Segurança de Dados em Aplicações Centradas em Dados por Análise de Fluxo de Informação

Luísa Lourenço and Luís Caires

CITI e Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal

Resumo As aplicações centradas em dados e disponibilizadas na internet são cada vez mais pervasivas. Contudo, devido ao facto destas manipularem um volume elevado de dados, assim como ao meio em que estas são executadas ser por natureza inseguro, é frequente propiciaren falhas de segurança, nomeadamente relativamente à privacidade e a integridade dos dados que manipulam. O objectivo deste artigo é motivar, discutir e ilustrar uma técnica de análise de fluxo de informação sobre uma linguagem para modelar e programar aplicações centradas em dados com primitivas DML para manipulação e criação de entidades relacionais de dados. As contribuições técnicas deste trabalho são duas: por um lado, um estudo e caracterização precisa dos fluxos de informação inseguros que podem ocorrer em sistemas com primitivas típicas de uma DML; assim como a definição da respectiva propriedade de não-interferência.

1 Introdução

Nas últimas décadas as aplicações centradas em dados têm sido uma ferramenta importante no meio empresarial e, recentemente, no dia-a-dia do utilizador comum, devido à popularidade da internet e à difusão no seu acesso. Contudo, dado que estas aplicações manipulam um volume elevado de dados, assim como ao meio em que estas são executadas ser por natureza inseguro, é frequente encontrarem-se falhas de segurança neste tipo de aplicações. Por esse motivo, tem surgido nos últimos anos um forte interesse em desenvolver técnicas para garantir que as aplicações centradas em dados são, por construção (em tempo de desenvolvimento), seguras no que diz respeito aos dados que estas manipulam. Nesse sentido, propusemos em [1] uma pequena linguagem com primitivas de Linguagens de Manipulação de Dados (DML), juntamente com construções para criar e compor tarefas e *workflows*, com o intuito de modelar aplicações centradas em dados, que normalmente se apoiam no uso de bases de dados para guardar informação.

Neste artigo, apresentamos uma linguagem funcional λ_{DB} (baseada em [2] e [1]) que adiciona primitivas de DML a um cálculo lambda com registos, colecções e condicionais. O nosso objectvo consiste no desenvolvimento de um sistema de verificação estática, baseado num sistema de tipos, que permita garantir propriedades de segurança dos dados em tempo de compilação / construção, que estenda em muito a expressividade do sistema proposto em [2]. Para isso, vamos desenvolver uma análise de fluxo de informação, em vez do sistema de [2] que se baseia em tipo de refinamento, tarefa que apresenta vários desafios. Como primeiro passo, estudamos neste artigo uma

caracterização não trivial dos fluxos de informação inseguros que podem ocorrer em sistemas com primitivas típicas de uma DML; e apresentamos uma definição precisa da respectiva propriedade de não-interferência [3].

É interessante discutir a nossa abordagem geral através de um exemplo motivador. Assim começamos por introduzir λ_{DB} através de um cenário onde queremos gerir farmácias, o seu *stock* e as suas encomendas. Uma aplicação possível para esse fim teria como requisito que a informação relativa a medicamentos, farmacêuticos, o *stock* de cada farmácia, encomendas pendentes e empregados assim como informação relativa às próprias farmácias seja guardada, fisicamente, numa base de dados. Estes dados, porém, respeitam um modelo relacional que permite um acesso e manipulação mais fácil da informação guardada. Assim, neste cenário, precisamos das seguintes entidades relacionais

```
entity Pharmacies(id_pharm, name, location) in
entity Drugs(id_drug, name, type, prescriptionOnly) in
entity Stocks(id_pharm, id_drug, quantity) in
entity Orders(id_pharm, id_drug, quantity) in
entity Pharmacists(id_p, name, age, address) in
entity Employees(id_p, id_pharm, salary, rank) in (...)
```

que podem ser manipuladas através das habituais construções, ao estilo SQL, das DML: select, insert, delete e update. Portanto, por exemplo, se quisermos adicionar uma nova farmácia na base de dados, escrevemos **insert** `[id_pharm = 99, name =` "New Pharm", `location =` "Caparica"] **in** `Pharmacies`, que adiciona o tuplo `(99,` "New Pharm", "Caparica") na entidade `Pharmacies`.

Precisamos agora de fazer encomendas para construir o *stock* inicial da nova farmácia. Digamos que será, para cada medicamento, a média do *stock* de todas as farmácias. É necessário, então, uma colecção com os identificadores de todos os medicamentos, a qual podemos obter através de uma projecção da relação `Drugs`: **from** `(`x **in** `Drugs)` **where true select** x`.id_drug`. Vamos assumir que associamos o identificador `drugs` ao resultado da operação **select** anterior. Assim, se projectarmos a entidade `Stocks` para cada identificador de medicamento em `drugs`, então podemos obter todo o *stock* disponível em todas as farmácias. Para tal, iteramos a colecção `drugs` e guardamos os resultados intermédios na variável `result`:

```
foreach (d in drugs) with result = {} do
  from (x in Stocks) where x.id_drug = d select x.quantity
```

