Lightweight Information Flow

Flemming Nielson(✉) and Hanne Riis Nielson

Department of Mathematics and Computer Science,
Technical University of Denmark, 2800 Kgs., Lyngby, Denmark
fnie@dtu.dk, hrni@dtu.dk

Abstract. We develop a *type system* for identifying the information flow between variables in a program in the Guarded Commands language. First we characterise the types of information flow that may arise between variables in a non-deterministic program: *explicit*, *implicit*, *bypassing*, *correlated* or *sanitised*. Next we allow to specify security policies in a number of traditional ways based on mandatory access control: defining a *security lattice*, working with *components* or *decentralised labels*, both as pertains to *confidentiality* and *integrity*. Offending information flows are those identified by the type system and that violate the security policy; a program is *sufficiently secure* if it contains only acceptable information flows.

1 Introduction

Motivation. Much of the work of Rocco De Nicola has been within the general area of process algebras [5]. This is a fascinating area containing a wide range of fundamental ideas and many deep developments on topics such as semantics, equivalences (including testing equivalences [3,10] and bisimulations [1,12]) and model checking [11] to name just some of the key ones.

Some of the work of Rocco De Nicola has been on type systems ensuring various desirable properties of systems, including security properties [7,8]. In order to make these developments accessible to the wider computer science and computer engineering communities it is essential to choose the primitives of the process algebras at an appropriate level of abstraction. The work on *Klaim* (a Kernel Language for Agents Interaction and Mobility) [2,6,9] incorporates a choice that is sufficiently abstract to allow a rich theory and prototype systems to be developed, while at the same time being sufficiently concrete to appeal to a wide variety of researchers, engineers, programmers and students.

Contribution. In this paper we define a type system for identifying the security vulnerabilities that may arise in non-deterministic programs.

The traditional approach is to define a type system that intends to ensure that there are absolutely no security violations in well-typed programs. Non-interference results, or generalisations of these, then provide guarantees about the soundness of the type system. However, this does not close the loophole that security vulnerabilities may exist below the level of formalisation, as when

M. Boreale et al. (Eds.): De Nicola-Festschrift, LNCS 11665, pp. 455–470, 2019.
https://doi.org/10.1007/978-3-030-21485-2_25

timing attacks can still be performed on systems only achieving non-termination-sensitive security, nor that minor amounts of quantitative leakage might be acceptable in practice.

Our approach is to define a type system that intends merely to identify the security vulnerabilities that may still be present in well-typed programs. The aim is to do so in a manner that appeals to a wide variety of researchers, engineers, programmers and students, also outside the area of language based security. The acceptability of the security vulnerabilities should then be assessed as part of a code review.

The first step is to characterise the types of information flow that may arise between variables in non-deterministic programs: they may be *explicit* (as in assignments), *implicit* (as in conditional choices), *bypassing* (when one conditional choice may bypass another), *correlated* (when variables are modified in the same conditional branches that could have been bypassed) or *sanitised* (when the flow is regarded as permissible regardless of the security policy).

The next step is to admit security policies in a number of traditional ways based on mandatory access control: explicitly defining a *security lattice*, working with *components* or *decentralised labels*, both as pertains to permissions (often used for *confidentiality*) and restrictions (often used for *integrity*). This step is fairly standard.

The final step is to develop a *type system* for identifying the information flow between variables in a program in the Guarded Commands language. Offending information flows are those identified by the type system and that violate the security policy. A program is secure if it contains no information flows or only acceptable ones (like the sanitised ones); in the absence of any information flow we may establish a non-interference result but we would be more interested in being able to quantify the amount of leakage so as to provide guidance to engineers and programmers as part of a code review. The type system has been implemented and is available for experimentation at http://FormalMethods.dk/if4fun and makes use of heuristics for satisfiability of boolean expressions in Guarded Commands.

2 Guarded Commands for Security

We shall base our development on Dijkstra's language of Guarded Commands [15] extended with arrays and a security primitive (`san`, to be explained shortly). The conditional takes the form `if` $b_1 \rightarrow C_1$ `[]` ... `[]` $b_k \rightarrow C_k$ `fi`; as an example, to express that C_1 should be executed when b holds and that otherwise C_2 should be executed, we shall write `if` $b \rightarrow C_1$ `[]` $\neg b \rightarrow C_2$ `fi`. The iteration construct takes the form `do` $b_1 \rightarrow C_1$ `[]` ... `[]` $b_k \rightarrow C_k$ `od`; as an example, to express that C should be executed as long as b holds, we shall write `do` $b \rightarrow C$ `od`.

An example program is shown in the righthand half of Fig. 1; it non-deterministically updates the entries of arrays `A` and `E`, of lengths `A#` and `E#`, respectively. (It can be seen as an interleaved version of the parallel composition of two programs handling each of the arrays, as shown in the lefthand half of Fig. 1, but this is not part of the Guarded Commands language.)

```
par
   a:=0;
   do san(a)<san(A#) →
      A[a]:=A[a]+27;
      a:=a+1
   od
[]
   e:=0;
   do san(e)<san(E#) →
      E[e]:=E[e]+12;
      e:=e+1
   od
rap
```

```
a:=0;
e:=0;
do san(a)<san(A#) →
   A[a]:=A[a]+27;
   a:=a+1
[] san(e)<san(E#) →
   E[e]:=E[e]+12;
   e:=e+1
od
```

Fig. 1. Two arrays being simultaneously updated. On the left we pretend it happens in parallel, on the right we pretend it happens interleaved. Only the program on the right is within Guarded Commands for Security as studied here.

