Segurança de fluxos de informação em redes: uma aplicação da teoria de tipos

Ana Almeida Matos

Resumo

Este trabalho incide sobre problemas da segurança de fluxos de informação em redes e é dedicado aos temas da confidencialidade e declassificação[1] num contexto distribuído com mobilidade de código. Dá-se especial atenção a dois problemas principais ainda praticamente inexplorados:

- Como encontrar um mecanismo flexível de segurança de fluxos de informação, que permita declassificação, i.e. a revelação voluntária (total ou parcial) de informação sensível? Propõe-se a introdução em linguagens de programação de uma *declaração de fluxo*, que implementa uma política local de fluxo de informação.
- Que problemas de segurança de fluxos de informação podem surgir em redes de computação, e como controla-los? Considerando um modelo de rede em que a computação é distribuída por diferentes domínios, podendo envolver migração de código e de recursos, são reveladas novas formas de fuga de informação até aqui ignoradas, introduzidas pela mobilidade de processos.

De modo a poder lidar com políticas de fluxo dinâmicas, declaradas de forma independente por processos distribuídos por uma rede, introduz-se uma generalização da propriedade clássica de não-interferência, aqui denominada *não-divulgação em redes*. É apresentada uma metodologia geral para definir e provar a correcção de sistemas de tipos e efeitos que asseguram (estaticamente) políticas de segurança flexíveis para linguagens imperativas de ordem superior com distribuição e mobilidade de código.

Palavras chave: Fluxos de informação, sistemas de tipos e efeitos, declassificação, distribuição, mobilidade, não-interferência.

Área na classificação ACM: C.2, D.3, F.3

1 Introdução

Desde os primeiros computadores, o problema da segurança informática tem ganho importância crescente. Um dos seus principais objectivos é a *confidencialidade*, isto é, a garantia de que apenas entidades autorizadas podem aceder a certa informação.

Os primeiros esforços neste sentido, que datam dos princípios da informática, tinham como objectivo criar partições de memória, assegurando que os programas em execução não acedessem a partições reservadas a outros programas. Isto constitui um dos primeiros exemplos de *controlo de acessos*, um mecanismo de protecção de confidencialidade que permite ao sistema autorizar ou proibir o acesso a certos dados, bem como a execução de determinadas acções. No entanto, uma vez dada a autorização, o controlo de acessos não regula a propagação da informação que é desvendada durante a execução de um programa [11, 18]. Esta observação suscitou uma atenção crescente sobre o *controlo de fluxos de*

[1] Neste trabalho utilizaremos "declassificação" para traduzir a palavra inglesa *declassification*, em vez de "desclassificação", para evitar a interpretação de "desqualificação" que lhe está normalmente associada.

informação. O seu objectivo é, precisamente, saber e controlar como a informação circula num sistema informático, com o objectivo de interditar o seu acesso a entidades não autorizadas.

Em apenas algumas décadas, os sistemas informáticos evoluíram desde as máquinas partilhadas às redes globais de computadores, onde os programas se deslocam de maneira descentralizada, e a topologia das redes se altera incessantemente. Assistimos mesmo à emergência de um novo paradigma de computação, a *computação "grid"*, que envolve a divisão de programas em sub-tarefas a ser executadas paralelamente em diferentes plataformas. Enormes quantidades de recursos, fisicamente dispersos pelo globo, são mobilizados para participar na computação de programas. Nesse ambiente de computação global (e móvel), os problemas de segurança tornaram-se cruciais. De facto, os novos potenciais oferecido pela globalização têm sido frequentemente exploradas com objectivos duvidosos, sendo os vírus, *worms*, recusa de serviço, e outras formas te ataque já popularizados, apenas a ponta do icebergue. Estranhamente, pouco se tem estudado sobre os problemas que afectam os fluxos de informação em redes. É sobre isso que se debruça este trabalho.

1.1 Sistemas de tipos para a segurança de fluxos de informação

Neste trabalho adoptamos uma abordagem *orientada a linguagens* [30], restringindo a nossa atenção aos fluxos de informação que resultam da execução de programas. Por outras palavras, preocupamo-nos com as fugas de informação associadas às transferência de informação entre objectos computacionais de uma dada linguagem, recorrendo a técnicas clássicas de análise de linguagens de programação.

Afim de especificar quais as trocas de informação que são seguras, é natural atribuir *níveis de segurança* aos objectos de uma linguagem (variáveis, canais...), que apenas poderão ser lidos por entidades (sujeitos) com a autorização correspondente. Dada uma relação de ordem entre esses níveis de segurança [11], um programa pode transferir informação de um objecto para outro se o primeiro tiver um nível de segurança inferior ao segundo. Este conceito foi formalizado pela noção de *dependência forte* por [8], e é agora conhecido por *não-interferência*, na terminologia de Goguen e Meseguer [13].

Um trabalho considerável tem sido dedicado à concepção de métodos de análise dos fluxos de informação estabelecido por um programa [2]. A análise pode ser efectuada dinamicamente, utilizando verificações em tempo de execução. Estes métodos podem ser criticados pelos elevados custos de computação e memória, e por poderem revelar informação pela simples falha na verificação durante a execução [11, 25]. Foram também desenvolvidos métodos de análise estática de fluxos de informação, permitindo a rejeição de programas inseguros antes da sua execução. De salientar o uso de sistemas de tipos, iniciado pelo trabalho de Volpano, Smith e Irvine [40]. Apesar de oferecerem apenas uma análise aproximativa, os sistemas de tipos decidíveis têm vantagens reconhecidas, tais como a prevenção de erros de programação. Foram desenvolvidos sistemas de tipos para segurança de fluxos de informação para inúmeras linguagens [38, 36, 14, 40, 35, 4, 42, 10, 27], incluindo linguagens realistas como Jif (ou JFlow) [26] e Flow Caml [34].

1.2 Problemas abordados

Tal como foi observado em [41], "apesar da sua longa história e qualidades atraentes, os mecanismos de controlo de fluxo de informação não foram ainda utilizados com sucesso na prática". Segue-se uma discussão deste problema e das contribuições deste trabalho para a sua resolução. Esta discussão é organizada ao longo de três linhas principais: mais refinamento, maior flexibilidade e melhor integração.

Refinamento Uma grande parte dos esforços consagrados ao controlo de fluxos de informação consiste em determinar que fluxos de informação são indesejados. Mesmo adoptando a política mais restritiva de rejeitar todas as possíveis fugas de informação, resta ainda um longo caminho a percorrer até se perceber quais são os tipos de troca de informação que podem ser explorados maliciosamente. Este problema está fortemente ligado ao nível de expressividade do contexto computacional em que o programa é executado. Tipicamente, a introdução de novas funcionalidades numa linguagem acarreta o aparecimento de novas formas de fugas de informação. É por isso real a necessidade de mecanismos de análise de segurança que sejam aplicáveis a linguagens pelo menos tão expressivas quanto aquelas que são utilizadas na prática. Basearemos este trabalho sobre o estudo de uma versão distribuída do Core ML [23], um λ-cálculo com chamada por valor e comandos imperativos, ao qual adicionamos processos concorrentes.

É fácil encontrar mecanismos para seleccionar apenas programas seguros: um exemplo extremo seria não seleccionar nenhum. Já a validação do maior número possível de programas seguros acrescenta outra dimensão ao problema. De facto, determinar se um programa é seguro é frequentemente indecidível, o que implica que os procedimentos de rejeição de programas inseguros são necessariamente excessivos. Durante o desenvolvimento de sistemas de tipos para controlo de fluxo de informação, a chave parece ser a identificação dos efeitos e dos níveis de segurança de informação de que dependem esses efeitos. Formalizando a noção de *efeito* de maneira cada vez mais precisa, torna-se possível exprimir condições cada vez mais rigorosas para aceitar programas. Isto é demonstrado neste trabalho, considerando um sistema de tipos e efeitos [20] que inclui efeitos de leitura, escrita e terminação de programas.

Flexibilidade É interessante notar que, mesmo em sistemas em que a segurança tem uma importância crucial, normalmente a *não-interferência* não é a política desejada. De facto, a rejeição cega de qualquer possibilidade de fugas de informação impediria a utilização de programas muito comuns e mesmo indispensáveis. Programas de verificação de palavras chave ou de encriptação são exemplos típicos, em que o próprio princípio implica a declassificação (mesmo que residual) de uma informação secreta para observadores públicos. A não-interferência é portanto proibitivamente restritiva para ser utilizada na prática. Este facto tem motivado recentemente muita investigação em propriedades de segurança alternativas, mais flexíveis do que a não-interferência, e permitindo alguma forma de declassificação (ver [37, 39, 24, 31, 7, 22, 19] e um estudo comparativo em [33]). No entanto, a maior parte destas abordagens são influenciadas pelo receio de que a declassificação, uma vez autorizada, possa ser utilizada de forma a declassificar mais informação do que seria considerado seguro. Como resultado, as propriedades de segurança que existem na literatura incluem frequentemente restrições que diminuem a sua adequação como alternativa à não-interferência.

Neste trabalho, defendemos que, antes de prever restrições a impor sobre a utilização da declassificação, há que providenciar meios flexíveis e simples de a exprimir, e propomos a *não-divulgação* como uma generalização natural da não-interferência. Em particular, seria bom poder exprimir operações que deliberadamente aceitam fluxos de informação que são rejeitados pela ordem de segurança subjacente. Com este fim, propomos um mecanismo para estender localmente a relação de ordem que regula os fluxos permitidos, graças a uma *declaração de fluxo*. Isto permite ao programador configurar a política de segurança em cada situação particular, com a ajuda de simples condições de fluxo impostas sobre níveis de segurança.

Integração Afim de construir aplicações reais para a segurança de fluxos de informação, é necessário integrá-la com os mecanismos de segurança existentes. Tal como já mencionámos, a articulação entre o controlo de acessos e o controlo de fluxos é particularmente importante. O controlo de acessos é tipicamente assegurado pelos sistemas operativos com base em *listas de controlo de acessos* (listas de entidades autorizadas). Os sistemas de controlo de fluxo podem ser especificados em termos de etiquetas de segurança [25]. Aqui vamos mais longe, e exprimimos as próprias políticas de segurança em termos de relações entre entidades, a partir das quais se pode obter as usuais relações de ordem entre níveis de segurança. Sugerimos assim que o controlo de acessos e o controlo de fluxos de informação podem ser facilmente combinadas.

A um nível mais elevado, proteger a confidencialidade de informação é um problema particularmente relevante num contexto de computação global. Quando informação e programas se movimentam por redes, são expostos a utilizadores com interesses e responsabilidades diferenciados. Isto motiva a procura de mecanismos práticos para impor o respeito pela confidencialidade de informação, minimizando assim a necessidade de se depender da confiança mútua. Neste trabalho apresentamos um primeiro estudo dos fluxos de informação que são introduzidos pela mobilidade no contexto de uma linguagem distribuída com estado. A pertinência da ideia de que a computação global traz novos desafios no domínio da análise de fluxo de informação será confirmada pela identificação de uma nova forma de fugas de informação, as *fugas por migração*, que podem aparecer em ambientes distribuídos com mobilidade.

1.3 Contribuições e conteúdo

Consideramos como contribuições principais deste trabalho:

- A introdução de um comando de declaração de fluxo que permite declassificação.
- A apresentação de uma propriedade de segurança que é uma generalização directa da não-interferência: a propriedade de não-divulgação.
- A identificação de um novo tipo de fugas de informação que aparece em ambientes em que a mobilidade de recursos joga um papel explícito.
- A generalização da propriedade de não-divulgação a contextos de computação global.
- O estudo e desenvolvimento de sistemas de tipos e efeitos que garantem segurança de fluxos de informação para linguagens baseadas num cálculo lambda imperativo de ordem superior com criação de processos e referências.
- A integração da propriedade de não-divulgação, na sua versão local e global, nos referidos sistemas de tipos e efeitos.

Na secção que se segue formaliza-se o modelo de computação em que nos baseamos, definindo a linguagem de programação, na forma de um cálculo. Segue-se uma secção que expõe e discute os conceitos e problemas de segurança pelos quais nos interessámos, e antecipa as soluções propostas. São dados exemplos de fugas de informação com base na linguagem definida anteriormente. A quarta secção é dedicada à formalização da propriedade de segurança que apresentamos, a propriedade de não-divulgação. Por fim, a quinta secção define um mecanismo para garantir a nossa propriedade de segurança, na forma de um sistema de tipos e efeitos, sublinhando a sua aplicabilidade noutros contextos. Este trabalho termina com uma discussão dos resultados e comparações com trabalhos relacionados.

O trabalho apresentado baseia-se na tese de doutoramento da autora. Os resultados principais foram sujeitos a um processo completo de revisão independente por pares

numa conferência e dois workshops internacionais, estando dois artigos em vias de ser publicados numa revista da área. Face às limitação de espaço, e sem prejuízo do rigor necessário, optou-se por dar maior relevância ao fornecimento de motivações e de explicações intuitivas, por via de exemplos clarificadores, no lugar de apresentar as provas, que são extensas e complexas. As provas, bem como uma descrição mais detalhada do desenvolvimento técnico que deu origem aos resultados aqui apresentados, poderão ser consultados integralmente na referida tese.