Assumamos agora o identificador `dq` para o resultado de cada projecção na expressão acima. Resta-nos calcular a quantidade média para cada colecção `dq`, que é obtida pela divisão da soma das quantidades pela contagem dos elementos da coleccão:

```
let sum = foreach (x in dq) where y = 0 do y + dq in
let count = foreach (x in dq) where y = 0 do y + 1 in
[drug = d.id_drug, quantity = (sum/count)]:: result
```

De notar que acumulamos o resultado numa colecção de registos, ao invés de uma colecção de inteiros que representam as médias. Deste modo guardarmos também informação acerca dos identificadores dos medicamentos, informação essa necessária mais tarde para efectuar as encomendas. Finalmente, precisamos de colocar as encomendas

```
let drugs = from (x in Drugs) where true select x.id_drug in
  let orders =
      foreach (d in drugs) with result = {} do
          let dq = from (x in Stocks)
                   where x.id_drug = d
                   select x.quantity in
            let sum = foreach (x in dq) where y = 0 do y + dq in
              let count = foreach (x in dq) where y = 0 do y + 1 in
                [drug = d.id_drug, quantity = sum/count] :: result
  in foreach (x in orders) with y = _ do
       insert [id_pharm = 99, id_drug = x.drug, quantity =
           x.quantity]
       in Orders
```

Figura 1: Código completo do exemplo

na relação `Orders`. Para esse efeito, iteramos a colecção de registos resultante do código anterior, que assumimos estar associada ao identificador `orders`:

```
foreach (x in orders) with y = _ do
  insert [id_pharm = 99, id_drug = x.drug, quantity = x.quantity]
  in Orders
```

O código completo do exemplo que acabámos de descrever é apresentado na Fig. 1. Este exemplo ilustra, usando a nossa linguagem, uma aplicação centrada em dados típica que guarda e usa informação numa base de dados, a qual nós representamos através de um modelo relacional de dados. Contudo, o nosso objectivo não consiste em modelar este tipo de aplicações mas em permitir raciocinar sobre estes modelos no que diz respeito à informação que é usada. Nomeadamente, estamos interessados em assegurar, estaticamente, propriedades de segurança relacionadas com os dados tais como a confidencialidade. Para tal, aplicamos à nossa linguagem λ_{DB} uma análise de fluxo de dados baseada num sistema de tipos que impede a divulgação de informação classificada como privada. O princípio por detrás do nosso sistema de tipos consiste na classificação de expressões da nossa linguagem em tipos com anotações de níveis de segurança, algo *standard* em análises de fluxos de informação via sistema de tipos [4]. Esses níveis de segurança estabelecem um reticulado e respeitam uma relação de ordem parcial $\leq$. Para simplificar a apresentação dos conceitos, iremos assumir que o nosso reticulado de níveis de segurança só contém dois níveis de segurança, $\top$ e $\bot$, onde $\bot \leq \top$. Pelo mesmo motivo, omitimos os tipos nos exemplos, escrevendo somente as suas respectivas anotações de níveis de segurança.

Tomando, de novo, como exemplo a aplicação anterior, podemos classificar a morada de um farmacêutico e o seu salário como privados declarando que estes campos têm nível de segurança $\top$ ($\bot$ representa informação pública):

```
entity Pharmacists(id_p:⊥, name:⊥, age:⊥, address:⊤) in
entity Employees(id_p:⊥, id_pharm:⊥, salary:⊤, rank:⊥)
```

Assim, neste contexto em particular, o que pretendemos é garantir que, aquando da execução do programa, a morada e o salário de um farmacêutico não é visível num contexto de nível de segurança $\bot$, isto é, a informação não é disponibilizada num canal público. Uma maneira possível de tentar divulgar informação privada seria guardar essa informação num objecto público. No nosso exemplo, entidades e registos (e seus respectivos campos) são objectos. Portanto, caso declarássemos a seguinte entidade

```
entity Leaks(id_p:⊥, address:⊥, salary:⊥)
```

então poderiamos tentar projectar a informação classificada e inseri-la em `Leaks`:

```
let p_address = from (x in Pharmacists)
                where x.id_p = 42 select x.address in
let p_salary = from (x in Employess)
               where x.id_p = 42 select x.salary in
insert [id_p = 42, address = p_address, salary = p_salary]
in Leaks
```

Contudo, esta última operação é considerada insegura logo o programa é rejeitado pelo sistema de tipos. Isto deve-se ao facto do registo `[id_p = 42, address = p_address, salary = p_salary]` ser tipificado com `[id_p: ⊥, address: ⊤ , salary:⊤]`, dado que `p_address` e `p_salary` são classificados com o nível de segurança $\top$. Assim, como estamos a tentar guardar informação num objecto com um nível de segurança inferior (porque $\bot \leq \top$) através da operação **insert**, o nosso sistema de tipos rejeita o programa.

Este artigo pretende discutir e lançar as ideias base de um sistema de tipos que garanta a não-interferência, na perspectiva da análise de fluxo de informação, em aplicações centradas em dados. As contribuições deste trabalho são duas: o estudo dos fluxos de informação inseguros que podem ocorrer em aplicações centradas em dados, nomeadamente nas primitivas DML; e a definição da propriedade de não-interferência no contexto das aplicações centradas em dados.