Syntax. The syntax of the commands C and guarded commands GC of the Guarded Commands for Security language are mutually recursively defined using the following BNF notation:

$$C ::= x := a \mid A[a_1] := a_2 \mid \texttt{skip} \mid C_1 \,;\, C_2 \mid \texttt{if}\ GC\ \texttt{fi} \mid \texttt{do}\ GC\ \texttt{od}$$
$$GC ::= b \rightarrow C \mid GC_1\ \texttt{[]}\ GC_2$$

We make use of arithmetic expressions a (used in $x := a$ and $A[a_1] := a_2$) and boolean expressions b (used as a guard for when to execute a command C as in $b \rightarrow C$) given by

$$a ::= n \mid s \mid x \mid A[a_1] \mid A\# \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 * a_2 \mid \texttt{san}\ a_1$$
$$b ::= \texttt{true} \mid a_1 = a_2 \mid a_1 > a_2 \mid a_1 \geq a_2 \mid b_1 \wedge b_2 \mid b_2 \mid \neg\, b_1$$

where numbers n, strings s, variables x and arrays A are left unspecified. The `san` construct will be used to bypass the security policy and will be explained in Sect. 3.

Semantics. We shall present the key ideas behind giving an operational semantics for Guarded Commands.

A command will be interpreted relative to a memory σ that assigns values (say integers) to all variables and array entries in the command of interest. More precisely, for each array A of length $A\#$ the memory will provide values for $A[0] \cdots A[A\#]$ as well as for $A\#$.

An arithmetic expression a is then evaluated with respect to a memory σ and we obtain a value $\mathcal{A}[\![a]\!]\sigma$ as result. Evaluation is undefined if the arithmetic expression accesses a variable or array entry for which the memory does not assign a value. Also, the value of `san` a_1 is the same as the value of a_1.

A boolean expression b is also evaluated with respect to a memory σ and we obtain a truth value $\mathcal{B}[\![b]\!]\sigma$ as result. Evaluation is undefined if one of the constituent arithmetic expressions is undefined.

We then define an operational semantics for interpreting commands (and guarded commands) and list some of the key axioms and rules:

$$\frac{\sigma(x) \text{ and } \mathcal{A}[\![a]\!]\sigma \text{ are defined}}{(x := a, \sigma) \to \sigma[x \mapsto \mathcal{A}[\![a]\!]\sigma]}$$

$$\frac{(C_1, \sigma) \to (C_1', \sigma')}{(C_1\,; C_2, \sigma) \to (C_1'\,; C_2, \sigma')} \qquad \frac{(C_1, \sigma) \to \sigma'}{(C_1\,; C_2, \sigma) \to (C_2, \sigma')}$$

$$\frac{\mathcal{B}[\![b_i]\!]\sigma \text{ is true}}{(\texttt{if } b_1 \to C_1 \texttt{ [] } \cdots \texttt{ [] } b_n \to C_n \texttt{ fi}, \sigma) \to (C_i, \sigma)}$$

$$\frac{\mathcal{B}[\![b_i]\!]\sigma \text{ is true}}{(\texttt{do } \cdots \texttt{ [] } b_i \to C_i \texttt{ [] } \cdots \texttt{ od}, \sigma) \to (C_i\,; \texttt{do } \cdots \texttt{ [] } b_i \to C_i \texttt{ [] } \cdots \texttt{ od}, \sigma)}$$

$$\frac{\text{all of } \mathcal{B}[\![b_i]\!]\sigma \text{ are false}}{(\texttt{do } b_1 \to C_1 \texttt{ [] } \cdots \texttt{ [] } b_n \to C_n \texttt{ od}, \sigma) \to \sigma}$$

In particular, this semantics is purely non-deterministic and does not make use of a scheduler. (If needed, we would model a scheduler by explicitly modifying the guards in guarded commands. Doing so would influence the results of the information flow type system. This would be our way of modelling an attacker that might collude with a scheduler.)

3 Types of Information Flow

We now introduce the types of information flow in non-deterministic programs. These are not only between variables as we also have array entries and array lengths. We therefore use the term *data container* to stand for any one of variable, array entry or length. The notions of explicit, implicit and sanitised flows are standard [13,14] whereas our treatment of bypassing flows grew out of [21] and our focus on correlation flows is more novel.

Explicit Flows. In the command $\texttt{y} := \texttt{x}$ there is a direct and explicit flow from $\texttt{x}$ to $\texttt{y}$. We write this as $\texttt{x} \rightarrow^{\mathsf{E}} \texttt{y}$ to indicate that it is an explicit flow.

In general, *explicit* flows arise whenever a data container is used to compute the value of a data container.

A slightly more complex example is the command $\texttt{y} := \texttt{x}\,;\ \texttt{z} := \texttt{y}$ where there are direct explicit flows from $\texttt{x}$ to $\texttt{y}$ and from $\texttt{y}$ to $\texttt{z}$. The flow from $\texttt{x}$ to $\texttt{z}$ is an *indirect* flow, and in general we use indirect to indicate that we exploit the transitive nature of the flow relation. As for the type of flow we shall say that the indirect flow is also explicit.

Implicit Flows. In the guarded command $\texttt{x} = 0 \rightarrow \texttt{y} := 0$ there is a direct and implicit flow from x to y. We write this as $\texttt{x} \rightarrow^{\mathsf{I}} \texttt{y}$ to indicate that it is an implicit flow.

In general, *implicit* flows arise whenever a data container is modified inside the body of a command governed by a boolean condition containing some data container.

The command $\texttt{if x} = 0 \rightarrow \texttt{y} := 0 \,[]\, \neg(\texttt{x} = 0) \rightarrow \texttt{y} := 1 \,\texttt{fi};\ \texttt{z} := \texttt{y}$ has a direct implicit flow from x to y and a direct explicit flow from y to z. The flow from x to z is an indirect flow and as for the type of flow we shall say that the indirect flow is implicit (since x is not directly copied into z).

Bypassing Flows. In $\texttt{y} := 0;\ \texttt{if x} = 0 \rightarrow \texttt{skip} \,[]\, \texttt{true} \rightarrow \texttt{y} := 1 \,\texttt{fi}$ there are no explicit flows from x to y and also no implicit flows. However, it is still the case that the final value of y might reveal something about x if one is able to run the program many times and observe the different non-deterministic outcomes. We write this as $\texttt{x} \rightarrow^{\mathsf{B}} \texttt{y}$ to indicate that it is a bypassing flow.