2 O cálculo

Antes de mergulhar nos conceitos de segurança relacionados com este estudo, começamos por definir o enquadramento formal em que ele se baseia. Começamos por descrever o modelo de redes em que ele se baseia: a configuração de redes e de domínios, os comportamentos de migração, e a distribuição de recursos. Depois, definimos a sintaxe e semântica do cálculo que descreve as computações que ocorrem a nível local e a nível da rede.

Escolha de um cálculo para a computação global Uma rede pode ser vista como um conjunto de domínios de computação em que os recursos físicos são apenas acessíveis aos programas locais, em que falhas podem ser geradas por tentativa de aceder a recursos que não estão presentes. Interessa-nos definir um modelo geral e simples, que reflicta a natureza imprevisível do acesso a recursos em redes, bem como problemas de confiança mútua entre entidades que seguem diferentes orientações em segurança. Notando que a ocorrência de uma falha pode fornecer informação acerca do sistema, pretende-se esclarecer se essas falhas podem ser exploradas maliciosamente, originando fugas de informação significativas. Tendo isso em vista, concebemos uma linguagem em que a localização de um programa e de um recurso tem impacto nas computações.

O desenho de modelos de redes é uma área de investigação *per se*, existindo na literatura um espectro de cálculos que focam em diferentes aspectos da mobilidade [44]. A característica principal do nosso cálculo será a de que os acessos a recursos apenas podem ser efectuados por processos que se situem no mesmo local; acessos remotos são suspensos até que os recursos se tornem disponíveis.

Neste trabalho, uma *rede* consiste num conjunto de *domínios*, locais onde computações podem ocorrer fisicamente. Processos podem executar concorrentemente dentro dos domínios, podendo gerar outros processos, ou *migrar* para outros domínios. Podem criar e ser proprietários de espaço em *memória*, que atribui valores a *referências* (endereços de recipientes de memória). As memórias movem-se juntamente com os processos a que pertencem, o que significa que cada processo e suas referências estão, em qualquer momento, sempre localizados no mesmo domínio. No entanto, um processo pode aceder a referências que não lhe pertencem, desde que estejam localizados no mesmo domínio (caso contrário o processo é implicitamente suspenso).

2.1 Sintaxe

A linguagem escolhida consiste num cálculo lambda de *ordem superior*, estendido com as construções imperativas do ML [23], testes condicionais e booleanos, e a criação dinâmica de referências e processos. Inclui-se ainda um mecanismo de declassificação, uma noção de domínio e uma primitiva de migração básica. De seguida definimos a sintaxe das anotações de segurança, tipos, expressões e redes (configurações). As definições que a caracterizam encontram-se nas Figuras 1, 2 e 3.

Anotações de segurança e de tipo Assume-se dado um conjunto $\boldsymbol{Pri}$ de todas as *entidades* do sistema, denotadas p, q. Um *nível de segurança* l, j, k é então um subconjunto de $\boldsymbol{Pri}$. Aparecem explicitamente na sintaxe (Figura 1) sendo associadas a referências (e criadores de referências). O nível de segurança de uma referência representa o conjunto das entidades que estão autorizadas a ler informação contida nessa referência. Uma política de fluxo F é um conjunto de pares de entidades, onde um par $(p, q) \in F$, normalmente escrito $p \prec q$, significa que "informação pode fluir da entidade p para a entidade q", ou mais precisamente, "*tudo aquilo que a entidade p pode ler também pode ser lido por q*". Estes conceitos são explicados cuidadosamente na secção seguinte.

Tipos e efeitos também são aparentes na sintaxe da linguagem (Figura 1) e serão explicados na Secção 5. Estas anotações não jogam nenhum papel na semântica operacional, mas são usadas nas provas de correcção.

Expressões A sintaxe das expressões está definida na Figura 2. Assumimos a existência de quatro conjuntos disjuntos numeráveis $\boldsymbol{Dom} \neq \emptyset$, $\boldsymbol{Nam}$, $\boldsymbol{Var}$, e $\boldsymbol{Ref}$. *Nomes* são dados a domínios ($d \in \boldsymbol{Dom}$), processos ($m, n \in \boldsymbol{Nam}$) e referências ($a, b, c$), que também podem chamar-se *endereços*. Associamos anotações (índices) aos nomes: *nomes de processo decorados* incluem os níveis de segurança de processos, enquanto que os *nomes de referência decorados* incluem o nível de segurança da referência e o tipo dos valores que podem apontar, bem como o nível de segurança do processo a que pertencem. Assim, nomes de processo decorados m_j consistem em pares compostos por um nome de processo m e um nível de segurança j, enquanto que um nome de referência decorado $m_j.u_{l,\theta}$ é um 5-tuplo composto por um nome de processo m, o seu nível de segurança j, um identificador de referência u, um tipo θ e um nível de segurança l. As referências são lexicalmente associadas aos processos que os criam: são da forma $m_j.u$, onde u é um identificador dado por um processo. Nomes de processo e de referência podem ser criados durante a execução. No que se segue poderemos omitir índices sempre que não sejam relevantes, seguindo a convenção de que o mesmo nome tem sempre o mesmo índice.

Valores, abrangidos por $V \in \boldsymbol{Val}$, são casos especiais de expressões que não computam, e incluem: o comando $()$ que não faz nada; a abstracção de função $(\lambda x.M)$ com corpo M e parâmetro x; os valores booleanos tt e ff. A construção $(\varrho x.W)$, que liga as ocorrências de x no pseudo-valor W, é usado para exprimir valores recursivos – executa recursivamente o resultado de aplicar $(\lambda x.W)$ a si próprio.

O conjunto $\boldsymbol{Exp}$ das *expressões*, abrangido por M, N, inclui: a aplicação $(M\ N)$ que aplica a função que resulta de executar M ao resultado de N; a condição de teste (if M then N_t else N_f) que executa N_t ou N_f conforme M retornou tt ou ff[2]; a composição sequencial $(M; N)$ que executa N depois de M ter terminado; a operação de leitura $(?\ M)$ que, depois de M ter executado e retornado um nome de referência decorado, retorna o valor para o qual aponta a referência; a atribuição $(M :=^{?} N)$ do valor retornado pela execução de N ao nome de referência decorado que é retornado por M (a notação '?' assinala o facto de que estas operações podem potencialmente ser suspensas, quando a referência que está a ser, respectivamente, lida ou escrita não for acessível); o criador de processos (thread M) gera um novo processo M que será executado concorrentemente, e retorna $()$; a instrução de migração (goto d), em que d é o nome de um domínio, implica a migração do processo que executa esta operação para o domínio d; por fim, a declaração de fluxo (flow F in M), em que F é uma política de fluxo, tem o mesmo comportamento de M – veremos na próxima secção que esta declaração estabelece que os

[2] A notação N_t e N_f é usada apenas por conveniência notacional para distinguir os ramos, e não tem qualquer outro significado.

fluxos declarados em F podem ocorrer durante a execução de M.

Outros comandos úteis podem ser derivados a partir das expressões aqui definidas. Por exemplo, funções recursivas podem ser escritas como $(\varrho f.(\lambda x.M))$, e denotaremos por loop a expressão $(\varrho x.x)$. Podemos ainda codificar ciclos while (while M do N) como $((\varrho y.(\lambda x.(\text{if } M \text{ then } (N; (y\ x)) \text{ else } x)))\ ())$.

Redes e configurações Definimos *memórias* S, que são um mapeamento dum conjunto finito de nomes de referência decorados a $\boldsymbol{Val}$, e *processos*, que são expressões com nome M^{m_j}. Processos executam concorrentemente[3] em grupos P (conjuntos de processos), que são mapeamentos entre nomes de processo decorados e expressões. Redes são justaposições planas de domínios, cada um contendo uma memória e um grupo de processos. Assume-se que os nomes de processo e de domínio são distintos; além disso, assume-se que as referências estão localizadas no mesmo domínio que os processos a que pertencem (o nome de cada referência é prefixado pelo nome do processo a que pertence) e que têm sempre as mesmas decorações.

De notar que as redes são afinal apenas colecções de processos e referências correspondentes que executam em paralelo, e cujas execuções dependem da sua posição relativa. De modo a saber sempre a localização dos processos e referências, é suficiente guardar um mapeamento entre nomes de processo (decorados) e nomes de domínio. Para isso serve o *mapeador de posições*, que juntamente com o grupo P contendo todos os processos numa rede, e a memória S contendo todas as referências numa rede, formam as *configurações* $\langle P, T, S\rangle$, sobre as quais estão definidas as transições (ver próxima subsecção). Mais precisamente, dado o conjunto $\mathcal{D}$ de domínios numa rede, a partir de uma rede $d_1[\ P_1, S_1\ ] \parallel \cdots \parallel d_n[\ P_n, S_n\ ]$ obtemos uma configuração da forma $\langle P, T, S\rangle$, tal como na Figura 3 onde $P = P_1 \cup \cdots \cup P_n$, $S = S_1 \cup \cdots \cup S_n$ e:

$$T = \{m_j \mapsto d_1 | M^{m_j} \in P_1\} \cup \cdots \cup \{m_j \mapsto d_n | M^{m_j} \in P_n\}$$

2.2 Semântica

Nesta secção definimos a semântica operacional da linguagem (ver Figuras 4 e 5). Começamos por estipular algumas notações e convenções úteis. De seguida descrevemos as transições simples entre configurações, que são baseadas em contextos, e apresentamos uma propriedade importante da semântica.

Conjuntos base e funções Dada uma configuração $\langle P, T, S\rangle$, chamamos ao par (T, S) o *estado* da configuração. Definimos $\mathrm{dom}(T)$, $\mathrm{dom}(P)$ e $\mathrm{dom}(S)$ como sendo os conjuntos de nomes de processo e de referência decorados que são mapeados por T, P e S, respectivamente. Dizemos que um nome de processo ou de referência é novo em T ou S se não ocorre, com nenhum índice, em $\mathrm{dom}(T)$ ou $\mathrm{dom}(S)$, respectivamente. Denotamos por $\mathrm{tn}(P)$ e $\mathrm{rn}(P)$ o conjunto dos nomes de processo e de referência decorados, respectivamente, que ocorrem nas expressões de P (esta notação é estendida de forma óbvia a expressões). Reutilizamos ainda a notação tn definindo, para um conjunto R de nomes de referência, o conjunto $\mathrm{tn}(R)$ dos nomes de processo que são prefixos dos nomes em R.

Restringimos a nossa atenção às *configurações bem formadas* $\langle P, T, S\rangle$ que satisfazem as condições seguintes sobre memórias, valores guardados em memórias e nomes de processo: $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$; $a_{l,\theta} \in \mathrm{dom}(S)$ implica que $\mathrm{rn}(S(a_{l,\theta})) \subseteq \mathrm{dom}(S)$; $\mathrm{dom}(P) \subseteq \mathrm{dom}(T)$; $\mathrm{tn}(\mathrm{dom}(S)) \subseteq \mathrm{dom}(T)$; todos os processos de uma configuração têm nomes distintos; e todas as ocorrências de um nome numa configuração são decoradas do mesmo modo.

[3] A notação $\mathbb{N}^{\boldsymbol{Exp}}$ denota o conjunto dos subconjuntos de $\boldsymbol{Exp}$.

Entidades	$p, q \in \boldsymbol{Pri}$		
Níveis de Segurança	$l, j, k \subseteq \boldsymbol{Pri}$		
Políticas de Fluxo	$F, G \subseteq \boldsymbol{Pri} \times \boldsymbol{Pri}$		
Identificadores de Processo	$\check{m}, \check{n} \in \boldsymbol{\check{Nam}}$		
Efeitos	s	$::=$	$\langle l, l, l \rangle$
Variáveis de Tipo	t		
Tipos	$\tau, \sigma, \theta \in \boldsymbol{Typ}$	$::=$	$t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathrm{ref}_{l,\check{m}_j} \mid \tau \xrightarrow[G,\check{m}_j]{s} \sigma$

Figura 1: Sintaxe das Anotações de Segurança e dos Tipos

Variáveis	$x, y \in \boldsymbol{Var}$		
Nomes de Domínio	$d \in \boldsymbol{Dom}$		
Nomes de Processo	$m, n \in \boldsymbol{Nam}$		
Identificadores de Referência	$u, v \in \boldsymbol{Ref}$		
Nomes de Referência	a, b, c	$::=$	$m_j.u$
Nomes de Processo Decorados		$::=$	m_j
Nomes de Referência Decorados		$::=$	$a_{l,\theta}$
Valores	$V \in \boldsymbol{Val}$	$::=$	$() \mid x \mid a_{l,\theta} \mid (\lambda x.M) \mid tt \mid ff$
Pseudo-valores	$W \in \boldsymbol{Pse}$	$::=$	$V \mid (\varrho x.W)$
Expressões	$M, N \in \boldsymbol{Exp}$	$::=$	$W \mid (M\ N) \mid (M; N) \mid$ $(\mathrm{ref}_{l,\theta}\ M) \mid (?\ N) \mid (M :=^{?} N) \mid$ $(\text{if } M \text{ then } N_t \text{ else } N_f) \mid$ $(\mathrm{thread}_l\ M) \mid (\text{flow } F \text{ in } M) \mid$ $(\text{goto } d)$

Figura 2: Sintaxe das Expressões

Processos		$::=$	$M^{m_j} \quad (\in \boldsymbol{Exp} \times \boldsymbol{Nam} \times 2^{\boldsymbol{Pri}})$
Grupo de Processos	P	$:$	$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \to \boldsymbol{Exp}$
Mapeador de Posições	T	$:$	$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \to \boldsymbol{Dom}$
Memória	S	$:$	$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Ref} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Typ}) \to \boldsymbol{Val}$
Redes	X, Y	$::=$	$d[\ P, S\ ] \mid X \parallel Y$
Configurações		$::=$	$\langle P, T, S \rangle$

Figura 3: Sintaxe das Configurações

Denotamos por $\{x \mapsto W\}M$ a substituição por W (sem captura de variáveis) de todas as ocorrências livres de x em M. A operação de acrescentar ou modificar a imagem de um objecto z em z' num mapeamento Z é denotada $[z := z']Z$.

