcal{L}$.

The syntax of programs is the given by the following grammar:

$$
\begin{array}{lll}
\text{(EXPRESSIONS)} & e & ::= x_i \mid n \mid T[e] \mid T.\texttt{length} \mid e_1 + e_2 \mid e_1 - e_2 \mid n * e \mid e/n \\
& & \quad e_1 \vee e_2 \mid e_1 \wedge e_2 \mid e_1 \neq e_2 \mid e_1 = e_2 \mid \ldots \\
\text{(COMMANDS)} & c & ::= x_i :=^{\ell} e \mid T[e_1] :=^{\ell} e_2 \mid \texttt{allocate}^{\ell}\, T[e] \mid \texttt{skip}^{\ell} \mid \\
& & \quad \texttt{if}^{\ell}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \mid \texttt{while}^{\ell}\ e\ \texttt{do}\ P \\
\text{(PROGRAMS)} & P & ::= c \mid c; P
\end{array}
$$

Here, meta-variables x_i range over $\mathcal{I}$, n over integers, T over the set of array identifiers $\mathcal{T}$ and ℓ over the set of labels $\mathcal{L}$. The sequence operator ";" is implicitly assumed to be associative. Finally, we assume the labels appearing in a program to be pairwise distinct and we may write c^{ℓ} to emphasize the fact that the label of the command c is ℓ.

It should be noticed that we address only one dimensional arrays and do not support array aliases (expressions such as $T_1 := T_2$ are not allowed).

A program P is executed under a memory μ, which maps identifiers to values in the following way: for all x_i in $\mathcal{I}$, $\mu(x_i) \in \mathbb{Z}$ and for all $T \in \mathcal{T}$, $\mu(T)$ is an array of integers $\langle n_0, n_1, \ldots, n_{k-1} \rangle$, where $k > 0$. Additionally, we assume $\langle\rangle$, a special value for arrays. The length of $\langle\rangle$ is 0.

We assume that expressions are evaluated atomically and we denote by $\mu(e)$ the value of the expression e in the memory μ; the semantics of array, addition and division expressions is given on Fig. 1. The semantics of the others expressions is defined in an obvious way. Note that, as in C language, the result of the evaluation of an expression is always an integer. Moreover, following the lenient semantics proposed in [5], accessing an array with an out-of-bounds index yields 0, as well as division by 0. Hence, the programs we deal with are all error-free. The semantics of programs is given by a transition relation $\rightarrow$ on configurations. A configuration is either a pair (P, μ) or simply a memory μ: in the first case, P is the program yet to be executed, whereas in the second one, the command is terminated yielding the final memory μ. We write $\rightarrow^k$ for the k-fold self composition of $\rightarrow$ and $\rightarrow^*$ for the reflexive, transitive closure of $\rightarrow$.

The semantics of array commands is defined in Fig. 2 (where $\mu[U := V]$ denotes a memory identical to μ except for the variable U whose value is V). Given an expression e, $\texttt{allocate}\ T[e]$ allocates a 0-initialized block of memory for the array T, the size of this array is given by the value of e (when strictly positive). The remaining executions of programs are given by a standard structural operational semantics (SOS) presented on Fig. 3. Remark that when a program is executed, the execution either terminates successfully or loops (it cannot get stuck) : for each configuration of the form (P, μ) there always exists a rule that may apply.

$$\text{(ARR-READ)}\quad \frac{\mu(T) = \langle n_0, n_1, \ldots n_{k-1}\rangle,\ \mu(e) = i,\ 0 \leq i < k}{\mu(T[e]) = n_i}$$

$$\frac{\mu(T) = \langle n_0, n_1, \ldots n_{k-1}\rangle,\ \mu(e) \notin [0, k-1]}{\mu(T[e]) = 0} \qquad \frac{\mu(T) = \langle\rangle}{\mu(T[e]) = 0}$$

$$\text{(GET-LGTH)}\quad \frac{\mu(T) = \langle n_0, n_1, \ldots n_{k-1}\rangle}{\mu(T.\texttt{length}) = k}$$

$$\text{(ADD)}\quad \frac{\mu(e_1) = n_1,\ \mu(e_2) = n_2}{\mu(e_1 + e_2) = n_1 + n_2}$$

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

Fig. 1. Semantics of array, addition and division expressions

3 Over-approximation of the semantics

Our aim is to address security information for arrays in a non-uniform manner; this implies that slices/parts of the arrays have to considered on their own. As arrays are manipulated via indices stored in integer variables, we have to figure out what is the content of these variables during the execution of the program. Capturing the exact values taken by some variables during the execution of a program is, of course, an undecidable problem. Therefore, we require an approximation of the values of those variables. To do so, we associate with each command label ℓ an approximation α of the values of integer variables at this program point (just before the execution of the command labelled with ℓ); α is an approximation in the following sense: in any execution of the program reaching this program point, if the variable x_i has k for value, then α states that k is an admissible value for x_i at that point.

Note that such approximations can be obtained in various ways : tools like FAST [3] or TReX [2] performing reachability analysis of systems augmented with (unbounded) integer variables can be used. Indeed, since we do not need to evaluate values in array cells but just integer values, our programs can be safely abstracted as counter automata. Another possibility is to use tools for inferring symbolic loop invariants such as in [10].

Hence, our aim is not to compute approximations, neither to propose various formalisms to represent them. We will mainly present the set of conditions such approximations must satisfy and prove that this set of conditions is sufficient.

To make our approach more concrete we chose Presburger formulas both to express the required conditions on the approximation as well as the approximation itself.