In general, *bypassing* flows arise whenever two conditions can be simultaneously true and more than one branch can be taken; in this case there is a bypassing flow from the data containers in the condition of one branch to the data containers modified in the command of the other.

In the command $\texttt{x} := \texttt{z};\ \texttt{y} := 0;\ \texttt{if x} = 0 \rightarrow \texttt{skip} \,[]\, \texttt{true} \rightarrow \texttt{y} := 1 \,\texttt{fi}$ there is a direct explicit flow from z to x and a direct bypassing flow from x to y. The flow from z to y is an indirect flow and as for the type of flow we shall say that the indirect flow is a bypassing one.

Correlation Flows. Bypassing flows capture some of the power of non-determinism but not all of it. In $\texttt{if true} \rightarrow \texttt{y} := 0\,;\texttt{x} := 0 \,[]\, \texttt{true} \rightarrow \texttt{y} := 1\,;\texttt{x} := 1 \,\texttt{fi}$ there are no explicit, implicit or bypassing flows. Yet, if y was intended to be a private key (albeit a short one) and x is a public variable, then clearly we can learn something about y from knowing x. We write this as $\texttt{x} \rightarrow^{\mathsf{C}} \texttt{y}$ and $\texttt{y} \rightarrow^{\mathsf{C}} \texttt{x}$ to indicate that it is a possible correlation flow between x and y.

In general, *correlation* flows arise whenever two conditions can be simultaneously true and the choice of branch is resolved non-deterministically; in this case there is a correlation flow between the data containers modified in each branch.

In $\texttt{if true} \rightarrow \texttt{y} := 0\,;\texttt{x} := 0 \,[]\, \texttt{true} \rightarrow \texttt{y} := 1\,;\texttt{x} := 1 \,\texttt{fi};\ \texttt{z} := \texttt{x}$ there is a direct explicit flow from x to z and a correlation flow from y to x. The flow from y to z is an indirect flow and as for the type of flow we shall say that the indirect flow is a correlation one.

Sanitised Flows. Returning to the non-deterministic program in Fig. 1 there would seem to be bypassing flows from A# to E[] and similarly from E# to A[]. We might consider these flows to be absolutely unproblematic and a traditional approach is to use *sanitisation* for this; in our case this means using the san construct of Guarded Commands for Security as illustrated in Fig. 1.

Rather than *neglecting* the direct bypassing flows from $\mathtt{A}\#$ to $\mathtt{E}[\,]$ and from $\mathtt{E}\#$ to $\mathtt{A}[\,]$ we shall mark these as sanitised flows (that can be disregarded later as part of a code review) and we write $\mathtt{A}\# \rightarrow^{\mathsf{S}} \mathtt{B}[\,]$ and $\mathtt{B}\# \rightarrow^{\mathsf{S}} \mathtt{A}[\,]$.

In general, *sanitised* flows arise whenever at least one sanitisation step is involved in the flow. In line with previous decisions, if a sequence of flows involve a sanitised flow we shall regard the overall flow as a sanitised one.

Representation of Flows. We shall take the point of view that some types of flows are more worrying than others and that we only need to record the most worrying one. We order the explicit (E), implicit (I), bypassing (B), correlation (C) and sanitised (S) flows linearly by $\mathsf{S} < \mathsf{C} < \mathsf{B} < \mathsf{I} < \mathsf{E}$. We then use max and min for the corresponding least upper bound and greatest lower bound operations.

A *flow relation* is a partial map from pairs of data containers to $\{\mathsf{E}, \mathsf{I}, \mathsf{B}, \mathsf{C}, \mathsf{S}\}$, and we use F to range over flow relations and τ to range over types of flows. (This is isomorphic to a total map from pairs of data containers to $\{\bot, \mathsf{E}, \mathsf{I}, \mathsf{B}, \mathsf{C}, \mathsf{S}\}$ where $\bot < \mathsf{S} < \mathsf{C} < \mathsf{B} < \mathsf{I} < \mathsf{E}$ and thus gives rise to a pointwise definition of a partial order $\leq$ between flow relations.)

We write $\delta_1 \rightarrow^{\tau} \delta_2$ for the flow relation that is undefined everywhere, except that the pair (δ_1, δ_2) is mapped to τ.

Extending this notation to sets of data containers we write $\Delta_1 \rightrightarrows^{\tau} \Delta_2$ for the flow relation that is undefined everywhere, except that a pair $(\delta_1, \delta_2) \in \Delta_1 \times \Delta_2$ is mapped to τ.

As a special case, $\{\,\} \rightrightarrows^{\tau} \{\,\}$ denotes the flow relation that is undefined everywhere (regardless of the choice of τ).

4 Security Policies

The key motivation behind our development is to classify data containers according to a security domain, and to consider it secure to transfer data as expressed by an ordering on the elements of the security domain (see [17] for a general introduction).

A *security domain* $\boldsymbol{L}$ is a finite and non-empty set equipped with a preorder $\sqsubseteq$; this is a relation over $\boldsymbol{L}$ that is reflexive and transitive. The preorder indicates the direction in which it is secure to move data along; we shall use this approach regardless of whether we deal with confidentiality or integrity or mixtures or modifications of these. In the literature, the security domain is often required to be a (complete) lattice and hence is called a *security lattice*, but we do not need this assumption for our approach.

A *security association* $\mathcal{L}$ is a mapping from the set of data containers of interest into the security domain. Clearly security policies can be combined using cartesian products and hence be built in a compositional manner.

A *security policy* consists of a security domain and a security association.

An information flow $\delta_1 \rightarrow^{\tau} \delta_2$ is secure with respect to the security policy whenever $\mathcal{L}(\delta_1) \sqsubseteq \mathcal{L}(\delta_2)$. An information flow $\delta_1 \rightarrow^{\tau} \delta_2$ with $\mathcal{L}(\delta_1) \not\sqsubseteq \mathcal{L}(\delta_2)$ constitutes a security violation at level τ.