O resto do artigo está estruturado da seguinte forma: a secção 2 apresenta a sintaxe da nossa linguagem λ_{DB} e, informalmente, a sua semântica; na secção 3 discutimos os problemas de segurança, do ponto de vista da confidencialidade dos dados, que surgem numa perspectiva de análise de fluxo de informação e apresentamos o teorema da não-interferência; a secção 4 discute algum do trabalho relacionado; e na secção 5 apresentamos as observações finais.

2 Linguagem λ_{DB}

A sintaxe da linguagem λ_{DB} é dada pela gramática na Fig. 2 onde assumimos um conjunto infinito de *variáveis* (que pode tomar valores em $x,\ y, \ldots$), um conjunto infinito de *nomes* (que pode tomar valores em $n,\ m, \ldots$), e um conjunto infinito de *localizações* (que pode tomar valores em $t,\ s, \ldots$). Denotamos $[m_1 : e_1, \ldots, m_n : e_n]$ por $[\overline{m : e}]$, $\{e_1, \ldots, e_n\}$ por $\overline{e}$, e **entity** $t(m_1 : \tau_1^{\ell_1}, \ldots, m_n : \tau_n^{\ell_n})$ **in** e por **entity** $t(\overline{m : \tau^{\ell}})$ **in** e. As expressões na nossa linguagem podem ser variáveis, valores, acesso a um campo de um registo $e.m$, aplicação $e_1(e_2)$, declaração de constantes, adição de um elemento a

$$
\begin{array}{lr}
e ::= & \textit{(expressions)} \\
\quad | \; e.m & \text{(field access)} \\
\quad | \; \textbf{let } x = e_1 \textbf{ in } e_2 & \text{(let)} \\
\quad | \; e_1(e_2) & \text{(application)} \\
\quad | \; \textbf{if } c \textbf{ then } e_1 \textbf{ else } e_2 & \text{(conditional)} \\
\quad | \; [\overline{m : e}] & \text{(record)} \\
\quad | \; \overline{e} & \text{(collection)} \\
\quad | \; e_1 :: e_2 & \text{(cons)} \\
\quad | \; \textbf{foreach}(e_1, e_2, x.y.e_3) & \text{(iteration)} \\
\quad | \; \textbf{entity } t\ (\overline{m : \tau^{\ell}}) \textbf{ in } e & \text{(entity)} \\
\quad | \; \textbf{select}(t, x.c, x.e) & \text{(select)} \\
\quad | \; \textbf{insert}(t, e) & \text{(insert)} \\
\quad | \; \textbf{update}(t, e, x.c) & \text{(update)} \\
\quad | \; \textbf{delete}(t, x.c) & \text{(delete)} \\
\quad | \; x & \text{(variable)} \\
\quad | \; v & \text{(value)}
\end{array}
$$

$$
\begin{array}{lr}
c ::= & \textit{(conditions)} \\
\quad | \; \neg c & \text{(negation)} \\
\quad | \; c_1 \vee c_2 & \text{(disjuction)} \\
\quad | \; V = V & \text{(equality)} \\
\quad | \; V & \text{(term)}
\end{array}
$$

$$
\begin{array}{lr}
V ::= & \textit{(terms)} \\
\quad \overline{V} & \text{(collection)} \\
\quad | \; [\overline{m : V}] & \text{(record)} \\
\quad | \; \lambda^{\ell_2}(x : \tau_1^{\ell_1}).e & \text{(abstraction)} \\
\quad | \; \textbf{true} & \text{(true)} \\
\quad | \; \textbf{false} & \text{(false)} \\
\quad | \; x & \text{(variable)} \\
\quad | \; () & \text{(unit)} \\
\quad | \; V.m & \text{(field access)}
\end{array}
$$

$$
\begin{array}{lr}
v ::= & \textit{(values)} \\
\quad \overline{v} & \text{(collection)} \\
\quad | \; [\overline{m : v}] & \text{(record)} \\
\quad | \; \lambda^{\ell_2}(x : \tau_1^{\ell_1}).e & \text{(abstraction)} \\
\quad | \; \textbf{true} & \text{(true)} \\
\quad | \; \textbf{false} & \text{(false)} \\
\quad | \; () & \text{(unit)}
\end{array}
$$

Figura 2: Sintaxe

uma colecção, iteração de colecção, condicional e primitivas para criação e manipulação de dados. Os valores da linguagem incluem boleanos, valor unitário $()$, colecção de valores $\{v_1, \ldots, v_n\}$, registos $[m_1 : v_1, \ldots, m_n : v_n]$, e abstracções $\lambda^{\ell_2}(x : \tau_1^{\ell_1}).e$. No caso de uma abstracção, declaramos o nível de segurança esperado no seu parâmetro e em que nível de segurança pode esta ser aplicada.

Podemos efectuar computações com elementos de uma colecção através da primitiva de iteração de colecções. Por exemplo, caso quiséssemos somar todos os elementos de uma colecção, então obteríamos tal resultado com a expressão **foreach** $(\{v_1, \ldots, v_n\}, 0, x.y.(x + y))$, onde v_1 até v_n são inteiros. Portanto a primeira sub-expressão numa iteração de colecções é a colecção a iterar, a terceira sub-expressão corresponde à computação que queremos fazer a cada passo da iteração, e a segunda sub-expressão representa o acumulador de resultados intermédios. As variáveis x e y representam um elemento da colecção e o valor actual do acumulador, respectivamente, e são ambas ocorrências ligantes na terceira sub-expressão.