We recall that a Presburger formula ψ is given by the following syntax:

$$\psi := \exists x\psi \mid \forall x\psi \mid x = x_1 + x_2 \mid x = n \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\psi \mid n * x \mid x/n \mid x \mod n$$

x, x_1, x_2 being integer variables and n an integer.

$$\text{(UPD-ARR)}\quad \frac{\mu(T) = \langle n_0, \ldots n_{k-1}\rangle,\ \mu(e_1) = i \in [0, k-1],\ \mu(e_2) = n}{(T[e_1] :=^{\ell} e_2, \mu) \to \mu[T := \langle n_0, \ldots, n_{i-1}, n, n_{i+1}, \ldots n_{k-1}\rangle]}$$

$$\frac{\mu(T) = \langle n_0, \ldots n_{k-1}\rangle,\ \mu(e_1) \notin [0, k-1]}{(T[e_1] :=^{\ell} e_2, \mu) \to \mu} \qquad \frac{\mu(T) = \langle\rangle}{(T[e_1] :=^{\ell} e_2, \mu) \to \mu}$$

$$\text{(CALLOC)}\quad \frac{\mu(e) > 0,\ \mu(T) = \langle\rangle}{(\texttt{allocate}^{\ell}\ \ T[e], \mu) \to \mu[T := \langle \underbrace{0, 0, \ldots, 0}_{\mu(e)}\rangle]}$$

$$\frac{\mu(e) \leq 0}{(\texttt{allocate}^{\ell}\ \ T[e], \mu) \to \mu} \qquad \frac{\mu(T) \neq \langle\rangle}{(\texttt{allocate}^{\ell}\ \ T[e], \mu) \to \mu}$$

Fig. 2. Semantics of array commands

$$\text{(UPDATE)}\quad (x :=^{\ell} e, \mu) \to \mu[x := \mu(e)] \qquad \text{(NO-OP)}\quad (\texttt{skip}^{\ell}, \mu) \to \mu$$

$$\text{(BRANCH)}\quad \frac{\mu(e) \neq 0}{(\texttt{if}^{\ell}\ \ e\ \texttt{then}\ \ P_1\ \texttt{else}\ \ P_2, \mu) \to (P_1, \mu)}$$

$$\frac{\mu(e) = 0}{(\texttt{if}^{\ell}\ \ e\ \texttt{then}\ \ P_1\ \texttt{else}\ \ P_2, \mu) \to (P_2, \mu)}$$

$$\text{(LOOP)}\quad \frac{\mu(e) \neq 0}{(\texttt{while}^{\ell}\ \ e\ \texttt{do}\ \ P, \mu) \to (P; \texttt{while}^{\ell}\ \ e\ \texttt{do}\ \ P, \mu)} \qquad \frac{\mu(e) = 0}{(\texttt{while}^{\ell}\ \ e\ \texttt{do}\ \ P, \mu) \to \mu}$$

$$\text{(SEQUENCE)}\quad \frac{(c_1, \mu) \to \mu'}{(c_1; P_2, \mu) \to (P_2, \mu')} \qquad \frac{(c_1, \mu) \to (P, \mu')}{(c_1; P_2, \mu) \to (P; P_2, \mu')}$$

Fig. 3. Part of the SOS for programs

We will use the following conventions. Given two Presburger formulas $\psi(\bar{x})$ and $\psi'(\bar{x})$, we write $\psi \subseteq \psi'$ if $\models \forall \bar{x}, \neg\psi(\bar{x}) \vee \psi'(\bar{x})$. Given a Presburger formula $\psi(y, \bar{x})$ and an integer n, we write $\psi(n, \bar{x})$ instead of $\psi(y, \bar{x}) \wedge y = n$.

For the approximation, we consider here programs labelled by Presburger formulas whose free variables are essentially $x_1, \ldots, x_n$, which corresponds to the integer variable identifiers [1]. A labelling is then a mapping λ associating with each label $\ell \in \mathcal{L}$, a Presburger formula $\psi(\bar{x})$.

Given a program P and a labelling λ, we define $\lambda(P)$ to be the label of the first command of P, that is, recursively, $\lambda(c^{\ell}) = \lambda(\ell)$ and $\lambda(P_1; P_2) = \lambda(P_1)$. Intuitively, $\lambda(P) = \psi(\bar{x})$ means "enter in P with a memory μ whose integer variables satisfy $\psi(\bar{x})$", i.e., $[x_1 \mapsto \mu(x_1), \ldots, x_n \mapsto \mu(x_n)] \models \psi(\bar{x})$.

[1] In the rest of the paper, we will denote by $\bar{x}$ the sequence of variables $x_1, \ldots, x_n$

We start by defining in Fig. 4 for every expression e, the formula $V_e^\psi(y, \bar{x})$ giving the possible values of e when the integer variables of the memory satisfy a formula ψ. $V_e^\psi(y, \bar{x})$ is then a formula fulfilling for all memories μ: $[y \mapsto n, \bar{x} \mapsto \mu(\bar{x})] \models V_e^\psi(y, \bar{x})$ iff $[\bar{x} \mapsto \mu(\bar{x})] \models \psi(\bar{x})$ and $\mu(e) = n$. Hence, for instance, for $x_1 + x_2$ and

(O-INT)	$V_n^\psi(y, \bar{x}) \stackrel{def}{=} (y = n) \wedge \psi(\bar{x})$
(O-INT-VAR)	$V_{x_i}^\psi(y, \bar{x}) \stackrel{def}{=} (y = x_i) \wedge \psi(\bar{x})$
(O-ARR-READ)	$V_{T[e]}^\psi(y, \bar{x}) \stackrel{def}{=} \psi(\bar{x})$
(O-GET-LGTH)	$V_{T.\texttt{length}}^\psi(y, \bar{x}) \stackrel{def}{=} \psi(\bar{x})$
(O-ADD)	$V_{e_1+e_2}^\psi(y, \bar{x}) \stackrel{def}{=} \exists y_1, \exists y_2, V_{e_1}^\psi(y_1, \bar{x}) \wedge V_{e_2}^\psi(y_2, \bar{x}) \wedge y = y_1 + y_2$
(O-DIV)	$V_{e/0}^\psi(y, \bar{x}) \stackrel{def}{=} V_0^\psi(y, \bar{x})$
if $n \neq 0$	$V_{e/n}^\psi(y, \bar{x}) \stackrel{def}{=} [V_e^\psi(0, \bar{x}) \wedge y = 0] \vee [\exists y_1, V_e^\psi(y_1, \bar{x}) \wedge y = y_1/n]$

Fig. 4. Definition of $V_e^\psi(y, \bar{x})$ for relevant expressions e

$\psi = x_1 \leq 3 \wedge x_2 = 4$, we have that $V_{x_1+x_2}^\psi(y, x_1, x_2)$ is the formula $x_1 \leq 3 \wedge x_2 = 4 \wedge y = x_1 + x_2$. Note also that for an expression $T[e]$, the resulting formula $V_{T[e]}^\psi(y, \bar{x})$ imposes no constraints on y and thus, gives no information on the possible value of such an expression (there is no analysis of array contents). Similarly, let us point out, because of the programming language we have chosen, that the formulas $V_e^\psi(\bar{x})$ are always Presburger formulas. It may not be the case if expressions such as $e_1 * e_2$ were allowed: however, as we simply require an approximative computation, we could have defined $V_{e_1*e_2}^\psi(y, \bar{x})$ as $\psi(\bar{x})$, leading to coarse but correct approximation.