Fig. 2. Illustrating restriction ordering ($\subseteq$) versus the permission ordering ($\supseteq$).

Components. Describing the security domain explicitly becomes cumbersome once the security domain grows in size. We therefore consider ways of expressing the intended security lattice in more succinct ways following the approach standard in Mandatory Access Control [17].

Define a finite and nonempty set C of *security categories*. A *security component* then is a set of security categories and the *security domain* $\boldsymbol{L} = \mathsf{PowerSet}(\mathsf{C})$ is the set of all such security components. This security domain is indeed a (complete) lattice.

Whenever the security categories are considered to be *restrictions* that can be gained but cannot be lost, the security domain will be ordered by the subset ordering (taking $\sqsubseteq$ to be $\subseteq$). This is often the case for integrity policies.

Whenever the security categories are considered to be *permissions* that can be lost but cannot be gained, the security domain will be ordered by the superset ordering (taking $\sqsubseteq$ to be $\supseteq$). This is often the case for confidentiality policies.

The *security domain* is then specified by listing the finite and nonempty set of security categories and indicating whether to use the ordering for restrictions or for permissions.

In both cases we retain the important principle that data may flow along the preorder of the security domain. Determining which choice of ordering to go for depends on determining whether or not it is considered to be secure to gain or lose security categories along flows.

We shall write $\{*\}$ for the security component consisting of all security categories. In case of the restriction ordering, the least element then is $\{\,\}$ and the greatest element is $\{*\}$, whereas in the case of the permission ordering, the least element is $\{*\}$ and the greatest element is $\{\,\}$. This is illustrated in Fig. 2.

Decentralised Labels. The security perspective of components was of a rather global nature. To accommodate that different security principals might have different views on information flow, that all should be respected, we develop a notion of decentralised labels – motivated by the Decentralised Label Model of Myers and Liskov [18] and some of its adaptations [19,22] – while staying fully within the lattice-based approach.

Define a finite and nonempty set P of *security principals*. A *decentralised label* then is a total mapping from P to $\mathsf{PowerSet}(\mathsf{P})$ and the *security domain* $\boldsymbol{L} = (\mathsf{P} \to \mathsf{PowerSet}(\mathsf{P}))$ is the set of all such mappings. This security domain is indeed a (complete) lattice.

As in the previous section there are two ways of considering the labels: as restrictions that can be gained but cannot be lost, or as permissions that can be lost but cannot be gained.

Whenever the labels are considered to be *restrictions* that can be gained but cannot be lost, the security domain will be ordered as $\ell_1 \sqsubseteq \ell_2$ if and only if $\forall R \in \mathsf{R} : \ell_1(R) \subseteq \ell_2(R)$. This is often the case for integrity policies.

Whenever the security categories are considered to be *permissions* that can be lost but cannot be gained, the security domain will be ordered as $\ell_1 \sqsubseteq \ell_2$ if and only if $\forall R \in \mathsf{R} : \ell_1(R) \supseteq \ell_2(R)$. This is often the case for confidentiality policies.

The *security domain* is then specified by listing the finite and nonempty set of security categories and indicating whether to use the ordering for restrictions or for permissions.

In both cases we retain the important principle that data may flow along the preorder of the security domain. Determining which choice of ordering to go for depends on determining whether or not it is considered to be secure to gain or lose security categories along flows.

We shall allow to write $*$ for the list of all the security principals in P. In case of the restriction ordering, the least element then is $[* \mapsto \{\,\}]$ and the greatest element is $[* \mapsto \{*\}]$, whereas in the case of the permission ordering, the least element is $[* \mapsto \{*\}]$ and the greatest element is $[* \mapsto \{\,\}]$.

5 Information Flow Type System

We are now ready to develop the analysis for over-approximating the set of flows that may occur in a program. This takes the form of an inference system for defining a judgement $\vdash C : F$ associating the command C with the flow F and similarly for guarded commands. It may be seen as a generalisation of the approach of [24] to a non-deterministic language.

We shall use $\mathbf{fv}(a)$ to denote those data containers that occur in a outside any `san` construct and use $\mathbf{sv}(a)$ to denote those data containers that occur in a inside one or more `san` constructs. Similarly for $\mathbf{fv}(b)$ and $\mathbf{sv}(b)$. Finally, we shall use $\mathbf{mv}(C)$ to denote those data containers that may be modified within the command C; this generally represents an over-approximation of those modified in any execution.

Simple Assignments. For a simple assignment to a variable we need to record a flow from the data containers in the arithmetic expression to the variable being modified. These flows will be explicit unless the data container in question occurs inside at least one `san` construct. This motivates the following axiom scheme.

$$\vdash x := a : \begin{array}{l}(\mathbf{fv}(a) \rightrightarrows^{\mathsf{E}} \{x\}) \oplus \\ (\mathbf{sv}(a) \rightrightarrows^{\mathsf{S}} \{x\})\end{array}$$

The operation $\oplus$ is defined by

$$(F_1 \oplus F_2)(\delta_1, \delta_2) = \max\{F_1(\delta_1, \delta_2), F_2(\delta_1, \delta_2)\}$$

and incorporates the idea that whenever we have a choice between two different types of flow between two data containers we should always choose the most worrying one. (In case any of the $F_i(\delta_1, \delta_2)$ being undefined we revert to the isomorphic representation using total maps explained earlier and this amounts to disregarding such cases.)

This definition can also be seen as a pointwise addition of matrices where max plays the role of addition. It is immediate that both $F \oplus (\{\,\} \rightrightarrows^{\tau} \{\,\})$ and $(\{\,\} \rightrightarrows^{\tau} \{\,\}) \oplus F$ equal F.