As primitivas de manipulação de dados são as esperadas: criação de uma relação (entidade) localizada em t com o conjunto de atributos m_1 até m_n e seus respectivos tipos, **entity** $t\,(m_1 : \tau_1^{\ell_1}, \ldots, m_n : \tau_n^{\ell_n})$ **in** e; projecção de um conjunto de atributos expressos em e numa relação localizada em t segundo uma dada condição c, **select** $(t, x.c, x.e)$; inserção de um tuplo representado pela expressão e numa relação localizada em t, **insert** (t, e); actualização de um conjunto de elementos numa relação

localizada em t por um tuplo expresso em e, caso uma dada condição c seja satisfeita, **update** $(t, e, x.c)$; e a eliminação de tuplos numa relação localizada em t caso uma dada condição c seja satisfeita, **delete**(t,$x.c$).

De modo a tornar a apresentação mais clara, usamos nos nossos exemplos a sintaxe concreta destas primitivas DML (como fizemos na secção 1). Mais ainda, em qualquer uma destas primitivas, a variável x representa um elemento da entidade localizada em t. Assim, $x.c$ e $x.e$ significa que x é uma ocorrência ligante na expressão condicional c e na expressão e, respectivamente. Caso ocorram na mesma primitiva, então x representa o mesmo elemento da entidade em ambas as expressões c e e. As entidades são representadas como colecções de registos, logo um tuplo é um registo cujos campos correspondem aos atributos da entidade.

As condições lógicas usadas nas primitivas DML e nas condicionais são representadas pelo fragmento c na gramática apresentada. Como condições lógicas temos a disjunção $c_1 \vee c_2$, a negação $\neg c$, e a igualdade de termos $V = V$. Fazemos distinção entre valores lógicos V e valores da linguagem v, os primeiros são usados nas condições lógicas como termos e denotam os valores da nossa linguagem e, adicionalmente, o acesso a um campo de um registo assim como variáveis (ambos necessários para as condições lógicas nas primitivas DML). A semântica destas expressões condicionais é dada através de uma função de interpretação $\mathcal{C}$ que as avalia para um valor boleano.

Assim, o resultado de uma operação de projecção, **select** $(t, x.c, x.e)$, consiste numa colecção de expressões. Estas expressões correspondem à substituição das ocorrências livres de x em e por cada elemento do subconjunto dos registos guardados em t obtido através da aplicação do critério de filtragem representado por c. Ou seja, $\overline{e\{^x/_s\}}$, onde $\overline{s} = \{r_i \in t \mid \mathcal{C}[\![c\{^x/_{r_i}\}]\!] = \texttt{true}\}$. No caso da operação de inserção, **insert** (t, e) , a expressão avalia para um valor unitário $()$ e modifica o estado do programa. Essa alteração consiste em adicionar o registo obtido da avaliação da expressão e à colecção de registos localizada em t. A operação de actualização, **update** `(t,x.c,e)` , resulta igualmente num valor unitário e altera o estado. Contudo, a expressão e representa um registo que pode ser um subconjunto de um tuplo da entidade t. Por exemplo, para actualizar a entidade `Pharmacists` podemos usar um registo com um subconjunto dos campos declarados: `[name:"John", age:64]`. Assim, o estado do programa é alterado para uma nova colecção de registos em t cujos elementos são dados por $\{r_i \in t \mid (\mathcal{C}[\![c\{^x/_{r_i}\}]\!] = \texttt{true})?\ r_i \bullet v : r_i\}$, onde v é a avaliação da expressão e e $\bullet$ é a operação de actualização de registos. Por exemplo, `[id_p = 0, name:"Doe", age:63, address = priv_addr]` $\bullet$ `[age:64]` = `[id_p = 0, name:"Doe", age:64, address = priv_addr]`. Por último, uma operação de remoção, **delete** `(t,x.c)` , resulta num valor unitário e na remoção de todos os tuplos da entidade t que satisfaçam a condição c. Isto é, após a execução desta primitiva a colecção de registo localizada em t é dada por $\{r_i \in t \mid (\mathcal{C}[\![c\{^x/_{r_i}\}]\!] = \texttt{false})\}$.