Definition 1. *Let μ be a memory and $\psi(\bar{x})$ be a Presburger formula. We say that $\psi(\bar{x})$ is an over-approximation of μ (denoted $\psi(\bar{x}) \geq \mu$) if $[x_1 \mapsto \mu(x_1), \ldots, x_n \mapsto \mu(x_n)] \models \psi(x_1, \ldots, x_n)$.*

Lemma 1. *Let μ be a memory and $\psi(\bar{x})$ be a Presburger formula. If $\psi(\bar{x}) \geq \mu$ then $V_e^\psi(\mu(e), \bar{x}) \geq \mu$ (that is, $\models V_e^\psi(\mu(e), \mu(x_1), \ldots, \mu(x_n))$).*

Given a program P, a labelling λ, and a Presburger formula $\psi(\bar{x})$, we write $P \vdash_\lambda \psi$ to say that if we execute P under a memory μ satisfying $\lambda(P)$, then we get a new memory satisfying ψ. Formally,

Definition 2. *A program P is well labelled by λ if there exists a Presburger formula $\psi(\bar{x})$ such that $P \vdash_\lambda \psi$, where $\vdash_\lambda$ is the relation defined inductively in Fig. 5 according to the possible shape of P.*

(O-UP-ARR) $\dfrac{\lambda(\ell) \subseteq \psi}{x[e_1] :=^{\ell} e_2 \vdash_\lambda \psi}$ (O-CALLOC) $\dfrac{\lambda(\ell) \subseteq \psi}{\texttt{allocate}^{\ell}\ x[e] \vdash_\lambda \psi}$

(O-UPDATE) $\dfrac{(\exists y, [\exists x_i, V_e^{\lambda(\ell)}(y, x_1, \ldots x_{i-1}, x_i, x_{i+1}, \ldots, x_n,)] \wedge x_i = y) \subseteq \psi}{x_i :=^{\ell} e \vdash_\lambda \psi}$

(O-NO-OP) $\dfrac{\lambda(\ell) \subseteq \psi}{\texttt{skip}^{\ell} \vdash_\lambda \psi}$

(O-BRANCH) $V_e^{\lambda(\ell)}(0, \bar{x}) \subseteq \lambda(P_2), \exists y \neq 0. V_e^{\lambda(\ell)}(y, \bar{x}) \subseteq \lambda(P_1)$
$\dfrac{P_2 \vdash_\lambda \psi_2,\ P_1 \vdash_\lambda \psi_1,\ \psi_2 \vee \psi_1 \subseteq \psi}{\texttt{if}^{\ell}\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 \vdash_\lambda \psi}$

(O-LOOP) $\dfrac{\exists y \neq 0. V_e^{\lambda(\ell)}(y, \bar{x}) \subseteq \lambda(P),\ P \vdash_\lambda \lambda(\ell),\ V_e^{\lambda(\ell)}(0, \bar{x}) \subseteq \psi}{\texttt{while}^{\ell}\ e\ \texttt{do}\ P \vdash_\lambda \psi}$

(O-SEQ) $\dfrac{P_1 \vdash_\lambda \lambda(P_2),\ P_2 \vdash_\lambda \psi}{P_1; P_2 \vdash_\lambda \psi}$

Fig. 5. Definition of $P \vdash_\lambda \psi$

Let us point out that from rule (O-LOOP), the label $\lambda(P)$ in a command $\texttt{while}^{\ell}\ e\ \texttt{do}\ P$ has to be an invariant for this loop.

We give an example for the rule (O-UPDATE): if $x_1 :=^{\ell} x_1 + x_2$ and $\lambda(\ell) = x_1 \leq 3 \wedge x_2 = 4$, we have that $V^{\psi}_{x_1+x_2}(y, x_1, x_2)$ is $x_1 \leq 3 \wedge x_2 = 4 \wedge y = x_1 + x_2$ and then $\exists y[\exists x_1, V^{\psi}_{x_1+x_2}(y, x_1, x_2)] \wedge x_1 = y$ is equivalent to $\exists y, \exists x'_1, x'_1 \leq 3 \wedge x_2 = 4 \wedge y = x'_1 + x_2 \wedge x_1 = y$ i.e. to $x_2 = 4 \wedge x_1 \leq 7$. Then $x_i :=^{\ell} e \vdash_\lambda \psi$ for all ψ such that $(x_2 = 4 \wedge x_1 \leq 7) \subseteq \psi$.

Remark that for each line of a program, checking the label costs the test of the validity of a Presburger formula with at most $2 + k$ quantifier alternations, where k is the number of quantifier alternations of the label formula.

Lemma 2. *If $P \vdash_\lambda \psi$ and $(P, \mu) \rightarrow (P', \mu')$ then there exists ψ' s.t. $P' \vdash_\lambda \psi'$ and $\psi' \subseteq \psi$.*

Proposition 1. *If P is well-labelled by λ, $(P, \mu) \rightarrow^* (P', \mu')$ and $\lambda(P) \geq \mu$, then $\lambda(P') \geq \mu'$.*

4 A Type System for Information Flow in Arrays

In this section, we present the type system we devise to guarantee the absence of illicit information flows in programs manipulating arrays. We exemplify our approach giving the type of some programs.

The goal of this type system is to guarantee that the execution of the program does not disclose confidential information that is, the contents of public variables do not depend on the contents of private ones through explicit and implicit flows.

4.1 The type system

Our type system relies on a labelling of the program with approximation of the values taken by integer variables. This labelling is just supposed to be a well-labelling in the sens of Definition 2.

Types for arrays are couples (ϕ_H, τ), where ϕ_H is a Presburger formula, meaning that for all $n \geq 0$, $\phi_H(n)$ holds iff the cell of index n in the array has type H (otherwise it has type L) and that the length of the array has type τ.