Assignments to Arrays. For assignment to arrays we take the point of view that the data containers inside the index give rise to implicit rather than explicit flows. This is based on the consideration that for an array `A` having 5 elements the command `if` $a_1 = \texttt{1} \rightarrow \texttt{A[1]} := a_2$ `[]` $\cdots$ `[]` $a_1 = \texttt{5} \rightarrow \texttt{A[5]} := a_2$ `fi` is equivalent to the program $\texttt{A}[a_1] := a_2$. This motivates the following axiom scheme.

$$\vdash A[a_1] := a_2 : \begin{array}{l} (\mathbf{fv}(a_2) \rightrightarrows^{\mathsf{E}} \{A[\,]\}) \oplus \\ (\mathbf{fv}(a_1) \rightrightarrows^{\mathsf{I}} \{A[\,]\}) \oplus \\ (\mathbf{sv}(a_1) \cup \mathbf{sv}(a_2) \rightrightarrows^{\mathsf{S}} \{A[\,]\}) \end{array}$$

Skip. The axiom for skip is immediate: $\vdash \texttt{skip} : (\{\,\} \rightrightarrows^{\mathsf{E}} \{\,\})$.

Sequencing. For sequential composition $C_1\,;C_2$ we need to compose the flows arising from C_1 and C_2. However, as our designation of the set of data containers modified represents an over-approximation, and as we have not required information flows to contain explicit flows from a data container to itself, we need to take care to also include the flows from each of the components.

$$\frac{\vdash C_1 : F_1 \qquad \vdash C_2 : F_2}{\vdash C_1\,;C_2 : (F_1 \otimes F_2) \oplus F_1 \oplus F_2}$$

The operation $\otimes$ is defined by

$$(F_1 \otimes F_2)(\delta_1, \delta_2) = \max\left\{ \min\left\{ \begin{array}{c} F_1(\delta_1, \delta), \\ F_2(\delta, \delta_2) \end{array} \right\} \mid \delta \text{ is a data container} \right\}$$

and incorporates the following two ideas: (1) If we have flows $\delta_1 \rightarrow^{\tau_1} \delta_2$ and $\delta_2 \rightarrow^{\tau_2} \delta_3$ then we also have a flow $\delta_1 \rightarrow^{\tau} \delta_3$ where τ is the least worrying of the two flows, i.e. $\tau = \min\{\tau_1, \tau_2\}$. (2) If additionally there is another scenario where we have flows $\delta_1 \rightarrow^{\tau_1'} \delta_2'$ and $\delta_2' \rightarrow^{\tau_2'} \delta_3$ then we want τ to be the most worrying of the two candidates $\min\{\tau_1, \tau_2\}$ and $\min\{\tau_1', \tau_2'\}$, i.e. $\tau = \max\{\min\{\tau_1, \tau_2\}, \min\{\tau_1', \tau_2'\}\}$.

This definition can also be seen as a matrix multiplication where max plays the role of addition and min plays the role of multiplication. It is immediate that both $F \otimes (\{\,\} \rightrightarrows^{\tau} \{\,\})$ and $(\{\,\} \rightrightarrows^{\tau} \{\,\}) \otimes F$ equal $\{\,\} \rightrightarrows^{\tau} \{\,\}$.

Conditional. For conditional most of the work is left to the analysis of the guarded command inside.

$$\frac{\vdash GC : F}{\vdash \texttt{if}\ GC\ \texttt{fi} : F}$$

Iteration. In the case of iteration we need to take the transitive closure to reflect the iterative nature.

$$\frac{\vdash GC : F}{\vdash \texttt{do}\ GC\ \texttt{od} : F^{\circledast}}$$

The operation $\circledast$ is defined by

$$F^{\circledast}(\delta_{\triangleright}, \delta_{\blacktriangleleft}) = \max \left\{ \min \left\{ \begin{array}{c} F(\delta_0, \delta_1), \\ \cdots, \\ F(\delta_{n-1}, \delta_n) \end{array} \right\} \mid \begin{array}{l} \delta_0, \cdots, \delta_n \text{ are data containers,} \\ n > 0, \delta_0 = \delta_{\triangleright}, \delta_n = \delta_{\blacktriangleleft} \end{array} \right\}$$

and incorporates the idea that the iteration can be performed any number of times. The definition can also be seen as a form of transitive closure of a matrix.

Guarded Commands. For a guarded command $b_i \rightarrow C_i$ we need to record the implicit flow from the condition b_i to the command C_i as well as incorporate the flows arising from C_i. Some of the implicit flows will actually be sanitised flows in case the data container inside b_i occurs within at least one `san` construct. In the context of a guarded command $b_1 \rightarrow C_1\ \texttt{[]}\ \cdots\ \texttt{[]}\ b_n \rightarrow C_n$ with multiple choices these considerations account for the first line of the flow constructed in the rule below.

$$\frac{\vdash C_1 : F_1 \qquad \cdots \qquad \vdash C_n : F_n}{\vdash \begin{array}{l} b_1 \rightarrow C_1 \\ \texttt{[]} \cdots \texttt{[]} \\ b_n \rightarrow C_n \end{array} : \bigoplus_{i \leq n} \left(\begin{array}{c} \left(\mathbf{fv}(b_i) \rightrightarrows^{\mathsf{I}} \mathbf{mv}(C_i)\right) \oplus \left(\mathbf{sv}(b_i) \rightrightarrows^{\mathsf{S}} \mathbf{mv}(C_i)\right) \oplus F_i \oplus \\ \bigoplus_{j \in \mathbf{cosat}(i)} \left(\mathbf{fv}(b_j) \rightrightarrows^{\mathsf{B}} \mathbf{mv}(C_i)\right) \oplus \left(\mathbf{sv}(b_j) \rightrightarrows^{\mathsf{S}} \mathbf{mv}(C_i)\right) \\ \oplus \left(\mathbf{mv}(C_i) \rightrightarrows^{\mathsf{C}} \mathbf{mv}(C_i)\right) \end{array} \right)}$$

The remaining two lines take care of the additional complications arising in a non-deterministic language where more than one choice is possible. To express this we shall assume that '$j \in \mathbf{cosat}(i)$' over-approximates when $b_j \wedge b_i$ might be satisfiable for *different* choices of j and i, i.e. if $b_j \wedge b_i$ is satisfiable and $j \neq i$ then '$j \in \mathbf{cosat}(i)$' must be true. Whenever '$j \in \mathbf{cosat}(i)$' we create the bypassing flows possible, taking care of those that will actually be sanitised flows instead, and we create the correlation flows between all data containers modified in either body.