3 Análise de Fluxo de Informação no λ_{DB}

Nesta secção discutimos os fluxos de informação inseguros, no que diz respeito à confidencialidade dos dados, que podem ocorrer no λ_{DB} sem a nossa análise. Tradicionalmente, na literatura [5,4], estes fluxos inseguros estão divididos em dois conjuntos:

explícitos e implícitos. Além disso, a confidencialidade dos dados é assegurada através da imposição de que os programas preservam a propriedade de não-interferência: os dados observáveis (que estão ao mesmo nível de segurança do contexto de execução) não podem depender de dados classificados com níveis de segurança superior. Na prática, isto quer dizer que nenhuma informação pode fluir de um nível superior para um nível inferior de segurança. Nesse sentido, um fluxo explícito corresponde a um mapeamento directo de uma informação classificada para um objecto classificado com nível inferior (advém do fluxo dos dados do programa), ao passo que um fluxo implícito corresponde a informação pública que depende de informação classificada (advém do fluxo de controlo do programa). Exemplos clássicos de tais fluxos inseguros consistem na afectação de uma variável de nível baixo por um valor de nível alto, `l:=h`, para o caso de fluxo explícito; e de um condicional guardado por uma condição de nível alto cujos ramos estão classificados com nível baixo, **if** `h > 0` **then** `l:= 1` **else** `l:= 0`, para o caso de fluxos implícitos. Contudo, tanto quanto sabemos, não existe nenhum trabalho que discuta que fluxos inseguros podem ocorrer via primitivas DML.

Relembramos que uma entidade consiste numa colecção de registos e que esta é declarada através da primitiva **entity** $t(m_1 : \tau_1^{\ell_1}, \ldots, m_n : \tau_n^{\ell_n})$, em que cada campo m_i é declarado com um tipo τ_i que é anotado com um nível de segurança ℓ_i. Visto que os tipos sem anotações não interessam para a nossa análise, omitimos os tipos na nossa discussão, referindo somente aos níveis de segurança das expressões. Iremos usar as entidades introduzidas na secção 1 juntamente com algumas declarações de constantes:

```
entity Pharmacists(id_p:⊥, name:⊥, age:⊥, address:⊤) in
entity Leaks(id_p:⊥, address:⊥, salary:⊥) in
let priv_addr = from (x in Pharmacists)
                where x.id_p = 42 select x.address in
let pub_addr = from (x in Leaks)
                where x.id_p = 36 select x.address in
```

onde `priv_addr` tem nível de segurança $\top$ e `pub_addr` tem nível de segurança $\bot$.

3.1 Fluxos Explícitos

A nossa linguagem não tem afectações mas tem estado. Este estado é representado pelo conjunto de entidades que um programa contém e os seus respectivos tuplos. Assim sendo, a linguagem tem expressões que são equivalentes a uma operação de afectação no sentido que essas expressões alteram o estado do programa. Estas expressões consistem em algumas das primitivas DML que modificam entidades: **insert** e **update**.

$$\textbf{insert}\ e\ \textbf{in}\ t$$

Como vimos, a expressão e corresponde a um novo tuplo para a entidade localizada em t e é representado por um registo, onde cada campo tem um nível de segurança. Portanto, a operação insert pode ser vista como sendo uma afectação de uma nova colecção de registos na localização t (contendo o novo registo). Isto é, se a entidade localizada em t é representado por uma colecção $\overline{r}$, então uma inserção de um tuplo corresponde à expressão $t := e::\overline{r}$. Assim, temos de assegurar que esta afectação é segura e que não irá divulgar informação sensível (isto é, privada). Consideremos, por exemplo, a entidade `Leaks`. Se inserirmos o seguinte tuplo:

$$\textbf{insert}\ [\texttt{id\_p} : 01, \texttt{address} : \texttt{priv\_addr}, \texttt{salary} : 0]\ \textbf{in}\ \texttt{Leaks}$$

então teríamos um fluxo explícito visto que estaríamos a guardar `priv_addr` (com nível de segurança $\top$) em `address` (um objecto de nível de segurança inferior, $\bot$), quebrando assim a propriedade de não-interferência. Contudo, o inverso não é verdade. Vejamos a seguinte inserção

$$\textbf{insert}\ [\texttt{id\_p} : 02, \texttt{name} : \text{"}X\text{"}, \texttt{age} : 47, \texttt{address} : \texttt{pub\_addr}]\ \textbf{in}\ \texttt{Pharmacists}$$

Neste caso estamos a inserir um tuplo onde atribuímos o valor `pub_addr` (com nível de segurança $\bot$) ao campo `address` (um objecto de nível de segurança superior $\top$). Isto não viola a não-interferência pois estamos a aumentar o nível de segurança do valor ao guardar em memória de nível de segurança superior e, portanto, não estamos a quebrar a confidencialidade dos dados (embora possa violar a integridade destes).

$$\textbf{update}\ (x\ \textbf{in}\ t)\ \textbf{with}\ e\ \textbf{where}\ c$$

Aqui a expressão e representa um subconjunto de uma tuplo da entidade t, logo corresponde a um registo onde cada campo tem um nível de segurança. De notar que este caso é muito parecido com o da primitiva **insert**: temos de garantir que a confidencialidade dos dados é preservada aquando das actualizações dos campos. Por exemplo:

$$\textbf{update}\ (x\ \textbf{in}\ \texttt{Leaks})\ \textbf{with}\ [\texttt{address} : \texttt{priv\_addr}]\ \textbf{where}\ \texttt{true}$$

actualiza o campo `address`, que tem nível de segurança $\bot$, com `priv_addr`, que tem nível de segurança $\top$. Tal como no caso anterior, o inverso não se aplica. Assim sendo, é seguro ter

$$\textbf{update}\ (x\ \textbf{in}\ \texttt{Pharmacists})\ \textbf{with}\ [\texttt{address} : \texttt{pub\_addr}]\ \textbf{where}\ \texttt{true}$$

visto que estamos a actualizar um campo com nível de segurança superior ao do valor usado para a actualização.