Contextos de avaliação e suas políticas de fluxo De modo a definir a ordem de avaliação, torna-se conveniente escrever expressões usando contextos de avaliação. Intuitivamente, as expressões que são colocadas dentro desses contextos devem ser executadas em primeiro lugar. Escrevemos $\mathrm{E}[M]$ para denotar uma expressão em que a sub-expressão M é colocada dentro do contexto E, obtida substituindo todas as ocorrências de $[]$ em E por M. Os contextos de avaliação desta linguagem definem uma ordem de avaliação de chamada por valor (ver Figura 4). A avaliação *não* é permitida dentro de processos que ainda não tenham sido criados.

Os fluxos de informação que ocorrem em M serão ou não permitidos dependendo das políticas de fluxo que são declaradas no contexto de avaliação em que M executa. Denotamos por $\lceil \mathrm{E} \rceil$ a política de fluxo que é permitida pelo contexto de avaliação E. Ela colecciona todas as políticas que são declaradas nesse contexto, usando união entre conjuntos, numa política de fluxo única. Obtemos então a seguinte definição:

Definição 2.1 (Política de fluxo declarada por um contexto de avaliação)**.** *A* política de fluxo declarada pelo contexto de avaliação E *é dada por* $\lceil \mathrm{E} \rceil$ *onde:*

$$\lceil [] \rceil = \emptyset, \qquad \lceil (\text{flow } F \text{ in } \mathrm{E}) \rceil = F \cup \lceil \mathrm{E} \rceil,$$
$$\lceil \mathrm{E}'[\mathrm{E}] \rceil = \lceil \mathrm{E} \rceil, \ \textit{se } \mathrm{E}' \textit{ não contém declarações de fluxo.}$$

Semântica etiquetada de passo único As transições etiquetadas da nossa semântica (de passo único) são definidas entre configurações. As regras de avaliação estão definidas na Figura 5. Começamos por definir as transições de processos isolados (omitimos as chavetas na notação de grupos que contêm um único elemento). Estas podem ser decoradas com um processo N^{n_k} que tenha sido gerado durante essa transição, onde caso contrário $N^{n_k} = ()$. As últimas três regras usam informação contida na etiqueta de modo a adicionar ao grupo os processos que são eventualmente gerados. Pela última regra podemos ver que a execução de um grupo de processos é composicional.

As transições são também decoradas com a política de fluxo que é declarada pelo contexto de avaliação em causa. Poderá no entanto verificar que as transições não dependem da etiqueta de fluxo F que as decora. A avaliação de (flow F in M) consiste simplesmente na avaliação de M, anotada com a política de fluxo F que a engloba (como vimos acima). A vida de uma declaração de fluxo termina quando a expressão M que está a ser avaliada termina (isto é, M torna-se num valor).

A avaliação das seguintes expressões depende unicamente das próprias expressões: a aplicação de uma função com parâmetro x e corpo M ao um valor V retorna a substituição de todas as ocorrências livres de x em M por V; a condição que testa um valor booleano V e tem ramos N_t e N_f retorna N_t se o valor for tt e N_f se o valor for ff; a composição sequencial de uma valor e de uma expressão N resulta em N; o ponto fixo de um pseudo-valor W ligado por x resulta na substituição das ocorrências livres de x em W pela própria expressão.

Por outro lado, a avaliação de algumas expressões pode mudar e depender da memória: a criação de uma referência de nível de segurança l e tipo θ contendo o valor V retorna uma referência com um nome que não ocorre ainda na memória (digamos a), e adiciona o par $((a, l, \theta), V)$ à memória; a leitura de uma referência, se não for suspensa, retorna o valor

$$\textit{Contextos de Avaliação}\quad \mathrm{E} ::= [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid (\mathrm{ref}_{l,\theta}\ \mathrm{E}) \mid (?\ \mathrm{E}) \mid (\mathrm{E} :=^{?} N) \mid (V :=^{?} \mathrm{E}) \mid (\text{if E then } N_t \text{ else } N_f) \mid (\text{flow } F \text{ in E})$$

Figura 4: Contextos de Avaliação

$$\langle \mathrm{E}[((\lambda x.M)\ V)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[\{x \mapsto V\}M]^{m_j}, T, S\rangle$$

$$\langle \mathrm{E}[(\text{if } tt \text{ then } N_t \text{ else } N_f)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[N_t]^{m_j}, T, S\rangle$$

$$\langle \mathrm{E}[(\text{if } \mathit{ff} \text{ then } N_t \text{ else } N_f)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[N_f]^{m_j}, T, S\rangle$$

$$\langle \mathrm{E}[(V; N)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[N]^{m_j}, T, S\rangle$$

$$\langle \mathrm{E}[(\varrho x.W)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[\{x \mapsto (\varrho x.W)\}W]^{m_j}, T, S\rangle$$

$$\langle \mathrm{E}[(\text{flow } F \text{ in } V)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[V]^{m_j}, T, S\rangle$$

$$\frac{T(n_k) = T(m_j)}{\langle \mathrm{E}[(?\ n_k.u_{l,\theta})]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[V]^{m_j}, T, S\rangle},\ \textit{onde } S(n_k.u_{l,\theta}) = V$$

$$\frac{T(n_k) = T(m_j)}{\langle \mathrm{E}[(n_k.u_{l,\theta} :=^{?} V)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[()]^{m_j}, T, [n_k.u_{l,\theta} := V]S\rangle}$$

$$\langle \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[a_{l,\theta}]^{m_j}, T, [a_{l,\theta} := V]S\rangle,\ a = m_j.u \textit{ novo em } S$$

$$\langle \mathrm{E}[(\mathrm{thread}_k\ N)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{N^{n_k}} \langle \mathrm{E}[()]^{m_j}, [n_k := T(m_j)]T, S\rangle,\ n \textit{ novo em } T$$

$$\langle \mathrm{E}[(\text{goto } d)]^{m_j}, T, S\rangle \xrightarrow[\lceil \mathrm{E} \rceil]{\emptyset} \langle \mathrm{E}[()]^{m_j}, [m_j := d]T, S\rangle$$

$$\frac{\langle \{M^{m_j}\}, T, S\rangle \xrightarrow[F]{\emptyset} \langle \{M'^{m_j}\}, T', S'\rangle}{\langle \{M^{m_j}\}, T, S\rangle \xrightarrow[F]{} \langle \{M'^{m_j}\}, T', S'\rangle} \qquad \frac{\langle \{M^{m_j}\}, T, S\rangle \xrightarrow[F]{N^{n_k}} \langle \{M'^{m_j}\}, T', S'\rangle}{\langle \{M^{m_j}\}, T, S\rangle \xrightarrow[F]{} \langle \{M'^{m_j}, N^{n_k}\}, T', S'\rangle}$$

$$\frac{\langle P, T, S\rangle \xrightarrow[F]{} \langle P', T', S'\rangle \quad \langle P \cup Q, T, S\rangle \textit{ é bem formado}}{\langle P \cup Q, T, S\rangle \xrightarrow[F]{} \langle P' \cup Q, T', S'\rangle}$$

Figura 5: Semântica

em que a memória mapeia essa referência; a atribuição de um valor V a uma referência $a_{l,\theta}$, se não for suspensa, retorna $()$ e actualiza a memória substituindo toda a ocorrência do par $((a, l, \theta), V')$ (em que V' é o antigo valor de a) por $((a, l, \theta), V)$. A relação destas operações com o mapeador de posições é a seguinte: quando uma referência é criada por um processo m, é-lhe atribuída um nome novo da forma $m.u$, sendo u um identificador de referência novo; a leitura e escrita de uma referência que pertence a um processo n apenas pode ser realizada por outro processo m se m e n estiverem localizados no mesmo domínio; quando um processo é criado, o seu nome (e posição) é adicionado ao mapeador de posições; quando a instrução (goto d) é executada por um processo m, a posição de m no mapeador de posições é actualizada para d.

Quando um novo processo é criado, a política de fluxo que é permitida pelo contexto de avaliação do processo pai não é mantida. Como consequência, o processo que é gerado executa sob uma política de fluxo mais rígida, sendo por isso mais restrito do ponto de vista da confidencialidade, do que o processo que o gerou.

Propriedades da semântica Pode-se provar que a semântica preserva as condições de boa formação das configurações, e que uma configurações contendo uma única expressão tem no máximo uma possibilidade de transição, a menos da escolha de novos nomes.

3 Posicionamento do problema

Nesta secção explicamos informalmente os conceitos fundamentais relacionados com a propriedade de não-interferência, e damos exemplos de tipos diferentes de fugas de informação, culminando nas *fugas por migração*. Apresentamos também a nossa abordagem à declassificação, através da introdução de uma instrução – a declaração de fluxo – e uma propriedade alternativa à não-interferência – a não-divulgação. Abordamos ainda o processo de generalização desta propriedade a um contexto de computação em rede.

3.1 Princípios básicos da não-interferência

Começamos por recordar a ideia da não-interferência, considerando um sistema com apenas dois níveis de segurança, *público* (ou baixo, "low", L) e *privado* (alto, "high", H). Estes níveis de segurança são atribuídos a referências, significando que a informação nelas contida apenas pode ser lida por entidades com as permissões de segurança correspondentes. Informalmente, a propriedade de não-interferência declara que a informação apenas deve ser autorizada a fluir de um nível mais baixo para outro mais alto (mais seguro). Vamos ver exemplos de diferentes formas de *fluxos inseguros de informação*, ou *interferências*, que ocorrem quando o *valor inicial* de referências altas pode influenciar o *valor final* de referências mais baixas[4].

Fugas directas e fugas por controlo A forma mais simples de fluxo inseguro é a atribuição do valor de uma referência alta a uma referência baixa:

$$(b_L :=^{?} (?\ a_H)) \tag{1}$$

[4]Utilizando alguma liberdade de linguagem, designaremos por "referências altas" ou "referências baixas" aquelas a que foi atribuído um nível de segurança alto ou baixo, respectivamente. Esta prática será alargada a outras situações, como por exemplo a partes de memórias ou de estados, testes, relações de igualdade e domínios, sendo o seu significado preciso clarificado pelo contexto.

É designada por fluxo inseguro explícito, e consiste numa *fuga de informação directa*. Formas mais subtis de fluxo, designadas fluxos implícitos, podem ser induzidas pelo fluxo de controlo (*fugas por controlo*), como no programa

$$(\text{if } a_H = tt \text{ then } (b_L :=^? tt) \text{ else } (b_L :=^? ff)) \tag{2}$$

em que no final da execução o valor b_L pode revelar informação acerca de a_H. Um programa parecido pode ser escrito usando um ciclo:

$$((b_L :=^? ff); (\text{while } a_H = tt \text{ do } ((b_L :=^? tt); (a_H :=^? ff)))) \tag{3}$$

Outros programas podem ser considerados seguros ou não dependendo do contexto em que aparecem. Por exemplo, o programa

$$((\text{while } a_H = tt \text{ do } ()); (b_L :=^? ff)) \tag{4}$$

pode ser considerado seguro num contexto sequencial (já que sempre que termina apresenta o mesmo valor ff em b_L), enquanto que na presença de paralelismo ele poderá tornar-se crítico, como veremos a seguir.

Fugas de ordem-superior A linguagem que estamos a considerar é de *ordem superior*, o que significa intuitivamente que programas podem ser guardado na memória de outro programa, e executados a partir daí. Isto possibilita a aparição de outros tipos de fugas de informação, aqui designados *fugas de ordem-superior*. Para ilustrá-las, vamos analisar o programa seguinte, notando que a referência alta a_H pode conter funções com comportamentos diferentes, por exemplo uma que quando aplicada a um argumento entra num ciclo infinito ($\lambda y.\mathsf{loop}$) ou outra que não faz nada ($\lambda y.(\lambda x.x)$):

$$((?\ a_H)(); (b_L :=^? tt)) \tag{5}$$

A aplicação das funções contidas em a_H a $()$ desencadearia duas expressões com comportamentos diferentes, de maneira análoga aos exemplos anteriores. Nesta perspectiva, a operação de leitura da referência alta a_H pode ser vista como um "teste alto" (no sentido dos exemplos 2 e 4) que pode terminar ou não.