Here are the types used by our type system:

$$\begin{array}{ll} \text{(data types)} & \tau ::= L \mid H \\ \text{(phrase types)} & \rho ::= \tau \mid \tau\, var \mid \tau\, cmd \mid (\phi_H, \tau) \end{array}$$

Intuitively, τ is the security level for integer expressions, $\tau\, var$ is the one of integer variables and (ϕ_H, τ) is the security level for the cells and the length of arrays. For commands and programs, $\tau\, cmd$ stands for a program that can safely write in storage (variables, array cells) of level higher than τ.

Moreover, we adopt the following constraint of types : in any array type (ϕ_H, τ), we require that if $\tau = H$ then ϕ_H is the formula $True$ **(Array type constraint)**. Indeed, due to the lenient semantics of arrays, reading the content of `T[3]` after the execution of `allocate T[h]; T[3] := 2;` leaks information about `h` (if `T[3]` is zero then `h` is smaller than 3).

Our type system allows to prove judgments of the form $\Gamma, \psi \vDash e : \tau$ for expressions and of the form $\Gamma \vDash_\lambda P : \tau\, cmd$ for programs as well of subtyping judgments of the form $\rho_1 \subseteq \rho_2$. Here Γ denotes an environment, that is an identifier typing, mapping each integer identifier in $\mathcal{I}$ to a phrase type of the form $\tau\, var$ and each array identifier in $\mathcal{T}$ to a phrase type of the form (ϕ_H, τ), ψ is a Presburger formula, e is an expression, P is a program and λ is a labelling.

The subtyping rules are on Figure 6: informally, a low level expression can always be used in the high level situation (hence, constants can be assigned to high level variables) whereas high level commands can always be used in a low level context (if only private variables are affected then only variables greater than L have been affected).

$$\begin{array}{llll} \text{(BASE)} & L \subseteq H & \text{(CMD}^-\text{)} & \dfrac{\tau' \subseteq \tau}{\tau\, cmd \subseteq \tau'\, cmd} \\[2ex] \text{(REFLEX)} & \rho \subseteq \rho & \text{(TRANS)} & \dfrac{\rho_1 \subseteq \rho_2,\ \rho_2 \subseteq \rho_3}{\rho_1 \subseteq \rho_3} \\[2ex] \text{(SUBSUMP)} & \dfrac{\Gamma, \psi \vDash e : \tau_1,\ \tau_1 \subseteq \tau_2}{\Gamma, \psi \vDash e : \tau_2} & & \dfrac{\Gamma \vDash_\lambda P : \rho_1,\ \rho_1 \subseteq \rho_2}{\Gamma \vDash_\lambda P : \rho_2} \end{array}$$

Fig. 6. Subtyping rules

The most relevant typing rules for expressions can be found in Fig. 7. Typing rules (INT), (R-VAL) and (QUOTIENT) are borrowed from the Volpano-Smith type system:

$$\text{(INT)}\quad \Gamma,\psi \vDash n : L \qquad\qquad \text{(R-VAL)}\quad \frac{\Gamma(x_i) = \tau\ var}{\Gamma,\psi \vDash x_i : \tau}$$

$$\text{(SUBSCR)}\quad \frac{\Gamma,\psi \vDash e : \tau}{\Gamma,\psi \vDash T[e] : H}$$

$$\frac{\Gamma(T) = (\phi_H, L),\ \Gamma,\psi \vDash e : L,\ \models \forall y, [(\exists \bar{x}, V_e^{\psi}(y,\bar{x})) \implies \neg\phi_H(y)]}{\Gamma,\psi \vDash T[e] : L}$$

$$\text{(LENGTH)}\quad \frac{\Gamma(T) = (\phi_H, \tau)}{\Gamma,\psi \vDash T.\texttt{length} : \tau} \qquad \text{(QUOTIENT)}\quad \frac{(\Gamma,\psi) \vDash e_1 : \tau,\ \Gamma,\psi \vDash n : \tau}{\Gamma,\psi \vDash e_1/n : \tau}$$

Fig. 7. Typing rules for expressions

the rule (INT) states that constants are from level L (and also, from level H thanks to subtyping). For variables, as usual, the environment Γ fixes the type of identifier. Finally, for arithmetical operations such as (QUOTIENT), the type of the parameters and of the result have to be the same. The rule (LENGTH) is simply the direct translation of the meaning of types for arrays in the spirit of (R-VAL).

The first rule for (SUBSCR) states that any cell of arrays can be typed as H. The second rule for (SUBSCR) states that the content of an array cell is L only if the expression used to address this cell is L and the possible values for this expression according to the approximation given by λ are set to be not H by the formula ϕ_H of the array type. Note that due to the "array type constraint", the fact that some cells being not of high level H implies that the length of the array is not high (and thus, low).

Typing rules for programs are described in Fig. 8. The types deduced for the commands control mainly the security level of the context in which they are executed; it aims to treat implicit control flows. The typing rules (ASSIGN), (SKIP),(IF), (WHILE), (COMPOSE) are borrowed from the Volpano-Smith type system: the last three simply force the components of a composed program to have the same type. The rule (ASSIGN) states that the assigned value and the variable must be of the same type. This prevents illicit direct flow of information. Finally, because of the rule (SKIP), the command `skip` is used in a high security context (but it can also be used in a low security context thanks to subtyping).

Let us point out that for array cells, just as scalar variables, we distinguish between the security level associated with a cell from the security level of the content of this cell.

The typing rule for (ALLOCATE) states that the type of an expression used in the declaration of an array must coincide with the type of the size of this array. Finally, for (ASSIGN-ARR), the rules aim to guarantee that, as for scalar variables, high level values can not be assigned to low level array cells (to address explicit flows); moreover, a command modifying a low level cell has to be a low level command (to address implicit flows).

In our typing approach, information on the security level for variables is fixed for the whole program; hence, the type of arrays are not modified during the execution according to updates.