3.2 Fluxos Implícitos

Os fluxos implícitos podem surgir nas primitivas DML sempre que temos uma expressão condicional. Ou seja, o estado observável, resultante de executar estas primitivas, não pode depender de expressões condicionais classificadas com um nível de segurança superior ao nível do contexto de execução. Portanto é necessário ter em conta os possíveis fluxos implícitos das seguintes primitivas DML: **select**, **update** e **delete**.

$$\textbf{from}\ (x\ \textbf{in}\ t)\ \textbf{where}\ c\ \textbf{select}\ e$$

A expressão e corresponde à informação que queremos projectar de um subconjunto de tuplos contidos na entidade t. Este subconjunto é obtido pela filtragem da entidade t segundo a expressão condicional c. No entanto, durante a execução da expressão e, o estado do programa pode ser alterado. Logo temos de assegurar que estas mudanças no estado são seguras no que diz respeito à confidencialidade dos dados. Isto é, temos de garantir que nenhuma informação será divulgada através de fluxos implícitos. Este problema é, na realidade, muito semelhante a um condicional com um ramo. Suponhamos que h tem nível de segurança $\top$ e que a variável l tem nível se segurança $\bot$:

$$\texttt{if}\ h > 0\ \texttt{then}\ l := 1$$

o problema aqui não consiste no facto de nos ser permitido escrever esta expressão, mas sim que o nível de segurança do ramo é de nível inferior ($\bot$) ao nível de segurança da expressão condicional que o guarda (que tem nível de segurança $\top$). Isto é, dado que a expressão do ramo, $l := 1$, usa a variável l então a expressão é classificada com o nível de segurança $\bot$ mas, ao mesmo tempo, depende no valor corrente de h, que tem nível de segurança $\top$. Assim, a afectação viola a propriedade de não-interferência pois se olharmos para o valor da variável l podemos inferir se h é ou não superior a 0. Para prevenir este problema, temos de classificar a expressão do ramo, $l := 1$, como tendo pelo menos o mesmo nível de segurança que a expressão condicional $h > 0$ (neste caso $\top$). O mesmo se aplica à primitiva **select**, temos de assegurar que a expressão e é classificada com, pelo menos, o mesmo nível de segurança que a expressão condicional c. Tomemos como exemplo a entidade `Pharmacists`, fazendo a projecção

$$\textbf{from } (x \textbf{ in } \texttt{Pharmacists}) \textbf{ where } x.\mathrm{address} = \texttt{pub\_addr} \textbf{ select } \mathrm{leakUpdate}$$
$$\mathrm{leakUpdate} \triangleq \textbf{update } (y \textbf{ in } \texttt{Pharmacists}) \textbf{ with } [\texttt{age} : 0] \textbf{ where } x.\mathrm{name} = y.\mathrm{name}$$

teríamos um fluxo implícito visto que estaríamos a alterar informação que é pública (`age` para 0) através do uso de dados privados, campo $\mathrm{address}$. Isto é, estaríamos numa situação semelhante ao seguinte condicional:

$$\texttt{if } (x.\mathrm{address} = \texttt{pub\_addr} \text{ and } x.\mathrm{name} = y.\mathrm{name}) \texttt{ then } \mathrm{age} := 0$$

onde a expressão do ramo, $\mathrm{age} := 0$, é classificada com o nível de segurança $\bot$, e a expressão condicional com o nível de segurança $\top$ (pois usa um campo classificado com $\top$, address). Além disso, esta projecção iria permitir que qualquer um pudesse escrever uma *query* para projectar todos os farmacêuticos com idade igual a 0, da qual se poderia inferir todos os farmacêuticos que moram na morada `pub_addr`, violando claramente a confidencialidade dos dados.

$$\textbf{update } (x \textbf{ in } t) \textbf{ with } e \textbf{ where } c$$

Neste caso, a decisão de actualizar um tuplo depende da expressão condicional c. Por outras palavras, o que se tem aqui é um condicional com uma afectação (como vimos no exemplo anterior). Portanto, temos de assegurar que nenhuma informação é divulgada via fluxos implícitos. Voltemos a tomar como exemplo a entidade `Pharmacists`. Se efectuarmos a seguinte actualização

$$\textbf{update } (x \textbf{ in } \texttt{Pharmacists}) \textbf{ with } [\mathrm{age} = 54] \textbf{ where } x.\mathrm{address} = \texttt{pub\_addr}$$

então podemos inferir que quem tem idade igual a 54 vive na morada dada. Para uma melhor compreensão, olhemos para a expressão anterior como um condicional com uma afectação

$$\texttt{if } (x.\mathrm{address} = \texttt{pub\_addr}) \texttt{ then } \mathrm{age} := 54$$

onde age é classificado como $\bot$, o campo address como $\top$, o valor `pub_addr` como $\bot$, e finalmente a expressão $(x.\mathrm{address} = \texttt{pub\_addr})$ é classificada como $\top$. Então torna-se óbvio que estamos na presença de um fluxo implícito pois actualizamos um objecto de nível $\bot$ dada uma condição de nível $\top$, o que viola a propriedade de não-interferência.