Fugas por divergência Num contexto sequencial faz sentido olhar apenas para os valores finais produzidos por um programa, ignorando assim todas as suas computações divergentes (i.e. que não terminam). De facto, são apenas os valores finais de computações que terminam que podem ser utilizados por outros programas que executam sequencialmente com esse programa. Além disso, se nesse contexto uma computação entra num ciclo infinito, não é possível que outros programas o interrompam, já que nunca terão oportunidade de correr. No entanto, isto já não se verifica num contexto concorrente. O Exemplo 4 pode ser usado num programa, que termina sempre, sendo ainda inseguro no sentido mencionado acima. Já o programa seguinte, composto por três processos a executar concorrentemente (usando informalmente a notação ' || '), em que se admite que as referências c_H e c'_H têm inicialmente o valor ff,

$$\begin{array}{c}(\text{if } a_H \text{ then } (c_H :=^? tt) \text{ else } (c'_H :=^? tt)) \ ||\\ ((\text{while } \neg c_H \text{ do } ()); ((b_L :=^? ff); (c'_H :=^? tt))) \ ||\\ ((\text{while } \neg c'_H \text{ do } ()); ((b_L :=^? tt); (c_H :=^? tt)))\end{array} \tag{6}$$

verificamos que termina sempre, e que o valor final de b_L reflecte o valor inicial da referência alta a_H. O fluxo inseguro que ocorre aqui é normalmente designado por *fuga por divergência*, já que resulta do "comportamento divergente" de uma porção do programa.

Contextos concorrentes O último exemplo revela a importância da *composicionalidade* de propriedades de segurança. Na verdade, se cada um dos três processos fosse considerado seguro, então seria de esperar que a sua composição concorrente produzisse um programa seguro. Por esta razão, interessa-nos uma noção de não-interferência que permita identificar que alguma das componentes de um programa como o acima é insegura. As noções de não-interferência que são geralmente usadas para estudar computação em contextos concorrentes recorrem ao conceito de *bissimulação*.

Intuitivamente, bissimulações são relações entre programas que têm o mesmo comportamento, sendo esse comportamento observável a cada passo de execução dos dois programas a comparar. Elas permitem assim uma análise mais refinada das propriedades de fluxo de informação do que a mera observação dos estados inicial e final, pois observam o que acontece *durante a execução* de um programa. Em particular, permitem distinguir as possíveis computações divergentes de programas como 4, daquelas que acabam por provocar alterações nas referências baixas do programa.

3.2 Da não-interferência à não-divulgação

A aplicabilidade da não-interferência tem sido largamente debatida. Um dos seus problemas é que, *por definição*, rejeita programas que deliberadamente *declassificam* informação de níveis de segurança altos para níveis mais baixos, impedindo assim o uso de programas que são muito comuns, e mesmo imprescindíveis.

Um exemplo típico é um procedimento de verificação de palavras-passe, cuja finalidade é restringir o acesso a um serviço apenas a utilizadores possuidores de uma palavra-passe secreta. Isto poderia ser escrito do seguinte modo:

$$\text{if } (input_L = password_H) \text{ then } \ldots \text{ else } print_L(\text{“Palavra-passe errada.”}) \qquad (7)$$

Tais programas revelam, a qualquer utilizador que tente o “log-in”, pelo menos um bit de informação. Claramente, estes programas são considerados inseguros se a política de segurança proibir o fluxo dos níveis H pra L, e não poderia correr num sistema que exija que os seus programas satisfaçam a propriedade de não-interferência. No entanto, faz sentido aceitar tais programas, mantendo pois a possibilidade de controlar os fluxos de informação que ocorram noutras partes do programa (e.g. depois do utilizador ter entrado com sucesso no sistema).

Uma visão da declassificação A procura de mecanismos de aceitação da declassificação sob o escrutínio de alguma política de fluxo de informação é um problema aliciante que tem motivado muita investigação [33]. No entanto, diferentes abordagens a esta questão têm apontado para diferentes objectivos. Podemos agrupá-las segundo duas perspectivas de declassificação distintas:

1. Como *justificar* cada operação de declassificação implementada num programa, i.e., provar que ela não acaba por revelar “demasiado”?

   Abordagens que se preocupam com esta questão, que parece estar fora do alcance da análise estática, procuram garantir que não é comportável explorar as fugas de informação que são autorizadas pelo programa de modo a obter declassificação “não-intencional”. Técnicas para atingir este objectivo incluem a quantificação do volume de informação que um programa poderá declassificar, ou argumentos probabilísticos (por exemplo [37, 39].

2. Como *aceitar* programas que declassificam propositadamente informação, preservando simultaneamente alguma propriedade de segurança de fluxos de informação?

   Esta pergunta inclui três desafios: Como conceber ferramentas apropriadas para providenciar ao programador os meios de codificar a intenção de declassificar informação? Como exprimir formalmente a propriedade de fluxos de informação apropriada que programas considerados seguros devem satisfazer? Como criar mecanismos para rejeitar programas que não satisfaçam a propriedade desejada?

Aqui atacamos os três desafios da segunda pergunta, deixando de lado a tarefa de provar que os programas satisfazem as especificações do programador referentes a "o quê?" e "quanta?" informação deve ser declassificada.

Declaração de fluxo Uma vez deixada de lado a questão da validação de programas que declassificam informação intencionalmente, a linguagem de programação propriamente dita deve ser tão flexível quanto possível a exprimir essas intenções. Foi com esse objectivo que introduzimos na nossa linguagem uma declaração de fluxo que permite manipular dinamicamente a política de segurança, e que estabelece quais os fluxos de informação que são legais. Tal como vimos na secção anterior, a instrução escreve-se na forma (flow F in M), e recebe dois parâmetros: a política de fluxo F, que declara que os fluxos expressos em F são válidos e um programa M, que é o âmbito da declaração de fluxo. O significado é que M deve obedecer à política de fluxo que vigora no contexto em que (flow F in M) executa, *estendida* com F.

Como exemplo, se tivermos as entidades A e B, e construindo níveis de segurança como conjuntos de entidades, então o programa

$$(\text{flow } A \prec B \text{ in } (b_{\{B\}} :=^{?} (?\ a_{\{A\}}))) \tag{8}$$

é legal, já que a declaração $A \prec B$ estipula que a informação é autorizada a fluir de A para B no subprograma $(b_B :=^{?} (?\ a_A))$. Isto consiste numa operação de *declassificação* – a não ser que, evidentemente, os fluxos de A para B já sejam permitidos pelo contexto em que o programa é colocado. Deve estar claro, no entanto, que o comando

$$(\text{flow } C \prec B \text{ in } (b_{\{B\}} :=^{?} (?\ a_{\{A\}}))) \tag{9}$$

não é legal, a não ser que a política do contexto em que é executado permita que a informação flua do nível A para o C (ou B).

Usando mais uma vez '||' para exprimir concorrência, podemos também ter políticas de fluxo diferentes vigorando simultaneamente sobre diferentes partes do programa – incluindo o caso de programas concorrentes:

$$(\text{flow } A \prec B \text{ in } M) \mid\mid (\text{flow } C \prec D \text{ in } N) \tag{10}$$

O exemplo acima mostra que a declaração de fluxo é especialmente adequada para executar em contextos que envolvam a partição de programas em subprogramas com políticas de fluxo independentes a serem executados em concorrência, possivelmente em diferentes domínios de uma rede.

Introduzindo a não-divulgação Tal como foi apontado em [28], apesar da sua inflexibilidade, a não-interferência é ainda um conceito simples e elegante que a comunidade de investigação em segurança gostaria de reter. É por isso desejável encontrar uma alternativa à não-interferência que preserve a sua "essência". Tendo isso em vista, a nossa nova

propriedade de confidencialidade, que chamamos *não-divulgação*, foi concebida como uma generalização natural da não-interferência que permite declassificação. Informalmente, diz que um programa P é seguro se em cada passo de execução ele satisfizer não-interferência *em relação à política de fluxo que vigora nesse passo.*

Referimos acima que, em particular no caso de sistemas concorrentes, a não-interferência pode ser expressa convenientemente usando bissimulações. Estas são baseadas em semânticas de passo único, que especificam transições entre estados sucessivos de configurações de um programa. Exprimir execuções por meio de passos únicos, por oposição a descrevê-las com base nos resultados finais de uma computação, é apropriado para lidar com a nossa declaração de fluxo, em que a segurança de um passo de execução é definida localmente. Na verdade, é suficiente ler a etiqueta associada a cada transição

$$\langle P, T, S\rangle \underset{F}{\longrightarrow} \langle P', T', S'\rangle \tag{11}$$

de modo a determinar qual a política de fluxo que vigora para esse passo em particular. Por essa razão a nossa propriedade de não-divulgação vai ser formulada em termos de uma bissimulação, parecida com a que é classicamente usada para definir não-interferência.

3.3 Da não-divulgação à não-divulgação em redes

Estando interessados em controlar fluxos de informação num contexto distribuído com mobilidade de código e em estudar as questões de segurança que surgem nesse contexto, optámos por uma linguagem onde a noção de localização joga um papel importante. Programas estão distribuídos por uma rede, e a possibilidade de execução ou falha de um programa não pode ser garantida por um domínio isoladamente – pode, por exemplo, depender da sua localização. A pergunta agora é: Podem estas falhas ser exploradas como canais de fuga de informação escondidos? Na presença de mobilidade de código, a resposta é *Sim*, já que novas fugas de informação, as *fugas por migração*, surgem do facto de a execução ou suspensão de programas poder depender de informação secreta.

Fugas por migração A observação chave é que a situação de suspensão de um processo devido a uma tentativa de acesso a uma referência remota pode ser desbloqueada por outros processos. Isto permite escrever programas que são semelhantes ao Exemplo 6, substituindo a não-terminação por um acesso suspenso, e o seu desbloqueamento é codificado recorrendo a uma migração apropriada:[5]

$$\begin{array}{c} d[(\text{if } a_H \text{ then } (\text{goto } d_1) \text{ else } (\text{goto } d_2))^{n_k}] \parallel \\ d_1[((n_k.x_\top :=^? 0); (m_{1j_1}.y_L :=^? 1))^{m_{1j_1}}] \parallel \\ d_2[((n_k.x_\top :=^? 0); (m_{2j_2}.y_L :=^? 2))^{m_{2j_2}}] \end{array} \tag{12}$$

Então, dependendo no valor da referência alta a, ocorreriam atribuições diferentes às referências baixas $m_1.y_L$ e $m_2.y_L$. O mesmo exemplo mostra a potencial fuga de informação acerca das posições dos processos m_1 e m_2 por via das suas próprias referências baixas $m_1.y_L$ e $m_2.y_L$.

Estendendo a não-divulgação a redes As propriedades de segurança clássicas, concebidas para computações locais, não são adequados para lidar com fluxos de informação em ambientes distribuídos com mobilidade. De facto, como a localização das referências pode ser por si só uma fonte de fugas de informação, a noção de segurança de programas

[5] Para aligeirar a notação, omitiremos os estados quando estes não são relevantes.

deve ter isso em conta. Por isso, alargámos a observação do comportamento dos processos ao seu comportamento "migratório", para além das modificações que faz na memória.

Como a visibilidade de processos é uma consequência da possibilidade ou não de aceder a qualquer das suas referências, associamos aos processos um nível de segurança que é um minorante dos níveis das referências que lhe pertencem. De seguida, alargamos o campo de observação da bissimulação aos mapeadores de posições de programas numa rede, para além da tradicional observação das memórias em que se baseiam as propriedades de segurança de fluxo locais. Assim, a formalização da não-divulgação em redes torna-se uma generalização directa da não-divulgação simples. Na secção que se segue vamos formalizar as ideias que acabámos de expor.

4 A propriedade de segurança

Nesta secção definimos formalmente a propriedade de não-divulgação em redes. Tudo começa com a apresentação de uma "ordem" entre os níveis de segurança: isto será feito por meio de um pré-reticulado de segurança, definido em termos de uma relação de fluxo que é parametrizada pela política de fluxo do contexto. Depois exibimos uma relação de indistinguibilidade de estados e a definição de bissimulação na qual se baseia a nossa propriedade de segurança. Por fim, justificamos a propriedade de segurança com mais alguns exemplos, e enunciamos algumas das suas propriedades.

4.1 Pré-reticulados de segurança

Os exemplos da secção anterior assumiram um contexto simplificado com apenas dois níveis de segurança, em que L ("baixo") era um nível inferior a H ("alto"). Consideramos agora um quadro mais geral, em que qualquer número de níveis de segurança é organizado segundo uma variante de um reticulado de segurança [11], que passamos a definir.

Admitimos que, na nossa abordagem, níveis de segurança j, k, l são conjuntos de entidades $p, q \in \boldsymbol{Pri}$ que representam privilégios de leitura de referências. O nosso objectivo é assegurar que a informação contida numa referência a_{l_1} (omitindo a anotação de tipo) não flui para outra referência b_{l_2} que dê acesso de leitura a uma entidade não autorizada p, isto é, tal que $p \in l_2$ mas $p \notin l_1$. Podemos interpretar a inclusão inversa de níveis de segurança como indicando os fluxos de informação permitidos: se $l_1 \supseteq l_2$ então informação em a_{l_1} pode ser transferida para b_{l_2}, já que as entidades autorizadas a ler b já estavam também autorizadas a ler a. De facto, a inclusão inversa forma uma estrutura de *reticulado* sobre os níveis de segurança, em que as operações supremo e ínfimo correspondem respectivamente à união e intersecção de conjuntos. Veremos de seguida como este reticulado pode ser configurado usando relações entre entidades. Relembramos que uma *política de fluxo* F, G é uma relação binária em $\boldsymbol{Pri}$. Um par $(p, q) \in F$, mais frequentemente escrito $p \prec q$, deve ser lido como "informação pode fluir da entidade p para a entidade q", isto é, mais precisamente, "*tudo o que p é autorizado a ler também pode ser lido por q*". Denotamos, como é usual, por F^* o fecho reflexivo e transitivo de F.