(ASSIGN) $\dfrac{\Gamma(x) = \tau\, var,\ \Gamma, \lambda(\ell) \vDash e : \tau}{\Gamma \vDash_\lambda (x :=^\ell e) : \tau\, cmd}$ (SKIP) $\Gamma \vDash_\lambda (\mathtt{skip}^\ell) : H\, cmd$

(ASSIGN-ARR) $\dfrac{\Gamma, \lambda(\ell) \vDash e_1 : L,\ \Gamma, \lambda(\ell) \vDash e_2 : \tau,\ \Gamma(T) = (\phi_H, L), \models \forall y,\ [(\exists \bar{x},\ V_{e_1}^{\lambda(\ell)}(y, \bar{x}) \implies \phi_H(y)]}{\Gamma \vDash_\lambda (T[e_1] :=^\ell e_2) : H\, cmd}$

$\dfrac{\Gamma, \lambda(\ell) \vDash e_1 : H,\ \Gamma, \lambda(\ell) \vDash e_2 : \tau,\ \Gamma(T) = (\phi_H, H)}{\Gamma \vDash_\lambda (T[e_1] :=^\ell e_2) : H\, cmd}$

$\dfrac{\Gamma, \lambda(\ell) \vDash e_1 : L,\ \Gamma, \lambda(\ell) \vDash e_2 : L,\ \Gamma(T) = (\phi_H, L)}{\Gamma \vDash_\lambda (T[e_1] :=^\ell e_2) : L\, cmd}$

(ALLOCATE) $\dfrac{\Gamma(T) = (\phi_H, \tau),\ \Gamma, \lambda(\ell) \vDash e : \tau}{\Gamma \vDash_\lambda (\mathtt{allocate}^\ell\ T[e]) : \tau\, cmd}$

(IF) $\dfrac{\Gamma, \lambda(\ell) \vDash e : \tau,\ \Gamma \vDash_\lambda P_1 : \tau\, cmd,\ \Gamma \vDash_\lambda P_2 : \tau\, cmd}{\Gamma \vDash_\lambda (\mathtt{if}^\ell\ e\ \mathtt{then}\ P_1\ \mathtt{else}\ P_2) : \tau\, cmd}$

(WHILE) $\dfrac{\Gamma, \lambda(\ell) \vDash e : \tau,\ \Gamma \vDash_\lambda P : \tau\, cmd}{\Gamma \vDash_\lambda (\mathtt{while}^\ell\ e\ \mathtt{do}\ P) : \tau\, cmd}$

(COMPOSE) $\dfrac{\Gamma \vDash_\lambda c^\ell : \tau\, cmd,\ \Gamma \vDash_\lambda P : \tau\, cmd}{\Gamma \vDash_\lambda (c^\ell; P) : \tau\, cmd}$

Fig. 8. Typing rules for programs

Remark that for checking a given typing, at each line of the program, we have to test the validity of a Presburger formula with at most 2+$max(k + l)$ quantifier alternations, where k is the number of quantifier alternations of the label formula, and l is the number of quantifier alternations of the formula typing the array occurring in the program line.

In fact, if we infer a typing for the program, we get array formulas containing at most 1+m quantifier alternations, where m is the maximum of the number of quantifier alternations of formulas labelling the program.

4.2 Example

To illustrate our approach, let us give now examples obtained with a prototype that we have developed which, from a program P, a labelling λ and some initial typing, determines if P is well-labelled and when possible types the program P respecting the initial typing by giving the minimal type. The first example shows how an array allocation can force the type of an array to be H:

```
[true]                                  x1:=T1.length;
[true]                                  x2:=T2.length;
[true]                                  x3:=x1+x2;
[x3=x1+x2]                              allocate(T0[x3]);
```

```
[x3=x1+x2]                                  x4:=0;
[x3=x1+x2 and x4=0]                         x5:=0;
[x3=x1+x2 and 0<=x4<=x3+1 and x4=2x0]       while (x4 < x3) do
[x3=x1+x2 and 0<=x4<x3 and x4=2x0]             T0[x4]:=T1[x0];
[x3=x1+x2 and 0<=x4<x3 and x4=2x0]             T0[x4+1]:=T2[x0];
[x3=x1+x2 and 0<=x4<x3 and x4=2x0]             x4:=x4+2;
[x3=x1+x2 and 2<=x4<=x3+1 and x4=2x0+2]        x0:=x0+1;
```

If we impose the constaint `T2:(True,H)`, we get in a time less than one second:

```
T0: (True,High),T1: (False, Low),T2: (True, High)
x0: High, x1: Low, x2: High, x3:High, x4:High.
```

Indeed, line 2 and rule (ALLOCATE) imply `x2:High`, then line 3 and rule (ASSIGN) imply `x3:High` and then line 4 and rule (ALLOCATE) imply `T0:(True,High)`. If `T0` is allocated with a size typed by `Low`, then all the even indexes of `T0` can have the type `Low`, as shown in the following example:

```
[ True ]                                      allocate(T0[x1]);
[ True ]                                      x2 := 0;
[ x2=0 ]                                      x0 := 0;
[ 0 <= x2 <= 1+x1 and x2=2x0]              while ( x2 < x1 ) do
[ 0 <= x2 < x1 and x2=2x0]                      T0[x2] := T1[x0];
[ 0 <= x2 < x1 and x2=2x0]                      T0[x2 + 1] := T2[x0];
[ 0 <= x2 < x1 and x2=2x0]                      x2 := x2 + 2;
[ 2 <= x2<=1+x1 and x2=2+2x0]                   x0 := x0 + 1;
```

If we impose the constaint `T1:(True,H)`, our prototype returns:

```
T0: (Ex x2,x0.(y=x2 and x2=2x0),Low),T1:(True,High),T2:(False,Low)
x0: Low, x1: Low, x2: Low.
```

Remark that since all labels are quantifier free, the formula obtained for the type of `T0` is existential.

5 Properties of the Type system

We prove that our type system guarantees noninterference for well labelled programs. The proofs of some 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 the rule (SUBSUMP).

Lemma 3 (Subject Reduction). *If* $\Gamma \vDash_\lambda P : \tau\ cmd$ *and* $(P,\mu) \rightarrow (P',\mu')$ *then* $\Gamma \vDash_\lambda P' : \tau\ cmd$.