Offending Flows. Given a security policy $(\boldsymbol{L}, \mathcal{L})$ we can now obtain those flows that violate the security policy by means of the following rule.

$$\frac{\vdash C : F}{(\boldsymbol{L}, \mathcal{L}) \vdash C : F'} \quad \text{where } F'(\delta_1, \delta_2) = \begin{cases} F(\delta_1, \delta_2) & \text{if } \mathcal{L}(\delta_1) \not\sqsubseteq \mathcal{L}(\delta_2) \\ \text{undefined} & \text{otherwise} \end{cases}$$

A command C is said to be *offending at level* τ with respect to a security policy $(\boldsymbol{L}, \mathcal{L})$ whenever $(\boldsymbol{L}, \mathcal{L}) \vdash C : F$ and $F(\delta_1, \delta_2) \geq \tau$ for some (δ_1, δ_2). In practice, commands offending only at level S should be considered sufficiently secure if the code review reveals that the `san` construct has been used with due care.

Example 1. Consider again the program shown in the righthand half of Fig. 1 and suppose that the security domain $\boldsymbol{L} = \{\mathsf{cc}, \mathsf{aa}, \mathsf{ee}\}$ is ordered by $\mathsf{cc} \sqsubset \mathsf{aa}$ and $\mathsf{cc} \sqsubset \mathsf{ee}$. (You may read cc as clean, aa as Amazon and ee as eBay.)

If the security association has $\mathcal{L}(\mathtt{a}) = \mathsf{aa}$, $\mathcal{L}(\mathtt{e}) = \mathsf{ee}$, $\mathcal{L}(\mathtt{A}[\,]) = \mathsf{aa}$, $\mathcal{L}(\mathtt{E}[\,]) = \mathsf{ee}$, $\mathcal{L}(\mathtt{A}\#) = \mathsf{aa}$ and $\mathcal{L}(\mathtt{E}\#) = \mathsf{ee}$ then the only offending flows are sanitised. So the program is only offending at level S and should be considered sufficiently secure.

If the security association has $\mathcal{L}(\mathtt{a}) = \mathsf{cc}$, $\mathcal{L}(\mathtt{e}) = \mathsf{cc}$, $\mathcal{L}(\mathtt{A}[\,]) = \mathsf{aa}$, $\mathcal{L}(\mathtt{E}[\,]) = \mathsf{ee}$, $\mathcal{L}(\mathtt{A}\#) = \mathsf{cc}$ and $\mathcal{L}(\mathtt{E}\#) = \mathsf{cc}$ then we get the offending correlation flows $\mathtt{A}[\,] \rightarrow^{\mathsf{C}} \mathtt{a}$ and $\mathtt{E}[\,] \rightarrow^{\mathsf{C}} \mathtt{e}$ as well as offending sanitised flows. So the program is offending at level C which upon closer inspections might be considered not to be problematic.

Discussion of Soundness. There are other and more subtle ways in which information may flow than has been covered by our security analysis. The word *covert channel* is used to describe such phenomena. As an example, the program

$$\mathtt{y} := 0\,;\mathtt{x}' := \mathtt{x}\,;\mathtt{do}\ \mathtt{x}' > 0 \rightarrow \mathtt{x}' := \mathtt{x}' - 1\ [\!]\ \mathtt{x}' < 0 \rightarrow \mathtt{x}' := \mathtt{x}' + 1\ \mathtt{od}$$

always terminates. It has no flows of any kind from `x` to `y` but if we can observe the *execution time* it reveals some information about the absolute value of `x`. Similar examples can be constructed where the computation on `x` will only terminate successfully for some values of `x` and otherwise enter a loop or a stuck configuration. If we can observe the *non-termination* it also reveals some information about the value of `x`.

The above discussion may be construed to say that our type system is not sound. (But this holds for most published type systems for security: it is usually not too hard to find a finer semantics that allows observations disregarded when the type system was constructed.) This means that an engineer and programmer taking part in a code review must maintain a perspective on whether the covert channels not covered by the type system provide grounds for rejecting code exhibiting no offending flows.

Nonetheless it would be desirable to ensure the robustness of the type system against shortcomings other than the deliberate decision to ignore the covert channels mentioned above.

For this we would like to explore a quantitative approach based on entropy. The basic assumption is that we have joint probability distributions available to characterise how sets of data containers take their values. Shannon's *entropy* is then the expected value of information contained in each observation. An important derived concept is that of *conditional entropy*: the portion of the entropy of a data container that is independent from another data container.

There are two extreme cases of the conditional entropy. One extreme case is where the data containers are aliases for the same entity or are modified in exactly the same way. The other extreme case is where the data containers are

truly independent. The consideration of correlation flows were intended as an indicator of the first extreme case mentioned – but we are not close to be able to establish a result along these lines.

We find the quantitative approach more appealing than merely establishing a non-interference result [16,23,24] that guarantees how data containers of certain security classifications cannot influence data containers of another security classification. Using the developments in [20, Section 5.5] we may establish the following result. Suppose that $(\boldsymbol{L}, \mathcal{L}) \vdash C : \{\} \rightrightarrows^{\mathsf{S}} \{\}$ and $(C, \sigma_1) \rightarrow^* \sigma_1'$ and $(C, \sigma_2) \rightarrow^* \sigma_2'$; if $\forall y : \mathcal{L}(y) \sqsubseteq \mathcal{L}(x) \Rightarrow \sigma_1(y) = \sigma_2(y)$ we have $\sigma_1'(x) = \sigma_2'(x)$. However, $(\boldsymbol{L}, \mathcal{L}) \vdash C : \{\} \rightrightarrows^{\mathsf{S}} \{\}$ is likely to fail for non-deterministic programs since the nature of non-determinism is to open up for bypassing and correlation flows.