Vamos agora definir a *pré-ordem entre níveis de segurança* $\preceq_F$ que é determinada pela política de fluxo F. Para isso usamos a noção de *fecho-superior* por respeito a F de um nível de segurança l, definido do seguinte modo:

$$l \uparrow_F = \{q \mid \exists p \in l.\ p\ F^*\ q\} \tag{13}$$

O fecho superior por F de l contém todas as entidades que são autorizadas pela política F a ler o conteúdo da uma referência com etiqueta l. Podemos agora derivar uma *relação*

de fluxo mais permissiva

$$l_1 \preceq_F l_2 \stackrel{\text{def}}{\Leftrightarrow} \forall q \in l_2 \,.\, \exists p \in l_1 \,.\, p\ F^*\ q \Leftrightarrow (l_1 \uparrow_F) \supseteq (l_2 \uparrow_F) \tag{14}$$

e usá-la para definir o pré-reticulado que é determinado por uma política de fluxo. Note que $\preceq_F$ estende $\supseteq$ no sentido em que $\preceq_F$ é maior do que $\supseteq$ e em que $\preceq_\emptyset = \supseteq$.

Definição 4.1 (Pré-reticulado de segurança)**.** *Dado um conjunto* ***Pri*** *de* entidades *e uma política de fluxo F em* ***Pri*** $\times$ ***Pri****, o par* $(2^{\boldsymbol{Pri}}, \preceq_F)$ *é um* pré-reticulado de segurança*, onde as operações* supremo *($\curlywedge_F$) e* ínfimo *($\curlyvee_F$) são dadas respectivamente pela união e pela intersecção dos fechos-superiores por F:*

$$l_1 \curlywedge_F l_2 = l_1 \cup l_2 \qquad l_1 \curlyvee_F l_2 = (l_1 \uparrow_F) \cap (l_2 \uparrow_F)$$

Políticas de fluxo globais Tem sido tradicional em estudos sobre a não-interferência organizar os níveis de segurança segundo um reticulado de segurança estático, seguindo uma *política de fluxo global* que é definida externamente ao programa, não havendo por isso necessidade de parametrizar a relação de fluxo entre níveis de segurança. No entanto, é difícil imaginar qual a política de fluxo global a adoptar quando estamos perante uma rede descentralizada de computadores.

A abordagem concreta que acabámos de apresentar permite-nos utilizar a relação de fluxo de uma maneira mais flexível, escolhendo em cada caso qual a política de fluxo a considerar quando comparamos dois níveis de segurança. Por uma questão de generalidade, neste estudo vamos admitir a existência de uma *política de fluxo global* G (isto é, aquela que vigora na ausência de declarações de fluxo) que corresponde à intersecção de todas as políticas de fluxo de uma rede. Para instanciar G, podemos sempre recorrer à escolha prática e conservadora que é a política de fluxo vazia, da qual resulta a relação de fluxo mínima $\preceq$, que claramente respeita todos os pré-reticulados de segurança. Em suma, no que se segue, os fluxos de informação que iremos permitir numa expressão M, que é colocada num contexto de avaliação E[], deverão satisfazer a relação de fluxo $\preceq_{G \cup \lceil \mathrm{E} \rceil}$.

4.2 Uma definição baseada em bissimulação

Definimos agora a nossa propriedade de segurança, em termos das relações de fluxo $\preceq_F$ acima definidas, onde F é a política de fluxo em vigor.

Igualdade baixa "*Igualdade baixa*" é uma designação informal dada a uma relação de igualdade que considera indistinguíveis duas memórias que coincidem na sua "parte baixa". A relação é estabelecida entre memórias cujas referências são etiquetadas com níveis de segurança. A parte baixa de uma memória é então definida por respeito a um nível de segurança l que é considerado como sendo "baixo", e consequentemente todos os níveis inferiores a l (por respeito a uma relação de fluxo $\preceq_F$) são também baixos. Assim, tanto l como F são usados como parâmetros da igualdade baixa que definimos a seguir.

Como estamos num contexto distribuído, a nossa noção de igualdade baixa será estendida a estados (ver Subsubsecção 3.3). De facto, como mostra o Exemplo 12, a posição de processos na rede pode revelar informação acerca dos valores que têm na memória, pela detecção da presença das referências que lhe pertencem. Assim, processos que possuem "referências baixas" podem ser vistas como "processos baixos". Estamos então também interessados em identificar estados em que processos baixos têm a mesma localização.

Intuitivamente, dois estados são relacionados por uma igualdade baixa se tiverem o mesmo domínio baixo, e se mapearem do mesmo modo as referências e processos que estão etiquetados com um nível de segurança baixo.

Definição 4.2 (Parte baixa de estados)**.** *A* parte baixa de um estado $\langle T, S\rangle$ *é composta pela* parte baixa da memória S *e pela* parte baixa do mapeador de posições T*, por respeito a uma política de fluxo* F *e a um nível de segurança* l*, dados por:*

$$\begin{array}{lll} S \upharpoonright^{F,l} & \stackrel{def}{=} & \{(a_{k,\theta}, V) \mid (a_{k,\theta}, V) \in S \ \&\ k \preceq_F l\} \\ T \upharpoonright^{F,l} & \stackrel{def}{=} & \{(n_k, d) \mid (n_k, d) \in T \ \&\ k \preceq_F l\} \end{array}$$

Igualdade baixa entre estados é então definida pontualmente, para um nível de segurança considerado "baixo" da seguinte maneira:

Definição 4.3 (Igualdade baixa)**.** *A* igualdade baixa entre estados $\langle T_1, S_1\rangle$ *e* $\langle T_2, S_2\rangle$ *por respeito a uma política de fluxo* F *e um nível de segurança* l *é dada pela conjunção entre a* igualdade baixa entre memórias S_1 *e* S_2 *e a* igualdade baixa entre mapeadores de posições T_1 *e* T_2 *por respeito ao mesmo nível de segurança e política de fluxo:*

$$\langle T_1, S_1\rangle =^{F,l} \langle T_2, S_2\rangle \ \stackrel{def}{\Leftrightarrow}\ T_1 \upharpoonright^{F,l} = T_2 \upharpoonright^{F,l} \ \text{ e } S_1 \upharpoonright^{F,l} = S_2 \upharpoonright^{F,l}$$

Esta relação é transitiva, reflexiva e simétrica. Usaremos de aqui em diante o facto de que $F \subseteq F'$ e $\langle T_1, S_1\rangle =^{F',l} \langle T_2, S_2\rangle$ implica que $\langle T_1, S_1\rangle =^{F,l} \langle T_2, S_2\rangle$.

A propriedade de não-divulgação em redes Bissimulações são frequentemente utilizadas para relacionar programas não-determinísticos de acordo com o seu comportamento. Fornecem assim uma maneira natural de formular propriedades de segurança em contextos não-determinísticos [32, 35, 4, 12, 33]. A ideia é que, escolhendo cuidadosamente uma bissimulação, podemos exprimir a exigência de que dois programas estão relacionados se apresentam o mesmo comportamento na parte baixa de dois estados. Assim, mostrar que um programa é bissimilar a si próprio permite concluir que, durante a execução, a parte alta do estado em que executa nunca interfere com a parte baixa, i.e., não podem ocorrer fugas de informação. Um programa seguro é então um que se relaciona consigo mesmo por uma bissimulação apropriada.

A nossa bissimulação para redes é baseada numa noção de bissimulação entre conjuntos de processos P por respeito a um nível de segurança l e, já que a noção de "ser baixo" usa a relação de fluxo $\preceq_G$, a política de fluxo global G também aparece como um parâmetro. Denotaremos por $\twoheadrightarrow$ o fecho reflexivo da união das transições $\underset{F}{\longrightarrow}$, para todo o F.

Definição 4.4 (Bissimulação-(G,l) e $\approx_{G,l}$)**.** *Uma* bissimulação-(G,l) *é uma relação simétrica* $\mathcal{R}$ *entre conjuntos de processos tais que:*

$$\begin{array}{l} P_1 \ \mathcal{R}\ P_2 \ \&\ \langle P_1, T_1, S_1\rangle \underset{F}{\longrightarrow} \langle P_1', T_1', S_1'\rangle \ \&\ \langle T_1, S_1\rangle =^{G\cup F,l} \langle T_2, S_2\rangle \\ \text{e } (*) \text{ implica que} \\ \exists T_2', P_2', S_2' \,.\, \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle \ \&\ \langle T_1', S_1'\rangle =^{G,l} \langle T_2', S_2'\rangle \ \&\ P_1' \ \mathcal{R}\ P_2' \\ \\ \text{onde:} \\ (*)\ \mathrm{dom}({S_1}' - S_1) \cap \mathrm{dom}(S_2) = \emptyset \text{ e } \mathrm{dom}({T_1}' - T_1) \cap \mathrm{dom}(T_2) = \emptyset \end{array}$$

Como para todo G *e* l *existe uma bissimulação-*(G,l) *(por exemplo o conjunto* $\boldsymbol{Val} \times \boldsymbol{Val}$*), consequentemente, existe a maior bissimulação-*(G,l)*, que é a união de todas as bissimulações-*(G,l)*, que denotamos por* $\approx_{G,l}$*.*

Quando P_1 executa uma transição no âmbito da política de fluxo F, é-lhe permitido ler a parte baixa do estado inicial $\langle T_1, S_1\rangle$ de acordo com a política de fluxo corrente

$G \cup F$. Recorde-se que estes nomes são etiquetados com o nível de segurança l' tal que $l' \preceq_{G \cup F} l$. No entanto, as diferenças observáveis sobre os estados resultantes restringem-se à igualdade por respeito à política global de fluxo G.

A diferença principal entre esta definição e a que é usada para a não-interferência clássica é o requisito mais forte $\langle T_1, S_1 \rangle =^{G \cup F, l} T_2 S_2$, no lugar de uma igualdade relativa apenas à política de fluxo global. Começando o jogo da bissimulação com um par de estados que coincidem numa parte baixa "mais extensa", a condição sobre o comportamento do programa P_2 fica enfraquecida. Como consequência, esta definição pode relacionar mais programas, aceitando mais programas como sendo seguros.

É importante observar que a bissimulação $\approx_{G,l}$ não é reflexiva. Por exemplo, a expressão insegura $(v_B :=^? (?\ u_A))$ não é bissimilar a si própria se $A \not\preceq_G B$. Esta é, aliás, a chave da relação de segurança, já que, como podemos ver pela definição seguinte, apenas deve ser "reflexiva" por respeito a programas seguros. Intuitivamente, a nossa propriedade de segurança deve estipular que, a cada passo de execução de um processo numa rede, o fluxo de informação que ocorre respeita a política de fluxo global, estendida com a política de fluxo F que é declarada pelo contexto em que o programa executa.

Definição 4.5 (Não-divulgação em redes por respeito a G)**.** *Diz-se que um grupo de processos P satisfaz a política de* Não-divulgação em Redes *(ou é* segura *do ponto de vista da Não-divulgação em Redes) por respeito a uma política de fluxo G se se verifica que $P \approx_{G,l} P$ para todos os níveis de segurança l.*

Para motivar a propriedade de segurança que acabámos de apresentar, vamos agora exibir alguns exemplos de programas inseguros, focando aspectos da declassificação e das fugas por migração, que são os temas mais importantes deste trabalho. Assumimos novamente os dois níveis de segurança H e L, e uma política global $G = \{L \prec H\}$.

Usando a declaração de fluxo É possível verificar que nenhum dos exemplos da Subsecção 3.1 satisfaz não-divulgação em redes. Por outro lado, os programas dos Exemplos 8 e 10 já são seguros. O programa

$$(b_L :=^? (\text{flow } H \prec L \text{ in } (?\ a_H))) \tag{15}$$

é essencialmente o mesmo que o Exemplo 8, sendo portanto seguro. No entanto, o programa

$$(\text{if } (?\ a_H) \text{ then } (\text{flow } H \prec L \text{ in } (b_L :=^? tt)) \text{ else } ()) \tag{16}$$

é inseguro, pois a declaração de fluxo não engloba a declassificação da referência a_H.

De notar que aqui, como estamos a lidar com confidencialidade (e não integridade), uma política de fluxo apenas afecta as capacidades de leitura de programas (e não as suas capacidades de escrita).

Exemplos de migrações inseguras Vimos intuitivamente porque é que o nível de segurança da informação acerca da posição de um processo numa rede é um minorante do nível das suas referências, e é representado pelo nível de segurança que está associado a cada (nome de) processo. Consequentemente, encontramos outras maneiras de exprimir programas inseguros, que devem igualmente ser rejeitados.