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

1. Case $\texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. By our assumption, the typing derivation for P must end with a use of the rule (IF) followed by a use of (SUBSUMP):

$$\frac{\dfrac{(\Gamma,\lambda(\ell)) \vDash e : \tau',\ \Gamma \vDash_\lambda P_1 : \tau'\ cmd,\ \Gamma \vDash_\lambda P_2 : \tau'\ cmd}{\Gamma \vDash_\lambda \texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 : \tau'\ cmd,\ \tau'\ cmd \subseteq \tau cmd} }{\Gamma \vDash_\lambda \texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2 : \tau\ cmd}$$

Hence, by the rule (CMD$^-$), we must have $\tau \subseteq \tau'$. So by (SUBSUMP) we have $\Gamma \vDash_\lambda P_1 : \tau\, cmd$ and $\Gamma \vDash_\lambda P_2 : \tau\, cmd$. By the rule (BRANCH), P' can be either P_1 or P_2, therefore, we have $\Gamma \vDash_\lambda P' : \tau\, cmd$.

2. Case $\texttt{while}^\ell\ e\ \texttt{do}\ P_1$. By an argument similar to the one used in the previous case, we have $\Gamma \vDash_\lambda P_1 : \tau\, cmd$. By the rule (LOOP), P' is P_1, hence $\Gamma \vDash_\lambda P' : \tau\, cmd$ by rule (COMPOSE).
3. Case $P = {c_1}^\ell; P_2$. As above, we get $\Gamma \vDash_\lambda {c_1}^\ell : \tau\, cmd$ and $\Gamma \vDash_\lambda P_2 : \tau\, cmd$. By the rule (SEQUENCE) P' is either P_2 (if c_1 terminates in one step) or else $P_1; P$, where $({c_1}^\ell, \mu) \rightarrow (P_1, \mu')$. For the first case, we have then $\Gamma \vDash_\lambda P_2 : \tau\, cmd$. For the second case, we have $\Gamma \vDash_\lambda P_1 : \tau\, cmd$ by induction; hence $\Gamma \vDash_\lambda P_1; P_2 : \tau\, cmd$ by rule the (COMPOSE).

We also need an obvious lemma about the execution of a sequential composition.

Lemma 4. *If* $((c^\ell; P), \mu) \rightarrow^j \mu'$ *then there exist* k *and* μ'' *such that* $0 < k < j$, $(c^\ell, \mu) \rightarrow^k \mu$", $((c^\ell; P), \mu) \rightarrow^k (P, \mu")$ *and* $(P, \mu") \rightarrow^{j-k} \mu'$.

Definition 3. *Memories* μ *and* ν *are equivalent with respect to* Γ*, written* $\mu \sim_\Gamma \nu$*, if*

- μ *and* ν *agree on all* L *variables,*
- *for all* $T \in \mathcal{T}$*, if* $\Gamma(T) = (\phi_H, L)$*, then there exists* $k \geq 0$ *such that* $\mu(T) = \langle n_0, \ldots, n_{k-1} \rangle$ *and* $\nu(T) = \langle n'_0, \ldots, n'_{k-1} \rangle$ *and for all* $i \in [0, k-1]$, $\models \neg\phi_H(i)$ *implies* $n_i = n'_i$.

Lemma 5 (Simple Security). *If* $\Gamma, \psi \vDash e : L$ *and* $\mu \sim_\Gamma \nu$ *and* $\psi \geq \mu$*, then* $\mu(e)$=$\nu(e)$.

Proof. By induction on the structure of e:

1. Case n. Obviously, $\mu(e) = \nu(e) = n$.
2. Case x_i. By the typing rule (r-val), $\Gamma(x_i) = L\, var$. Therefore, by memory equivalence, $\mu(x_i) = \nu(x_i)$.
3. Case $T[e]$. By the typing rule (subscr), we have $\Gamma(T) = (\phi_H, L)$, $\Gamma, \psi \vDash e : L$, and $\models \forall y[(\exists \bar{x}. V_e^\psi(y, \bar{x}) \implies \neg\phi_H(y)]$. Since $\psi \geq \mu$, from Lemma 1, $\models V_e^\psi(\mu(e), \mu(\bar{x}))$, then $\models \neg(\phi_H(\mu(e)))$. From $\Gamma, \psi \vDash e : L$ and induction hypothesis, there exists an integer i such that $\mu(e) = \nu(e) = i$. Suppose that $\mu(T) = \langle n_0, \ldots, n_{k-1} \rangle$ and $\nu(x) = \langle n'_0, \ldots, n'_{k-1} \rangle$ ($\mu(T)$ and $\nu(T)$ have the same length by memory equivalence). We consider then two cases:
   - case $i \in [0, k-1]$. Since $\models \neg\phi_H(i)$, by memory equivalence we get $n_i = n'_i$. Then by the SOS rule (ARR-READ), $\mu(T[e]) = \nu(T[e])$.
   - case $i \notin [0, k-1]$. By the SOS rule (ARR-READ), $\mu(T[e]) = \nu(T[e]) = 0$.
4. Case $T.\texttt{length}$. Suppose that $\Gamma, \psi \vDash T.\texttt{length} : L$, then $\Gamma(T) = (\phi_H, L)$ and from memory equivalence, there exists an integer $k \geq 0$ such that $\mu(T) = \langle n_0, \ldots, n_{k-1} \rangle$ and $\nu(x) = \langle n'_0, \ldots, n'_{k-1} \rangle$. From the SOS semantics(GET-LGTH), $\mu(T.\texttt{length}) = \nu(T.\texttt{length}) = k$.
5. Case e/n. By the typing rule (QUOTIENT), we have $\Gamma, \psi \vDash e_1 : L$ and $\Gamma, \psi \vDash n : L$. By induction, we have $\mu(e_1) = \nu(e_1)$.Then by the SOS rule (DIV), we have $\mu(e_1/n) = \nu(e_1/n)$.