6 Algorithmic Issues

The type system is syntax-directed and easy to implement except for finding efficient ways to deal with transitive closure and satisfiability.

Transitive Closure. For an efficient construction of $F^{\circledast}$ using *dynamic programming* let us define

$$F^{[0]} = F \qquad F^{[n+1]} = F \oplus (F^{[n]} \otimes F^{[n]})$$

This is intended to ensure that $F^{[m]}$ correctly summarises the effect of all paths of length between 1 and 2^m.

Proposition 1. *If there are at most N data containers in the program considered then $F^{\circledast} = F^{[M]}$ where $M = \lceil log_2 N \rceil$.*

Proof. We may prove by induction that

$$F^{[m]}(\delta_\circ, \delta_\bullet) = \max \left\{ \min \left\{ \begin{array}{c} F(\delta_0, \delta_1), \\ \cdots, \\ F(\delta_{n-1}, \delta_n) \end{array} \right\} \mid \begin{array}{l} \delta_0, \cdots, \delta_n \text{ are data containers,} \\ 1 \leq n \leq 2^m, \delta_0 = \delta_\circ, \delta_n = \delta_\bullet \end{array} \right\}$$

and it then suffices to realise that we only need to consider paths of length between 1 and $N \leq 2^M$.

It is immediate that $F^{\circledast}(\delta_\circ, \delta_\bullet)$ is greater than or equal to $F^{[M]}(\delta_\circ, \delta_\bullet)$. If they are not equal there must be a sequence of data containers $\delta_\circ = \delta_0 = \cdots = \delta_n = \delta_\bullet$ with $n > N$ such that $\min\{F(\delta_0, \delta_1), \cdots, F(\delta_{n-1}, \delta_n)\}$ is not less than or equal to $F^{[M]}(\delta_\circ, \delta_\bullet)$. We proceed by contradiction and without loss of generality we may assume that n is as small as possible.

There must be a data container that occurs more than once in $\delta_\circ = \delta_0 = \cdots = \delta_n = \delta_\bullet$ so consider the reduced sequence obtained by omitting all data containers between the first and the last occurrence and retaining just one occurrence of the data container in question. The reduced sequence will provide a value τ of the $\min\{\cdots\}$ formula such that $\min\{F(\delta_0, \delta_1), \cdots, F(\delta_{n-1}, \delta_n)\}$ is less than or equal to τ, that is again less than or equal to $F^{[M]}(\delta_\circ, \delta_\bullet)$. This provides the desired contradiction.

Over-Approximating Satisfiability Using a DAG Construction. We next develop a heuristics for over-approximating whether or not two boolean expressions might be jointly satisfiable. Since the system at http://FormalMethods.dk/if4fun may be downloaded to personal devices and run locally we prefer this approach rather than recasting the problem as an SMT problem (Satisfaction Modulo Theories) that requires access to a solver such as Z3 [4].

Recall that we considered a construct $b_1 \rightarrow C_1$ [] $\cdots$ [] $b_n \rightarrow C_n$ and used the notation '$j \in \mathbf{cosat}(i)$' to over-approximate whether or not b_i and b_j can be jointly satisfied (for different choices of i and j). We shall define

$$(j \in \mathbf{cosat}(i)) = \big(\mathbf{sat}(b_i \wedge b_j) \wedge j \neq i\big)$$

and now explain our heuristics $\mathbf{sat}(\cdot)$ in Fig. 3 for over-approximating satisfiability.

```
function sat(b)
  convert b to disjunctive normal form ⋁_i b_1^i∧···∧b_{n_i}^i;
  global := false;
  iterating through all i do
    local := true;
    build the ordered DAG for b_1^i∧···∧b_{n_i}^i;
    if the DAG contains a marked node ¬t where also t is marked
      then local := false;
    if the DAG contains marked nodes t1 o1 t2 and t1 o2 t2
      with (o1, o2) ∈ E then local := false;
    global := global ∨ local;
  return global
```

Fig. 3. Algorithm for $\mathbf{sat}(b)$.

As a preparation we need to extend the syntax to use $<$ and $\leq$ (on top of $=$, $>$ and $\geq$) and to use $\vee$ (on top of $\wedge$ and $\neg$). Recall that a boolean expression is a *literal* when it has no occurrences of $\wedge$ or $\vee$ and at most one occurrence of $\neg$.

The first step in Fig. 3 is to translate b into disjunctive normal form; this is where $\vee$ may get introduced. The result is an equivalent formula

$$\bigvee_i b_1^i \wedge \cdots \wedge b_{n_i}^i$$

where each b_j^i is a literal.

Iterating through each conjunction of literals $b_1^i \wedge \cdots \wedge b_{n_i}^i$ the algorithm of Fig. 3 first constructs an ordered DAG (directed acyclic graph), and next inspects the ordered DAG to over-approximate satisfiability, as detailed below.

Constructing the Ordered DAG. To increase the amount of sharing in the ordered DAG we need to keep track of the 'transposed variants' of the arithmetic and relational operators:

$$\mathcal{T} = \{(+,+),(*,*),(<,>),(\leq,\geq),(=,=),(\geq,\leq),(<,>)\}$$

This takes care of characterising both those operators that are commutative (like $+$) and those that can be 'transposed' (like $a_1 < a_2$ may be transposed to $a_2 > a_1$). In general, whenever $(o_1, o_2) \in \mathcal{T}$ it must be the case that $t_1\, o_1\, t_2$ is equivalent to $t_2\, o_2\, t_1$.

Given a conjunction of literals we construct an ordered DAG by a bottom-up traversal over the parse tree. Leaves will be numbers n, strings s, variables x, arrays A, and `true`; internal nodes will be $[\,], \#, +, -, *, \mathtt{san}, <, \leq, =, >, \geq$ and $\neg$. Some of the nodes will be marked, and internal nodes will retain the order of their subgraphs.

When we encounter a potential new leaf in the bottom-up traversal over the parse tree of $b_1^i \wedge \cdots \wedge b_{n_i}^i$, we reuse the node in the DAG if it is already there, otherwise we construct a new leaf.