$$\textbf{delete } (x \textbf{ in } t) \textbf{ where } c$$

Nesta primitiva, a expressão condicional c indica quais os tuplos que irão ser removidos da entidade t, logo precisamos de ser cautelosos de modo a não apagar informação com nível de segurança inferior ao da expressão condicional. Caso contrário teríamos um fluxo implícito. Tomando como exemplo, mais uma vez, a entidade `Pharmacists`, se realizarmos a seguinte operação de remoção

$$\textbf{delete}\ (x\ \textbf{in}\ \texttt{Pharmacists})\ \textbf{where}\ x.\mathrm{address} = \texttt{priv\_addr}$$

então estaremos a alterar o estado observável do programa baseando num critério que usa dados privados. Isto é, todas as linhas com morada igual a `priv_addr` iriam ser removidas. Porém, essas linhas também contêm informação de nível $\bot$. Suponhamos que a entidade `Pharmacists` tinha três linhas: $(1, \text{"A"}, 35, \texttt{pub\_addr})$, $(2, \text{"B"}, 63,$ `priv_addr`$)$ e $(3, \text{"C"}, 12, \texttt{priv\_addr})$. Então o estado observável (toda a informação classificada com $\bot$) corresponderia aos seguintes tuplos: $(1, \text{"A"}, 35)$, $(2, \text{"B"}, 63)$ e $(3, \text{"C"}, 12)$. Assim, se efectuássemos a operação de remoção apresentada, ficaríamos somente com um tuplo $(1, \text{"A"}, 35)$. Portanto, um fluxo implícito tinha ocorrido pois deixámos de observar os tuplos $(2, \text{"B"}, 63)$ e $(3, \text{"C"}, 12)$, logo a não-interferência tinha sido violada.

3.3 Teorema de Não-Interferência

Antes de apresentar o teorema da não-interferência, vamos explicar sucintamente algumas definições necessárias para a sua compreensão. Comecemos pela relação de redução de um programa na nossa linguagem que tem a forma $(S, e) \longrightarrow (S', e')$ e diz que a expressão e no estado S reduz para a expressão e' passando para o estado S'. Um estado S consiste num mapeamento entre localizações e entidades (ou seja, colecções de registos). Dizemos que dois estados são iguais até ao nível de segurança ℓ, denotando como $S_1 =_\ell S_2$, se estes concordarem em todos os valores de nível inferior ou igual a ℓ. Definimos uma relação de equivalência entre expressões da nossa linguagem, designada por $\Delta \vdash e_1 \cong_\ell e_2 : \tau^{\ell'}$, que indica que duas expressões são equivalentes ao nível ℓ se forem iguais sintacticamente excepto nos valores, onde exigimos que estes sejam iguais caso tenham nível de segurança inferior ou igual a ℓ. Esta relação de equivalência exprime o que conseguimos observar com um contexto de nível de segurança ℓ. Por último, a relação de tipificação de uma expressão é representada da seguinte forma $\Delta \vdash e : \tau^\ell$, onde Δ é o ambiente de tipificação (contém atribuições entre identificadores e tipos assim como entre localizações e tipos) e τ^ℓ o tipo da expressão e anotado pelo nível de segurança ℓ da expressão. Não tendo este artigo como objectivo a descrição detalhada do sistema de tipos, é ainda assim importante realçar dois pormenores sobre a relação de tipificação acima apresentada. Primeiro, as anotações de nível de segurança nos tipos têm uma leitura distinta caso o tipo seja unitário ou não: se for de tipo unitário então esse nível representa o nível mínimo da informação que será alterada pela expressão; caso contrário, o nível indica o nível máximo a que uma expressão lê informação (não dependendo, assim, de informação de nível superior). Segundo, a relação de equivalência entre expressões está intrinsecamente ligada à relação de tipificação: se duas expressões são equivalentes então estão bem tipificadas e concordam no tipo e na anotação de nível de segurança.

Estamos agora em condições de enunciar o teorema da não-interferência.

Theorem 1 (Não-Interferência).

$$\begin{array}{ll} \textit{IF} & \Delta \vdash S_1 \ \wedge \ \Delta \vdash S_2 \ \wedge \ S_1 =_\ell S_2 \\ & (S_1, e_1) \longrightarrow (S_1', e_1') \ \wedge \ (S_2, e_2) \longrightarrow (S_2', e_2') \\ & \Delta \vdash e_1 \cong_\ell e_2 : \tau^{\ell'} \\ \textit{THEN} & \exists \Delta' \quad \Delta \subseteq \Delta' \ \wedge \ \Delta' \vdash S_1' \ \wedge \ \Delta' \vdash S_2' \\ & \qquad S_1' =_\ell S_2' \ \wedge \ \Delta' \vdash e_1' \cong_\ell e_2' : \tau^{\ell'} \end{array}$$

Prova: Por indução na relação $(S; e) \longrightarrow (S'; e')$.