Revendo o Exemplo 12, podemos agora verificar que ele é de facto inseguro do ponto de vista da propriedade de não divulgação em redes. Outro exemplo parecido, mas mais

directo, mostra que a mera chegada de um processo e suas referências a um domínio pode desbloquear uma atribuição a uma referência baixa:

$$\begin{array}{c} d[(\text{if } a_H \text{ then } (\text{goto } d_1) \text{ else } (\text{goto } d_2))^{n_k}] \ \| \\ d_1[(n_k.y_L :=^? 1)^{m_{1j_1}}] \ \| \\ d_2[(n_k.y_L :=^? 2)^{m_{2j_2}}] \end{array} \tag{17}$$

O exemplo anterior mostra como da migração de um processo pode resultar uma fuga de informação de uma referência alta para uma referência baixa, via um processo "observador". É a habilidade do observador de detectar a presença do primeiro processo que origina a fuga. No entanto, devemos também prevenir um processo de revelar ele mesmo informação acerca da sua própria posição, como via uma atribuição baixa precedida de uma atribuição remota ou de uma leitura remota ou mesmo quando a atribuição baixa é por sua vez remota:

$$d[((n.u_\top :=^? 0); (b_L :=^? 0))^{m_H}] \tag{18}$$

$$d[((?\ n.u_\top); (b_L :=^? 0))^{m_H}] \tag{19}$$

$$d[(b_L :=^? 0)^{m_H}] \tag{20}$$

Propriedades da propriedade de segurança Como seria de esperar, a propriedade de não-divulgação em redes pode ser usada em contextos não-distribuídos. Para isso basta considerar uma rede com um único domínio. Indo mais além, a não-divulgação é equivalente à não-interferência para programas que não usam declassificação. Por outras palavras, a segurança baseada em não-divulgação em redes é *monotónica* em relação à não-divulgação simples, que por sua vez o é em relação à não-interferência clássica.

Por outro lado a introdução de declassificação em programas não transforma programas seguros em programas inseguros, como seria de esperar. Esta propriedade é conhecido por *coerência semântica* [33], segundo a qual programas semanticamente equivalentes devem ser classificados coerentemente como sendo seguros ou não.

5 O sistema de tipos e efeitos

Nesta secção apresentamos um sistema de tipos e efeitos que apenas aceita programas que satisfazem a propriedade de não-divulgação em redes. Começamos por definir e explicar a notação usada para exprimir juízos de tipo; de seguida comentamos as condições de tipificação, com ênfase nas aspectos mais inovadores, usando exemplos para mostrar a necessidade dessas condições; finalmente, concluimos com a apresentação de algumas propriedades do sistema de tipos, incluindo os teoremas da preservação de tipos e de correcção.

5.1 Um sistema de tipos com identificadores de processos

O sistema de tipos e efeitos que apresentamos aqui selecciona processos seguros assegurando a conformidade de todos os fluxos de informação com a política de fluxo que vigora em cada ponto do programa. O sistema aproxima construtivamente os *efeitos* de cada expressão, que inclui informação acerca dos níveis de segurança das referências de que pode depender a terminação ou não-terminação das computações.

Uma observação chave é que aqui a não-terminação de uma computação pode surgir de uma tentativa de aceder a recursos *remotos*. De modo a distinguir os processos a que pertencem cada expressão e referência, associamos identificadores $\check{m}, \check{n} \in \boldsymbol{N\check{a}m}$ a nomes

de processos já existentes, bem como ao nome de processo desconhecido '?' no caso dos processos que são criados durante a execução.

Deve ser claro que, pela mera posse de referências baixas, a posição de um processo pode ser detectada por "observadores baixos". Por esta razão, escolhemos o nível de segurança de um processo – que representa, na realidade, o seu nível de "visibilidade" – como sendo um minorante do nível das referências que lhe pertencem. Tal como veremos em breve, os níveis de segurança de processos são usados para reforçar os efeitos de segurança: o nível de escrita é actualizado por instruções de migração, enquanto que o nível de terminação é actualizado por instruções de acesso a um recurso remoto.

Os juízos de tipo Tal como está definido na Figura 6, os juízos do nosso sistema de tipos e efeitos têm a forma:

$$\Sigma, \Gamma \vdash^{\check{m}_j}_{G,F} M : s, \tau$$

O significado dos parâmetros é o seguinte:

- O ambiente de tipificação Γ atribui tipos a variáveis.
- A expressão M é um programa.
- O efeito de segurança s, é da forma $\langle s.r, s.w, s.t \rangle$, onde: o efeito $s.r$ é o *nível de leitura*, majorante dos níveis de segurança das referências que são lidas por M; o efeito $s.w$ é o *nível de escrita*, minorante dos níveis das referências que são escritas por M; e o efeito $s.t$ é o *nível de terminação*, majorante do nível das referências das quais pode depender a terminação da expressão M. De acordo com estas intuições, os níveis de leitura e de terminação são compostos de maneira co-variante, enquanto que o nível de escrita é contra-variante.
- O tipo τ é o tipo da expressão M. A sintaxe dos tipos foi dada na Figura 1. Inclui anotações que são usadas para determinar os efeitos da expressão a ser tipificada. Os efeitos são calculados com base no nível das referências que são acedidas e na política de fluxo do contexto. Como distinguimos entre referências locais e remotas, identificadores de processo e seus níveis de segurança aparecem nos tipos também. Expressões tipificáveis que reduzem a $()$ têm tipo unit, e os que reduzem a booleanos têm tipo bool; expressões tipificáveis que reduzem a uma referência pertencente a um processo m de nível j, e que aponta para um valor de tipo θ e tem nível de segurança l têm o tipo de referência $\theta\ \mathrm{ref}_{l,\check{m}_j}$ (os níveis de segurança l e j são usados para determinar os efeitos de expressões que lidam com referências); expressões que reduzem a uma função que aceita um parâmetro do tipo τ, que retorna uma expressão do tipo σ e que tem *efeito latente* s [20], sendo G e $\check{m}_j$ respectivamente a política de fluxo e o identificador de processo em que o corpo da função deverá ser tipificado, têm o tipo de função $\tau \xrightarrow[G,\check{m}_j]{s} \sigma$.
- A política de fluxo G é a política de fluxo global. Tal como já vimos, é um parâmetro dos pré-reticulados de segurança, da relação de fluxo e em particular as operações supremo e ínfimo. A relação de fluxo é usada no sistema de tipos para impor restrições nos fluxos de informação que são permitidos pelas regras de tipificação, e as operações supremo e ínfimo são usadas para construir os níveis de segurança de expressões no sistema de tipos.
- A política de fluxo F é a *política de fluxo do contexto*, aquela que é válida no contexto de tipificação de M, e contribui para o significado das operações e relações sobre níveis de segurança. Assume-se que contém a política de fluxo global, que é estendida com as políticas que são localmente declaradas pelo contexto de avaliação.

$$
\begin{array}{lrcl}
\textit{Ambiente de Nomeação de Processos} & \Sigma & \subseteq & ((\boldsymbol{Nam} \cup \{?\}) \times 2^{\boldsymbol{Pri}}) \times (\check{\boldsymbol{Nam}} \times 2^{\boldsymbol{Pri}}) \\
\textit{onde} & \Sigma \downarrow_{\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}} & : & (\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \to (\check{\boldsymbol{Nam}} \times 2^{\boldsymbol{Pri}}) \\
\textit{Ambiente de Tipificação} & \Gamma & : & \boldsymbol{Var} \to \boldsymbol{Typ} \\
\textit{Juízos de Tipo} & & := & \Sigma; \Gamma \vdash^{\check{m}_j}_{G,F} M : s, \tau
\end{array}
$$

Figura 6: Sintaxe dos juízos de tipo (ver também Figura 1)

$$
[\textsc{Nil}]\ \Sigma; \Gamma \vdash () : \mathsf{unit} \qquad [\textsc{Flow}]\ \frac{\Sigma; \Gamma \vdash^{\check{m}_j}_{F \cup F'} M : s, \tau}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (\text{flow } F' \text{ in } M) : s, \tau}
$$

$$
[\textsc{Abs}]\ \frac{\Sigma; \Gamma, x : \tau \vdash^{\check{m}_j}_{F} M : s, \sigma}{\Sigma; \Gamma \vdash (\lambda x.M) : \tau \xrightarrow[F, \check{m}_j]{s} \sigma} \qquad [\textsc{Rec}]\ \frac{\Sigma; \Gamma, x : \tau \vdash^{\check{m}_j}_{F} W : s, \tau}{\Sigma; \Gamma \vdash (\varrho x.W) : \tau}
$$

$$
[\textsc{BoolT}]\ \Sigma; \Gamma \vdash tt : \mathsf{bool} \qquad [\textsc{BoolF}]\ \Sigma; \Gamma \vdash ff : \mathsf{bool}
$$

$$
[\textsc{Var}]\ \Sigma; \Gamma, x : \tau \vdash x : \tau \qquad [\textsc{Loc}]\ \Sigma; \Gamma \vdash n_k.u_{l,\theta} : \theta\ \text{ref}_{l,\Sigma(n_k)}
$$

$$
[\textsc{Ref}]\ \frac{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : s, \theta \qquad \begin{array}{r} j \preceq\ \ l \\ s.r, s.t \preceq_F l \end{array}}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (\text{ref}_{l,\theta}\ M) : s \curlyvee \langle \top, l, \top \rangle, \theta\ \text{ref}_{l,\check{m}_j}}
$$

$$
[\textsc{Der}]\ \frac{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : s, \theta\ \text{ref}_{l,\check{n}_k}}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (?\ M) : s \curlyvee \langle l, \top, (\text{if } \check{m} \neq \check{n} \text{ then } j \curlyvee k \text{ else } \bot) \rangle, \theta}
$$

$$
[\textsc{Ass}]\ \frac{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : s, \theta\ \text{ref}_{l,\check{n}_k} \quad \Sigma; \Gamma \vdash^{\check{m}_j}_{F} N : s', \theta \qquad \begin{array}{r} s.t \preceq_F s'.w \\ s.r, s'.r, s.t, s'.t, j \preceq_F l \end{array}}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (M :=^{?} N) : s \curlyvee s' \curlyvee \langle \bot, l, (\text{if } \check{m} \neq \check{n} \text{ then } j \curlyvee k \text{ else } \bot) \rangle, \mathsf{unit}}
$$

$$
[\textsc{Cond}]\ \frac{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : s, \mathsf{bool} \qquad \begin{array}{l} \Sigma; \Gamma \vdash^{\check{m}_j}_{F} N_t : s_t, \tau \\ \Sigma; \Gamma \vdash^{\check{m}_j}_{F} N_f : s_f, \tau \end{array} \qquad s.r, s.t \preceq_F s_t.w, s_f.w}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (\text{if } M \text{ then } N_t \text{ else } N_f) : s \curlyvee s_t \curlyvee s_f \curlyvee \langle \bot, \top, s.r \rangle, \tau}
$$

$$
[\textsc{App}]\ \frac{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : s, \tau \xrightarrow[F, \check{m}_j]{s'} \sigma \quad \Sigma; \Gamma \vdash^{\check{m}_j}_{F} N : s'', \tau \qquad \begin{array}{r} s.t \preceq_F s''.w \\ s.r, s''.r, s.t, s''.t \preceq_F s'.w \end{array}}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (M\ N) : s \curlyvee s' \curlyvee s'' \curlyvee \langle \bot, \top, s.r \curlyvee s''.r \rangle, \sigma}
$$

$$
[\textsc{Seq}]\ \frac{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : s, \tau \quad \Sigma; \Gamma \vdash^{\check{m}_j}_{F} N : s', \sigma \quad s.t \preceq_F s'.w}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (M; N) : s \curlyvee s', \sigma}
$$

$$
[\textsc{Thr}]\ \frac{j \preceq_F k \quad \check{n}\ \textit{novo em}\ \Sigma \quad \Sigma, ?_k : \check{n}_k; \Gamma \vdash^{\check{n}_k}_{\emptyset} M : s, \mathsf{unit}}{\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (\text{thread}_l\ M) : \langle \bot, s.w, \bot \rangle, \mathsf{unit}}
$$

$$
[\textsc{Mig}]\ \Sigma; \Gamma \vdash^{\check{m}_j}_{F} (\text{goto } d) : \langle \bot, j, \bot \rangle, \mathsf{unit}
$$

Figura 7: Sistema de tipos e efeitos

- O *ambiente de nomeação de processos* Σ é uma relação binária entre nomes de processo decorados, estendido com '$?_k$' (onde '?' representa o nome de processo desconhecido), e o conjunto de identificadores de processo decorados. Definimos $\mathrm{dom}(\Sigma)$ como $\{n_k \mid \exists \check{n}_k \,.\, (n_k, \check{n}_k) \in \Sigma\}$. Assume-se que a restrição de Σ ao domínio $\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}$ (escrito $\Sigma \downarrow_{\boldsymbol{Nam}\times 2^{\boldsymbol{Pri}}}$) é uma função, em que todos os nomes de processo n são distintos. Os únicos identificadores que são imagens dos nomes de processo são aqueles que correspondem a processos que já criaram uma referência – e cujo nome é o prefixo desse endereço. Os outros estão relacionados com o nome '$?_k$', para algum nível l, correspondente ao caso dos processo que são criados durante a execução.
- O identificador de processo $\check{m}_j$ nomeia o processo a que pertence a expressão M.
- O nível de segurança j representa um minorante das referências que o processo possui e cria. É o nível de segurança que é atribuído a cada processo quando é criado.