Lemma 6 (Confinement). *If $\Gamma \vDash_\lambda P : H\ cmd$, $(P,\mu)\rightarrow(P',\mu')$ (or $(P,\mu)\rightarrow\mu'$) and $\lambda(P) \geq \mu$ then $\mu \sim_\Gamma \mu'$.*

Proof. By induction on the structure of c:

1. Case $x_i :=^\ell e$. From the SOS rule (UPDATE), $\mu' = \mu[x := \mu(e)]$, and from the typing rule (ASSIGN) $\Gamma(x) = H\ var$, so $\mu \sim_\Gamma \mu'$.
2. Case $T[e_1] :=^\ell e_2$. We consider two cases according to the type of e_1.
   - Case $\Gamma, \lambda(\ell) \vDash e_1 : L$, then from the typing rule (ASSIGN-ARR), $\Gamma(T) = (\phi_H, L)$ and $\models \forall y[(\exists \bar{x}, V_e^{\lambda(\ell)}(y,\bar{x}) \implies \phi_H(y)]$. Since $\lambda(\ell) \geq \mu$, we have from Lemma 1: $\models V_{e_1}^{\lambda(\ell)}(\mu(e_1), \mu(\bar{x}))$, and then $\models \phi_H(\mu(e_1))$. Let us suppose that $\mu(x) = \langle n_0, \ldots, n_{k-1}\rangle$, we consider two cases:
     - if $\mu(e_1) \in [0, k-1]$, by the SOS rule (UPD-ARRAY) $\mu' = \mu[T[\mu(e_1)] := \mu(e_2)]$ and since $\models \phi_H(i)$, $\mu \sim_\Gamma \mu'$.
     - else, by the SOS rule (UPD-ARRAY) $\mu' = \mu$ and then $\mu \sim_\Gamma \mu'$.
   - Case $\Gamma, \lambda(\ell) \vDash e_1 : H$, then $\Gamma(T) = (\phi_H, L)$ and $\mu \sim_\Gamma \mu'$.
3. Case $\texttt{allocate}^\ell\ T[e]$. From the typing rule (ALLOCATE), $\Gamma(T) = (\phi_H, H)$ and from the SOS rule (CALLOC), $\mu \sim_\Gamma \mu'$,
4. Cases $\texttt{skip}^\ell$, $\texttt{if}^\ell\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$ and $\texttt{while}^\ell\ e\ \texttt{do}\ c$. These cases are trivial, because $\mu = \mu'$.
5. Case ${c_1}^\ell; P_2$. From the typing rule (COMPOSE), $\Gamma \vDash_\lambda {c_1}^\ell : H\ cmd$ and from the SOS rule (SEQUENCE), $({c_1}^\ell, \mu)\rightarrow\mu'$ or there exists P_1 such that $({c_1}^\ell, \mu)\rightarrow(P_1, \mu')$. Then $\lambda(c_1) = \lambda(P) \geq \mu$ and by induction, $\mu \sim_\Gamma \mu'$.

Corollary 1. *If P is well-labelled by λ, $\Gamma \vDash_\lambda P : H\ cmd$, $\mu : \Gamma$, $(P,\mu)\rightarrow^*\mu'$ and $\lambda(P) \geq \mu$ then $\mu \sim_\Gamma \mu'$.*

Proof. By induction on the length of the derivation $(P,\mu)\rightarrow^*\mu'$. The base case is proved by Confinement (Lemma 6). Let us prove the induction step. We suppose that $(P,\mu)\rightarrow(P_1,\mu_1)\rightarrow^*\mu'$. From Confinement, $\mu \sim_\Gamma \mu_1$, from Subject Reduction, $\Gamma \vDash_\lambda P_1 : H\ cmd$ and as P is well-labelled by λ, by Proposition 1, $\lambda(P_1) \geq \mu_1$. From Lemma 2, P_1 is well-labelled by λ, hence, by induction hypothesis, $\mu_1 \sim_\Gamma \mu'$, and then $\mu \sim_\Gamma \mu'$.

Theorem 1 (Noninterference). *Suppose that P is well-labelled by λ, $\mu \sim_\Gamma \nu$, $\Gamma \vDash_\lambda P : \tau\ cmd$, $\lambda(P) \geq \mu$ and $\lambda(P) \geq \nu$. If $(P,\mu)\rightarrow^*\mu'$ and $(P,\nu)\rightarrow^*\nu'$ then $\mu' \sim_\Gamma \nu'$.*

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

1. Case $x :=^\ell e$. By the SOS rule (UPDATE), we have $\mu' = \mu[x := \mu(e)]$ and $\nu' = \nu[x := \nu(e)]$. If $\Gamma(x) = L\ var$, then by the typing rule (ASSIGN), we have $(\Gamma, \ell) \vDash_\lambda e : L$. So by Simple Security, $\mu(e) = \nu(e)$, and so $\mu' \sim_\Gamma \nu'$. If instead, $\Gamma(x) = H\ var$ then trivially $\mu' \sim_\Gamma \nu'$.
2. Case $T[e_1] :=^\ell e_2$. We consider the possible values of τ.
   - If $\tau = H$, then from Corollary 1, $\mu \sim_\Gamma \mu'$ and $\nu \sim_\Gamma \nu'$. So $\mu' \sim_\Gamma \nu'$.
   - If $\tau = L$, then $\Gamma, \lambda(\ell) \vDash_\lambda e_1 : L$ and $\Gamma, \lambda(\ell) \vDash_\lambda e_2 : L$ and $\Gamma(T) = (\phi_H, L)$. By Simple Security, there exist i, j s.t. $\mu(e_1)$=$\nu(e_1)$=i and $\mu(e_2)$=$\nu(e_2)$=j. Since $\mu \sim_\Gamma \nu$, there exists $k \geq 0$ s.t $\mu(T)$=$\langle n_0, \ldots n_{k-1}\rangle$, $\nu(T)$=$\langle n'_0, \ldots n'_{k-1}\rangle$.

- if $i \in [0, k-1]$. By the SOS rule (UPDATE-ARR), $\mu' = \mu[T[i] := j]$ and $\nu' = \nu[T[i] := j]$. So $\mu' \sim_\Gamma \nu'$.
- else, from the SOS rule (UPDATE-ARR), $\mu' = \mu$ and $\nu' = \nu$. So $\mu' \sim_\Gamma \nu'$.