O teorema diz que se duas expressões são equivalentes até ao nível ℓ (e portanto, por definição da relação de equivalência, estão bem tipificadas), e os seus respectivos estados são iguais até ao nível ℓ, então preservam as relações de equivalência entre elas, assim como a igualdade entre os seus estados, após um passo de avaliação. É importante realçar que as expressões representam o mesmo programa só que em estados diferentes (o que, aliás, é *standard* nas propriedades de não-interferência). Porém, devido à semântica da primitiva **select**, pode ser o caso que durante a avaliação do programa as expressões não sejam sintacticamente iguais, daí a nossa relação de equivalência entre expressões.

4 Trabalho Relacionado

Tradicionalmente, as análises de fluxo de informação via sistemas de tipos, classificam frases de um programa com tipos anotados por etiquetas de segurança (níveis de segurança), assegurando, por composição de regras de tipificação, que o fluxo de informação é seguro. Existem muitos trabalhos, ao longo dos últimos anos, que estudam e abordam este tipo de análises mas que focam em aspectos distintos das aplicações. Citamos alguns dos trabalhos mais influentes na área: em [4,6], os autores focam-se numa linguagem imperativa com condicionais e ciclos mas sem registos ou colecções de valores; o trabalho apresentado em [7] diz respeito a um cálculo-lambda com tipos soma e produto; os autores de [8] propõem uma análise de fluxos de informação para uma linguagem orientada a objectos. Em qualquer um destes trabalhos é apresentado um resultado de não-interferência que garante que os programas bem tipificados não violam a confidencialidade dos dados.

Nos últimos anos têm surgido trabalhos que se focam em segurança da informação em aplicações centradas em dados. Em [9] é apresentado uma linguagem funcional, Dminor, com um pequeno conjunto de primitivas para manipulação de dados tais como uma expressão de acumulação sobre colecções e adição de elementos a uma colecção. O objectivo dos autores é estudar uma linguagem cujas expressões compilam para *queries* SQL e cujos tipos correspondem a esquemas relacionais, nomeadamente os autores fazem uso de tipos refinados para exprimir restrições sobre tabelas SQL no sistema de tipos que propõem. Mais recentemente, foi proposto em [2] um cálculo-lambda com primitivas de manipulação de dados ao estilo do SQL. Os autores fazem também uso de tipos refinados mas, neste caso, para especificar políticas de controlo de acesso que dependem do estado actual da base de dados do programa. Contudo esses trabalhos centram-se na questão do acesso seguro dos dados ao passo que o nosso trabalho foca-se na segurança da informação, durante a execução de um programa, depois de esta ser acedida de forma segura.

Tanto quanto sabemos, não existe nenhum trabalho que trate do problema da análise de fluxo de informação, via sistema de tipos, no contexto de aplicações centradas em dados nem tão pouco existe algum estudo sobre que tipo de fluxos inseguros podem ocorrer neste contexto.

5 Observações Finais

Neste artigo apresentámos uma linguagem funcional com primitivas DML, o λ_{DB}, baseada nos trabalhos [2] e [1], assim como a intuição e as ideias por detrás de um sistema de tipos que permite efectuar uma análise de fluxo de informação. O objectivo consiste em demonstrar que um programa bem tipificado no λ_{DB} não viola a confidencialidade dos dados, ou seja, que o programa não revela informação classificada como privada. Assim sendo, introduzimos o tema de forma informal através de um exemplo ilustrativo de uma aplicação centrada em dados e, com o auxilio desse exemplo, mostrámos que fluxos de informação inseguros podem ocorrer, dando a intuição por detrás das regras do nosso sistema de tipos. Finalmente, propusemos uma propriedade de não-interferência adequada ao contexto das aplicações centradas em dados que garante a confidencialidade dos dados em programas no λ_{DB}.

Com este artigo pretende-se contribuir para a discussão sobre os fluxos de informação inseguros que podem ocorrer nas aplicações centradas em dados, assim como definir a propriedade de não-interferência apropriada a essas aplicações.

Agradecimentos. Agradecemos aos revisores anónimos os comentários e sugestões dados a este artigo. Este trabalho é apoiado por CITI, Projecto INTERFACES, e FCT/MCTES pela Bolsa de Doutoramento SFRH/BD/68801/ 2010.

Referências

1. Lourenço, L.: A core language for data-centric processes. In: Simpósio de Informática, INForum 2011, Coimbra, Portugal. (2011)
2. Caires, L., Perez, J.A., Seco, J.C., Vieira, H.T., Ferrão, L.: Type-based access control in data-centric systems. In Barthe, G., ed.: ESOP'11. Number 6602 in LNCS, Springer-Verlag (2011)
3. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. (1982)
4. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2–3) (1996)
5. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Communications of the ACM **20**(7) (1977)
6. Sabelfeld, A., Sands, D.: A per model of secure information flow in sequential programs. Higher-Order and Symbolic Computation (2001)
7. Heintze, N., Riecke, J.G.: The slam calculus: Programming with secrecy and integrity. In: POPL'98. (1998)
8. Myers, A.C.: Jflow: Practical mostly-static information flow control. In: POPL'99. (1999)
9. Bierman, G.M., Gordon, A.D., Hritcu, C., Langworthy, D.E.: Semantic subtyping with an SMT solver. In Hudak, P., Weirich, S., eds.: ICFP'10, ACM (2010) 105–116