When we encounter a potential new internal node $t_1\, o_1\, t_2$, we reuse the node in the DAG if it is already there, otherwise we proceed as follows. If $(o_1, o_2) \in \mathcal{T}$ and there already is a node in the DAG for $t_2\, o_2\, t_1$, we use that node in the DAG, otherwise we construct the node $t_1\, o_1\, t_2$.

Once we encounter the root of one of the b_j^i we *mark* the node.

Inspecting the Ordered DAG. To detect cases where satisfiability fails we need to keep track of pairs of relational operators that exclude each other:

$$\mathcal{E} = \{(<,=),(<,\geq),(<,>),(\leq,>),(=,<),(=,>),(\geq,<),(>,<),(>,\leq),(>,=)\}$$

In general, whenever $(o_1, o_2) \in \mathcal{E}$ it must be the case that $a_1\, o_1\, a_2$ and $a_1\, o_2\, a_2$ are not jointly satisfiable for any choices of a_1 and a_2.

We can then establish the over-approximating nature of our heuristics.

Proposition 2. *If the boolean formula b is satisfiable then the algorithm* $\mathbf{sat}(b)$ *returns true.*

Proof. If the ordered DAG for a conjunction of literals contains a marked node t that has an ancestor $\neg\, t$ that is also marked, then clearly the conjunction of literals is not satisfiable. Similarly, if the ordered DAG for a conjunction of literals contains nodes t_1 and t_2 that have marked ancestors $t_1\, o_1\, t_2$ and $t_1\, o_2\, t_2$ with $(o_1, o_2) \in \mathcal{E}$, then the conjunction of literals is not satisfiable.

This shows that the resulting value of `local` for each iteration only reports false when the conjunction of literals is not satisfiable. It follows that the overall algorithm only reports false if none of the conjuncts of the disjunctive normal form are satisfiable.

We may conclude that '$j \in \mathbf{cosat}(i)$' is a correct over-approximation of joint satisfiability of b_i and b_j (for distinct i and j) from $b_1 \to C_1\ []\ \cdots\ []\ b_n \to C_n$.

7 Conclusion

We developed a type system for identifying the offending information flow between data containers in a program in the Guarded Commands language. It

was based on classifying flows as being explicit, implicit, bypassing, correlated or sanitised and on having general security policies incorporating multi-level security, components and decentralised labels; the bypassing and correlation flows were motivated by the need to deal with non-determinism. These developments are incorporated in the demonstration tool at http://FormalMethods.dk/if4fun; to allow it to be run on personal devices we make use of a heuristics for satisfiability of boolean expressions in Guarded Commands.

The approach taken in this paper has been inspired by working with engineers from safety critical software and observing how they react to incorporating security into their workflow. Ultimately this means leaving the decision of the acceptability of offending flows to the engineers and programmers taking part in a code review. The type support is intended to provide support for these decisions based on its classification of flows into the categories considered here.

References

1. Bernardo, M., De Nicola, R., Loreti, M.: Revisiting bisimilarity and its modal logic for nondeterministic and probabilistic processes. Acta Inf. **52**(1), 61–106 (2015)
2. Bettini, L., De Nicola, R., Pugliese, R.: XKlaim and Klava: programming mobile code. Electr. Notes Theor. Comput. Sci. **62**, 24–37 (2001)
3. Boreale, M., De Nicola, R.: Testing equivalence for mobile processes. Inf. Comput. **120**(2), 279–303 (1995)
4. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
5. De Nicola, R.: Testing equivalences and fully abstract models for communicating systems. Ph.D. thesis, University of Edinburgh, UK (1986)
6. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. IEEE Trans. Softw. Eng **24**(5), 315–330 (1998)
7. De Nicola, R., Ferrari, G.L., Pugliese, R., Venneri, B.: Types for access control. Theor. Comput. Sci. **240**(1), 215–254 (2000)
8. De Nicola, R., et al.: From flow logic to static type systems for coordination languages. Sci. Comput. Program. **75**(6), 376–397 (2010)
9. De Nicola, R., Gorla, D., Pugliese, R.: On the expressive power of Klaim-based calculi. Theor. Comput. Sci. **356**(3), 387–421 (2006)
10. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theor. Comput. Sci. **34**, 83–133 (1984)
11. De Nicola, R., Katoen, J.-P., Latella, D., Loreti, M., Massink, M.: Model checking mobile stochastic logic. Theor. Comput. Sci. **382**(1), 42–70 (2007)
12. De Nicola, R., Vaandrager, F.W.: Three logics for branching bisimulation. J. ACM **42**(2), 458–487 (1995)
13. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**(5), 236–243 (1976)
14. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977)
15. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)

16. Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, 26–28 April 1982, pp. 11–20. IEEE Computer Society (1982)
17. Gollmann, D.: Computer Security, 3rd edn. Wiley, Hoboken (2011)
18. Myers, A.C., Liskov, B.: Protecting privacy using the decentralized label model. ACM Trans. Softw. Eng. Methodol. **9**(4), 410–442 (2000)
19. Flemming Nielson and Hanne Riis Nielson: Atomistic Galois insertions for flow sensitive integrity. Comput. Lang. Syst. Struct. **50**, 82–107 (2017)
20. Nielson, F., Nielson, H.R.: Formal Methods: An Appetizer. Springer, Cham (2019)
21. Nielson, F., Nielson, H.R., Vasilikos, P.: Information flow for timed automata. In: Aceto, L., Bacci, G., Bacci, G., Ingólfsdóttir, A., Legay, A., Mardare, R. (eds.) Models, Algorithms, Logics and Tools. LNCS, vol. 10460, pp. 3–21. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63121-9_1
22. Nielson, H.R., Nielson, F.: Content dependent information flow control. J. Log. Algebr. Meth. Program. **87**, 6–32 (2017)
23. Volpano, D.M., Irvine, C.E.: Secure flow typing. Comput. Secur. **16**(2), 137–144 (1997)
24. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2/3), 167–188 (1996)