3. Case $\texttt{allocate}^\ell\ T[e]$. We consider two cases.
   - if $\tau = H$, then by Corollary 1, $\mu \sim_\Gamma \mu'$ and $\nu \sim_\Gamma \nu'$. So $\mu' \sim_\Gamma \nu'$.
   - if $\tau = L$, from the rule (ALLOCATE), $\Gamma, \lambda(\ell) \vDash e : L$ and from Simple Security $\mu(e)$=$\nu(e)$. So, by the SOS rule (CALLOC), $\mu'(T)$=$\nu'(T)$. So, $\mu' \sim_\Gamma \nu'$.
4. Case $\texttt{skip}^\ell$. From the SOS rule (NO-OP), $\mu = \mu'$ and $\nu = \nu'$. So, $\mu' \sim_\Gamma \nu'$.
5. Case $P = \texttt{if}^\ell\ e\ \texttt{then}\ P_1\ \texttt{else}\ P_2$. If $(\Gamma, \ell) \vDash_\lambda e : L$ then $\mu(e) = \nu(e)$ by Simple Security. If $\mu(e) \neq 0$ then $(\mu, P) \rightarrow (\mu, P_1) \rightarrow^* \mu'$ and $(\nu, P) \rightarrow (\nu, P_1) \rightarrow^* \nu'$. From the typing rule (IF), we have then $\Gamma \vDash_\lambda P_1 : L\ cmd$. In addition, from Proposition 1 , $\lambda(P_1) \geq \mu$ and $\lambda(P_1) \geq \nu$ and from Lemma 2, P_1 is well-labelled by λ, then by induction, $\mu' \sim_\Gamma \nu'$. The case $\mu(e) = 0$ is similar.
   If instead $\Gamma, \lambda(\ell) \vDash e : H$, then from the typing rule (IF), we have $\Gamma \vDash_\lambda P : H\ cmd$. Then by Corollary 1, $\mu \sim_\Gamma \mu'$ and $\nu \sim_\Gamma \nu'$. So $\mu' \sim_\Gamma \nu'$.
6. Case $\texttt{while}^\ell\ e\ \texttt{do}\ c'$. Similar to the `if` case.
7. Case $P = {c_1}^\ell; P_2$. If $(({c_1}^\ell; P_2), \mu) \rightarrow^j \mu'$, then by Lemma 4 there exist k and μ" such that $0 < k < j$, $({c_1}^\ell, \mu) \rightarrow^k \mu$", $({c_1}^\ell; P_2, \mu) \rightarrow^k (P_2, \mu")$ and $(P_2, \mu") \rightarrow^{j-k} \mu'$. Similarly, if $(({c_1}^\ell; P), \nu) \rightarrow^{j'}, \nu'$, then there exist k' and ν" such that $0 < k' < j'$, $({c_1}^\ell, \nu) \rightarrow^{k'} \nu$", $({c_1}^\ell; P_2, \nu) \rightarrow^{k'} (P_2, \nu")$ and $(\nu", P_2) \rightarrow^{j'-k'} \nu'$. Since P is well-labelled by λ, by the rule (O-SEQ), c_1 and P_2 are well-labelled by λ. In addition, from Lemma 1, $\lambda(P_2) \geq \mu$" and $\lambda(P_2) \geq \nu$". By induction, $\mu" \sim_\Gamma \nu"$. So by induction again, $\mu' \sim_\Gamma \nu'$.

Corollary 2. *Suppose that P is well-labelled, $\Gamma \vDash_\lambda P : \tau\ cmd$ and $\mu \sim_\Gamma \nu$ and $lab(P) = true$ and $(c, \mu) \rightarrow^* \mu'$ and $(c, \nu) \rightarrow^* \nu'$ then $\mu' \sim_\Gamma \nu'$.*

6 Conclusion and future work

In this paper, we have proposed a type system for an accurate information flow analysis of programs with arrays. Our type system is based on a pre-computed approximation of integer variables manipulated by the program. We believe that our approach can be extended to array aliases as in [17] as well as multi-dimensional arrays.

We have developed a prototype which, from a program P a labelling λ and some initial typing, determines if P is well-labelled and when possible types the program P respecting the initial typing by giving the minimal type. To improve this prototype, the next step will be the integration of a module for automatic generation of a well-labelling: the main ingredient for this will be an invariant generator such as in [10].
The approach we presented is flow-insensitive (the relative order of commands is irrelevant for typing); it is known that flow sensitive approach are more accurate and allows to consider polymorphic data structures since a specific type is inferred for each variable at each program point [8,4]. However, our work is fully compatible with the framework proposed in [11] (as our types can trivially be organised as lattices), allowing to transform our type system into a flow-sensitive one.

As we explain, we based the well-labelling of programs only on the values of integer variables ignoring the values contained in array cells. Recent works try to address the issue of reasonning about array contents [9,13]. It could be interesting to know whether this kind of works can be combined directly with our type system.

References

1. T. Amtoft, J. Hatcliff, and E. Rodríguez. Precise and automated contract-based reasoning for verification and certification of information flow properties of programs with arrays. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010*, volume 6012 of *Lecture Notes in Computer Science*, pages 43–63. Springer, 2010.
2. A. Annichini, A. Bouajjani, and M. Sighireanu. Trex: A tool for reachability analysis of complex systems. In *Proceedings of the 13th International Conference on Computer Aided Verification*, CAV '01, pages 368–372, London, UK, 2001. Springer-Verlag.
3. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolikc transition systems. In *Computer Aided Verification, 15th International Conference, CAV 2003,*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.
4. E. Bonelli, A. Compagnoni, and R. Medel. Information flow analysis for a typed assembly language with polymorphic stacks. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, 2005,*, volume 3956 of *LNCS*, pages 37–56. Springer, 2005.
5. Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, pages 115–. IEEE Computer Society, 2004.
6. D. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
7. D. Denning and P. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
8. S. Genaim and F. Spoto. Information flow analysis for java bytecode. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005,Proceedings*, volume 3385 of *LNCS*, pages 346–362. Springer, 2005.
9. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation,*, pages 339–348. ACM, 2008.
10. T. Henzinger, T. Hottelier, and L. Kovács. Valigator: A verification tool with bound and invariant generation. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008,*, volume 5330 of *Lecture Notes in Computer Science*, pages 333–342. Springer, 2008.
11. S. Hunt and D. Sands. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, pages 79–90. ACM, 2006.
12. A. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
13. V. Perrelle and N. Halbwachs. An analysis of permutations in arrays. In *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010,*, volume 5944 of *LNCS*, pages 279–294. Springer, 2010.
14. F. Pottier and V. Simonet. Information flow inference for ml. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.
15. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.
16. D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
17. J. Yao and J. Li. Discretional array operations for secure information flow. *Journal of Information and Computing Science*, 1(2):67 – 77, 2007.