Em algumas das regras de tipificação usamos o ínfimo entre dois efeitos de segurança:

Definição 5.1. $s \curlyvee_G s' \quad \overset{def}{\Leftrightarrow} \quad (s.r \curlyvee_G s'.r, s.w \curlywedge_G s'.w, s.t \curlyvee_G s'.t)$

O sistema de tipos e efeitos é dada na Figura 7. Usamos as seguintes abreviações: escrevemos a relação de fluxo por respeito à política global de fluxo como $\preceq$, supremo $\curlywedge$ e ínfimo $\curlyvee$, em vez de $\preceq_G$, $\curlywedge_G$ e $\curlyvee_G$, respectivamente; também omitimos a política global de fluxo que aparece nos índices de $\vdash^{\check{m}_j}_{G,F}$ escrevendo apenas $\vdash^{\check{m}_j}_{F}$; sempre que temos $\forall F, \check{m}_j \,.\, \Sigma; \Gamma \vdash^{\check{m}_j}_{F} M : \langle \bot, \top, \bot \rangle, \tau$ escrevemos apenas $\Sigma; \Gamma \vdash M : \tau$.

5.2 Condições de tipificação

Veremos agora como é que o sistema de tipos selecciona apenas programas seguros, de acordo com a propriedade de não-divulgação que foi definida na secção anterior, focando as condições de tipificação que controlam a migração e a declassificação.

Na regra Loc, como o nome do processo a que pertence a referência aparece no prefixo do seu nome, o identificador de processo correspondente pode ser encontrado usando Σ. Na regra Ref, a referência que é criada pertence ao processo que é identificado pelo índice do '$\vdash$'. Verificamos que o nível de segurança que é declarado para as referências novas é maior do que o nível de segurança do processo.

O corpo de uma função (regra Abs) é executado pelo processo que a aplica a um argumento (ver App), no mesmo contexto de fluxo da aplicação. É por isso que o identificador de processo e o contexto de fluxo da sua execução são latentes.

Na regra Thr, um identificador novo – imagem de um nome de processo desconhecido representado por '?' – é usado para tipificar o processo que é criado. Quando um processo novo é gerado durante a execução por outro processo também novo, o domínio de Σ que é usado para tipificar a criação de processos imbricados contém mais do que uma entrada usando '?'. O valor de '?' não é re-escrito aquando da tipificação de criação de processos imbricados pois devemos guardar informação completa acerca das imagens de Σ, de maneira a garantir que os novos identificadores de processo que são atribuídos pela regra Thr são novos. Veremos em breve que estes são usados basicamente para distinguir acessos a referências locais de acessos a referências remotas externas (que são potencialmente remotas).

Fugas por migração A suspensão de um processo num acesso a uma referência que não está presente pode ser vista como uma computação divergente (que não termina) que

pode ser desbloqueada por migração de processos concorrentes. Assim, podemos lidar com as fugas por migração de forma semelhante à que utilizamos para as fugas por divergência.

Vimos que as fugas por divergência aparecem quando uma mudança na parte baixa do estado dependente da terminação de uma computação que a precede, que por sua vez depende de informação de nível alto. O Exemplo 12 mostra como a informação alta pode escapar devido a um processo (n), diferente daquele que efectua a atribuição baixa (m_1 e m_2), que está noutro domínio. O ponto chave neste exemplo é que a sincronização entre dois processos é feita via migração de n para um domínio onde m_1 ou m_2 estão localizados, numa altura em que as atribuições baixas que deverão ocorrer em m_1 e m_2 estão bloqueadas pela suspensão de um acesso a uma das referências pertencentes a n.

Do ponto de vista de n, não é possível saber quando, nos domínios para onde n pode migrar, haverá outros processos suspensos à espera da sua chegada. Mas, desde que todos os outros processos sejam tipificáveis, pode se garantir que (ver abaixo) que as atribuições suspensas não são de nível inferior do que os acessos que estão a causar a suspensão (i.e. $k \preceq L$). Então, assumindo o pior caso para n, o efeito de escrita causado pela migração é incrementado com o nível de segurança de n, que é por sua vez um minorante k do nível de segurança de todas as suas referências. Note que como consequência, a regra Cond rejeita processos n de uma maneira usual, já que $H \not\preceq L$. Na regra Mig, ao adicionar o nível de segurança do processo ao efeito de escrita da instrução de migração, prevenimos que a migração de processos que possuem referências baixas dependa de informação alta.

Do ponto de vista de m_1 e m_2, não é possível saber se a chegada de um processo (n), que vai desbloquear a sua execução, depende de informação alta ou não. Mas, desde que n seja tipificável, podemos assegurar (como acima) que o nível da informação de que depende a migração de n é inferior ou igual ao próprio nível de n (i.e. $H \preceq k$). Por isso, nas regras Der e Ass, o efeito de terminação é actualizado com o nível do processo que é dono da referência remota que se quer aceder.

Nos Exemplos 18 e 19, a atribuição baixa pode apenas ocorrer se os processos m e n estiverem localizados no mesmo domínio. Por isso, também a posição de m pode ser revelada quando ocorre a atribuição baixa. Isto explica por que razão se actualiza o nível de terminação da atribuição (Ass) e leitura (Der) também com o nível de m.

Ref. A condição $j \preceq k$ assegura que as referências que são criadas por um processo respeitam o nível de segurança do processo, i.e. que não são inferiores (por respeito à política de fluxo global).

Ass. O programa inseguro do Exemplo 20 é rejeitado pela condição $j \preceq_G l$ da regra Ass. De forma semelhante, o programa do Exemplo 17 é rejeitado se j_1 ou $j_2 \not\preceq L$, de modo a prevenir a revelação de informação sobre as posições de m_1 e m_2. Note que, na regra de tipificação, para os mesmos casos em que $m = n$ a condição é satisfeita de qualquer modo graças ao significado de k. Existe uma condição implícita, $k \preceq l$, que é satisfeita por hipótese, já que o processo n é dono da referência $n.y_L$.

Thr. A condição $j \preceq_F l$ rejeita o seguinte programa inseguro $d[(\text{thread}_L\ M)^{m_H}]$. A razão pela qual este programa é considerado inseguro é que a presença do processo alto m, que só deveria ser "visível" ao nível H, é indicada ao nível L, nível em que o processo criado é visível.

Condições de tipificação relaxadas Observemos agora os aspectos do sistema de tipos que permitem a aceitação de programas por serem seguros do ponto de vista da não-divulgação, mas que seriam considerado inseguros do ponto de vista da não-interferência. Concentramo-nos assim no papel da política de fluxo que aparece como parâmetro dos juízos de tipo.

Para tipificar a declaração de fluxo (flow F in M), apenas é exigido da expressão M que seja tipificável no contexto da política de fluxo actual, *estendida com* F. A tipificabilidade por respeito a políticas de fluxo "maiores" é "mais fácil", pois recordemos que as condições que são impostas nas regras de tipificação na prática restringem o fluxo de informação que é codificado pelas expressões, de modo a serem compatíveis com a relação de fluxo actual. Como as relações de fluxo que são parametrizadas com políticas de fluxo mais permissivas são mais fracas, então mais fluxos satisfazem as restrições por elas impostas.

A forma mais simples de operação de declassificação é o do Exemplo 8, que autoriza um fluxo directo de informação que de outra maneira seria considerado inseguro. De facto, para tipificar esse programa usando as regras FLOW, ASS e DER, é suficiente que a condição $\{H\} \preceq_{H \prec L} \{L\}$ se verifique, o que é claramente o caso.

Já mencionámos que é inevitável que alguns programas seguros sejam rejeitados pelo sistema de tipos. Por exemplo, uma condição que testa uma referência alta nunca pode ser aceite logo que algum dos seus ramos M ou N contenha uma atribuição ou criação de uma referência baixa. Enquanto que esta restrição rejeita todas as possíveis fugas por controlo que poderiam ser codificadas pela condição, rejeita também muitos outros programas que inofensivamente modificam a parte baixa da memória num dos ramos (ver [1] em como assegurar, para uma linguagem simples, que tais ramos não causem problemas). Uma aplicação prática da declaração de fluxo poderia ser a de se escrever, em vez

$$(\text{flow } H \prec L \text{ in } (\text{if } (?\ a_H) \text{ then } M \text{ else } N)) \tag{21}$$

neutralizando assim o efeito das restrições de tipificação que são impostas por COND.

Como último exemplo, mostramos que o nível para o qual um programa vai declassificar o conteúdo de uma referência não pode ser antecipado estaticamente. Seja M a expressão:

$$(\text{if } N \text{ then } (\text{flow } p \prec q \text{ in } (a_{q,\theta} :=^{?} (?\ c_{p,\theta}))) \text{ else } (\text{flow } p \prec r \text{ in } (b_{r,\theta} :=^{?} (?\ c_{p,\theta})))) \tag{22}$$

Então é possível ver que M é tipificável se a condição $s.r \preceq_G \{q, r\}$ se verificar, onde $s.r$ é o efeito de leitura do booleano N. Como não existe nenhuma restrição referente ao nível de confidencialidade p da referência c, o nível q ou r para onde o conteúdo de p será declassificado depende apenas do valor resultante da computação de N.

5.3 Propriedades de expressões tipificáveis

O primeiro resultado que vamos enunciar estipula que a computação preserva os tipos dos processos, e que à medida que os efeitos de uma expressão são efectuados, os efeitos de segurança do processo que a contém "enfraquecem".

Teorema 5.2 (Preservação de tipos)**.**
Se para alguns $\Sigma, \Gamma, s, \tau, F, m_j$ *se tiver que* $\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} M : s, \tau$ *e* $\langle M^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle M'^{m_j}, T', S'\rangle$ *onde todos os* $a_{l,\theta} \in \mathrm{dom}(S)$ *satisfazem* $\Sigma; \Gamma \vdash S(a_{l,\theta}) : \theta$*, então* $\exists s'$ *tal que* $\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} M' : s', \tau$*, onde* $s'.r \preceq s.r$*,* $s.w \preceq s'.w$ *e* $s'.t \preceq s.t$*. Além disso, tem-se que* $\exists \check{n}, s''$ *tais que* $\Sigma, ?_k : \check{n}_k; \Gamma \vdash_\emptyset^{\check{n}_k} N : s'', \mathsf{unit}$ *onde* $\check{n}$ *é novo em* Σ*, e* $s.w \preceq s''.w$*.*

O resultado principal deste trabalho consiste na correcção do sistema de tipos, isto é, a garantia de que redes tipificáveis são de facto seguras no sentido da propriedade de não-divulgação em redes. Mais precisamente, há que provar que, sob qualquer política de fluxo global G, se todos os processos de uma rede (conjunto P) forem tipificáveis usando o sistema de tipos da Figura 7, então satisfazem a propriedade de não-divulgação para redes que é dada na Definição 4.5.

Teorema 5.3 (Correcção para não-divulgação em redes)**.**
Considere um grupo de processos P e uma política de fluxo global G. Se para todo $M^{m_j} \in P$ existirem Σ, Γ, s e τ tais que $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{G,G} M : s,\tau$, então P satisfaz a propriedade de Não-divulgação em Redes por respeito a G.

A prova do resultado de correcção, que dada a sua extensão e complexidade é omitida aqui, poderá ser encontrada na tese em que foi baseado este trabalho. Muito brevemente, as etapas seguidas foram: 1. Desenhar uma relação binária $\mathcal{T}$ entre processos que são tipificáveis com um nível de terminação baixo, e que têm o mesmo comportamento no que respeita a terminação e as alterações efectuadas sobre estados com partes baixas iguais; 2. Desenhar uma relação binária "maior" $\mathcal{R}$ entre processos tipificáveis que ou não alteram a parte baixa das memórias, ou então têm o mesmo comportamento no que respeita a terminação e as alterações efectuadas sobre estados com partes baixas iguais; 3. Desenhar uma bissimulação-(G,l) entre processos que ou não alteram a parte baixa das memórias, ou então estão relacionados por $\mathcal{R}$.

De notar que o resultado acima é composicional, no sentido em que é suficiente verificar a tipificabilidade de cada processo separadamente de modo a assegurar não-divulgação para a rede toda. A política de fluxo G pode ser tomada como sendo a "intersecção" das políticas de fluxo de todos os processos da rede, podendo ser convenientemente aproximado pela política de fluxo vazia, que fornece um pré-reticulado de segurança mínimo que todos os processos devem satisfazer.

6 Conclusão

Para concluir, resumimos as principais contribuições técnicas deste trabalho, numa perspectiva de comparação com trabalho relacionado, e tecemos alguns comentários finais.

6.1 Contribuições principais e trabalho relacionado

Propriedades de segurança Um trabalho recente [33] contém um apanhado exaustivo da literatura sobre o tópico da declassificação. Em particular, é observado que a declassificação pode ser controlada de acordo com quatro objectivos ortogonais, que *restringem* quando, por quem, qual e onde é que informação pode ser divulgada. Seguindo a argumentação da Subsecção 3.2, o nosso trabalho relaciona-se mais com as formas de *permitir* a divulgação. Nesta perspectiva, os trabalhos mais próximos do nosso são os de Sands et al. [22, 5]. No primeiro, é abordado o problema do *onde* é que a declassificação é permitida, restringindo-a a ocorrer em pontos muito precisos dos programas, entre níveis de segurança que são estaticamente fixos. A linguagem considerada é uma simples linguagem "while" com criação de processos, mas sem criação de referências, funções de ordem-superior; o sistema de tipos é também mais restritivo, sem refinamento de efeitos de segurança. Já em [5], são introduzidos os *flow locks*, que são trincos que abrem ou fecham acessos a posições de memória, já podendo ser manipulados dinamicamente. Esta proposta é deveras expressiva, podendo codificar por exemplo as nossas declarações de fluxo, embora não providencie resposta para o problema do uso da declassificação em cenários concorrentes.

No nosso trabalho, estudámos uma nova política de segurança que chamámos não-divulgação, que determina a ausência de fluxos de informação inseguros graças a uma ordem dinâmica sobre níveis de segurança. Mostrámos as vantagens da nossa propriedade por comparação com a propriedade de não-interferência clássica, que implica a total ausência de fugas de informação.

Definiu-se a propriedade de não-divulgação em termos de uma bissimulação, baseada naturalmente sobre transições passo único. Isto oferece a precisão necessária para, por um lado analisar as mudanças que ocorrem nos estados a cada passo de execução (análise necessária num ambiente concorrente), e por outro lado indicar a política de fluxo válida (necessário para restringir o âmbito das declarações de fluxo) etiquetando a semântica de transições. Mostrámos ainda que as ordens entre níveis de segurança podem ser expressas directamente como relações entre entidades.

Pensamos que a não-divulgação é uma generalização natural da não-interferência clássica, e que a ideia de utilizar bissimulações sobre semânticas de passo único para definir uma política de segurança que reflecte a natureza *local* da declassificação poderia ser utilizada noutros quadros.

Modelo de computação Um primeiro passo em relação ao estudo da confidencialidade em sistemas distribuídos foi o seu estudo sobre linguagens concorrentes. Podemos encontrar uma linha de trabalhos, elaborados sobre linguagens incrementalmente expressivas, de que [36, 35, 4, 17] são exemplos, em que o tratamento de fugas por divergência é uma questão central. Num contexto distribuído, mas ainda sem mobilidade de código, [21, 29] providenciaram um sistema de tipos para preservar confidencialidade sobre tipos diferentes de canais que são publicamente observáveis. Noutra perspectiva, em [43] considerou-se um sistema de domínios potencialmente corruptos, e uma maneira de partir e distribuir partes de programas por estes domínios de acordo com níveis de confiança atribuídos. Noutra linha de estudo, consideraram-se cálculos concorrentes puramente funcionais com mobilidade de canais, donde mencionamos [16, 15], em segurança de fluxos de informação para o cálculo pi. O primeiro estudo da não-interferência para uma linguagem com distribuição (hierarquizada) e mobilidade de código parece ter sido [9]. O trabalho foi feito sobre uma versão dos "Mobile Ambients" [6]. Neste modelo, que não inclui, uma noção de estado nem de declassificação, a computação ocorre por mobilidade e comunicação entre canais de processos cuja posição na rede é também considerada como informação confidencial.

A linguagem em que baseámos o nosso estudo é simples mas expressiva. O nosso ponto de partida foi um cálculo lambda de ordem superior com criação de processos e de referências. Este núcleo foi enriquecido com uma nova declaração de fluxo que permite parametrizar dinamicamente a relação de ordem entre os níveis de segurança. Por fim, uma noção de domínio de execução, em combinação com uma instrução de migração de processos e de referências deu à nossa linguagem os ingredientes necessários para estudar a distribuição e mobilidade.

Orientámos o nosso estudo na direcção do paradigma de computação distribuída com migração de processos, onde estes são executados em domínios diferentes, e onde a localização relativa de processos e de recursos determina as circunstâncias em que os programas são executados. O paradigma de computação local em que processos são criados dinamicamente e executados sobre uma plataforma de execução única corresponde a um caso particular do nosso estudo.

Descobrimos novas formas de fugas de informação, as fugas por migração, que podem ser codificadas em contextos semelhantes ao nosso. Estas fugas de informação parecem ter semelhanças com fugas de informação causadas pela localização de processos em redes to tipo "Ambient", o que suporta a conjectura de que os nossos resultados não se confinam ao nosso modelo particular. Escolhemos um modelo de mobilidade propositadamente simples, que é no entanto suficiente para apresentar os princípios subjacentes às fugas por migração. No entanto, pode-se esperar que modelos mais complexos de computação global teriam efeitos interessantes sobre o estudo do controlo de fluxos de informação.

Mecanismos de certificação É possível encontrar um progressivo refinamento na prevenção de fugas por divergência em linguagens concorrentes por meio de sistemas de tipos. Na verdade, os primeiros sistemas de tipos apresentados para este efeito [38, 36] eliminam a ocorrência deste tipo de fugas pela severa rejeição de testes sobre predicados de nível alto (o que corresponderia no nosso caso a restringir o efeito de leitura de condições ao nível $\perp$), o que mostra a vantagem em considerar outros efeitos como o de leitura [35, 4], e agora o de terminação. Mais recentemente, encontramos um estudo [3] que antecipa a necessidade de actualizar o nível de terminação de uma condição, de acordo com as características dos seus ramos.

Para aplicar a política de segurança sobre os programas da nossa linguagem, apresentámos um novo sistema de tipos e de efeitos. A prova de correcção do sistema de tipos segue o modelo apresentado detalhadamente na tese em que se baseia este trabalho, onde se usou o mesmo método para dois outros pares "propriedade de segurança – sistema de tipos e efeitos". Temos assim boas razões para crer que o mecanismo de prova da correcção do sistema de tipos, pode também ser aplicada a outros casos.

6.2 Notas finais

Para finalizar, gostaríamos de tecer alguns comentários acerca de dois aspectos fundamentais dos principais pontos aqui introduzidos.

Apenas mais um mecanismo de declassificação? Como visto na Subsecção 6.1, propostas de mecanismos para declassificação abundam na literatura. Mais do que propor um outro mecanismo de declassificação, sugerimos uma maneira de encarar a dificuldade de determinar qual a natureza que um mecanismo de declassificação deve ter, e qual o tipo de garantias de segurança que ele deveria fornecer [41]. A ideia chave é que, antes de reflectir sobre como controlar o uso da declassificação, seria bom dispor de um enquadramento teórico para exprimi-la. Pensamos que o enquadramento teórico para a declassificação que propusemos neste trabalho é atraente pelas seguintes razões:

- Fornece um mecanismo simples, mas flexível e poderoso para a declassificação. Em particular, não comporta restrições que ultrapassem os objectivos da declassificação.
- Inclui uma política de segurança, a não-divulgação, que possui propriedades semânticas muito satisfatórias. Notamos que estas propriedades são sugeridas como sendo "boas propriedades", a exigir de uma correcta política de segurança [33], como a *coerência semântica*, e a *monotonicidade da segurança* (ver Subsecção 4.2).
- É facilmente generalizável a outras linguagens e ambientes. Em particular, a não-divulgação é uma propriedade *extencional*, isto é ela é definida em termos de semântica de programas, e é independente das propriedades da linguagem.
- Fornece uma técnica fiável para rejeitar com precisão razoável todos os programas que não respeitam a política de segurança.

A primeira das qualidades acima enumeradas é talvez a mais importante. De facto, as declarações de fluxo podem exprimir a declassificação com qualquer grau de precisão, desde operações específicas a porções inteiras de programas, entre quaisquer níveis de segurança. Isto é atingido pela manipulação directa das políticas de fluxo, que são simples relações binárias entre entidades de um sistema.

Finalmente, notamos que, ao incorporar este novo mecanismo de declassificação no nosso estudo sobre segurança de fluxos de informação em redes, mostrámos a sua robustez quando usado em cenários computacionais novos.

Sobre a combinação de declassificação e mobilidade O estudo do impacto da declassificação e da mobilidade em fluxos de informação são temas bastante independentes. Não será por isso surpreendente que eles possam ser combinados de maneira tão natural. No entanto, devemos sublinhar que esta facilidade advém da natureza altamente descentralizada das declarações de fluxo. De facto, não é pressuposto nenhum acordo global sobre as políticas de declassificação (como é feito em [22]). Além disso, as mudanças que são efectuadas dinamicamente sobre as políticas de fluxo têm um âmbito local, não afectando por isso a totalidade do sistema.

É também interessante notar que a combinação da declassificação e da mobilidade poderá ser particularmente relevante em áreas em que a mobilidade e a segurança de informação são indissociáveis. Podemos por exemplo apontar a área da computação "grid", em que se aproveita o potencial oferecido pela computação paralela, separando programas em sub-processos relativamente independentes, que são distribuídos, quer por redes de processadores locais, como por redes de recursos fisicamente dispersos. Neste contexto, colocam-se dois problemas simétricos: por um lado, o da segurança da informação que está envolvida nas computações dos processos "hóspedes" que migram para plataformas externas; por outro, o da segurança da informação que é manipulada pelos programas nativos que correm nas plataformas "hospedeiras".

Perante a impossibilidade de garantir na prática propriedades tão restritivas como a não-interferência, poder-se-á optar por situações de compromisso. Assim, argumentando que a revelação de determinadas porções de informação por cada parte do programa não é por si só significativa, poder-se á então recorrer ao uso de declarações de fluxo, sendo então mais comportável garantir a propriedade mais flexível da não-divulgação, na prática.

Referências

[1] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2000.

[2] G. R. Andrews e R. P. Reitman. An axiomatic approach to information flow in programs. *ACM Transactions on Programming Languages and Systems*, 2(1):56–76, 1980.

[3] G. Boudol. On typing information flow. In *International Colloquium on Theoretical Aspects of Computing*, v. 3722 de *LNCS*. Springer-Verlag, 2005.

[4] G. Boudol e I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281(1):109–130, 2002.

[5] N. Broberg e D. Sands. Flow locks: towards a core calculus for dynamic flow policies. In *15th European Symposium on Programming*, v. 3924 de *LNCS*. Springer Verlag, 2006.

[6] L. Cardelli e A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[7] S. Chong e A. C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security*. ACM Press, 2004.

[8] E. Cohen. Information transmission in computational systems. In *Proceedings of the sixth ACM symposium on Operating systems principles*. ACM Press, 1977.

[9] S. Crafa, M. Bugliesi, e G. Castagna. Information flow security for boxed ambients. In *International Workshop on Foundations of Wide Area Network Computing*, v. 66(63) de Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers, 2002.

[10] K. Crary, A. Kliger, e F. Pfenning. A monadic analysis of information flow security with mutable state. *Journal of Functional Programming*, 15(02), 2005.

[11] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[12] R. Focardi e R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[13] J. A. Goguen e J. Meseguer. Security policies and security models. In *Proceedings of the 1982 Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, 1982.

[14] N. Heintze e J. G. Riecke. The slam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM Symposium on Principles of programming languages*. ACM Press, 1998.

[15] M. Hennessy e J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5):566–591, 2002.
[16] K. Honda, V. Vasconcelos, e N. Yoshida. Secure information flow as typed process behaviour. In *9th European Symposium on Programming*, v. 1782 de *LNCS*. Springer-Verlag, 2000.
[17] K. Honda e N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of the 29th ACM Symposium on Principles of programming languages*. ACM Press, 2002.
[18] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.
[19] P. Li e S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2005.
[20] J. M. Lucassen e D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1988.
[21] H. Mantel e A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 11(4):615–676, 2004.
[22] H. Mantel e D. Sands. Controlled declassification based on intransitive noninterference. In *Programming Languages and Systems: 2nd Asian Symposium*, v. 3302 de *LNCS*. Springer-Verlag, 2004.
[23] R. Milner, M. Tofte, R. Harper, e D. MacQueen. *The definition of Standard ML (Revised)*. The MIT Press, 1997.
[24] A. Myers, A. Sabelfeld, e S. Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2004.
[25] A. C. Myers e B. Liskov. A decentralized model for information flow control. In *Proceedings of the sixteenth ACM symposium on Operating systems principles SOSP'97*. ACM Press, 1997.
[26] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of programming languages*. ACM Press, 1999.
[27] F. Pottier e V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, 2003.
[28] P. Ryan, J. McLean, J. Millen, e V. Gligor. Non-interference: who needs it? In *Proceedings of the 14th Workshop on Computer Security Foundations*. IEEE Computer Society, 2001.
[29] A. Sabelfeld e H. Mantel. Static confidentiality enforcement for distributed programs. In *Static Analysis : 9th International Symposium*, v. 2477 de *LNCS*. Springer-Verlag, 2002.
[30] A. Sabelfeld e A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
[31] A. Sabelfeld e A. Myers. A model for delimited information release. In *International Symposium on Software Security (ISSS'03)*, v. 3233 de *LNCS*. Springer-Verlag, 2004.
[32] A. Sabelfeld e D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2000.
[33] A. Sabelfeld e D. Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2005.
[34] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), 2003.
[35] G. Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2001.
[36] G. Smith e D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM Symposium on Principles of programming languages*. ACM Press, 1998.
[37] D. Volpano. Secure introduction of one-way functions. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2000.
[38] D. Volpano e G. Smith. Eliminating covert flows with minimum typings. In *Proceedings of the 10th Computer Security Foundations Workshop*. IEEE Computer Society, 1997.
[39] D. Volpano e G. Smith. Verifying secrets and relative secrecy. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2000.
[40] D. Volpano, G. Smith, e C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
[41] S. Zdancewic. Challenges for information-flow security. In *1st International Workshop on the Programming Language Interference and Dependence*, 2004.
[42] S. Zdancewic e A. C. Myers. Secure information flow via linear continuations. *Higher Order Symbol. Comput.*, 15(2-3):209–234, 2002.
[43] S. Zdancewic, L. Zheng, N. Nystrom, e A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, 2002.
[44] Silvano Dal Zilio. Mobile processes: A commented bibliography. *Lecture Notes in Computer Science*, 2067, 2001.

Nota: Foram omitidas auto-referências, para preservação da anonimato.