Math. Struct. in Comp. Science (2011), *vol.* 21, *pp.* 1111–1181. © Cambridge University Press 2011
doi:10.1017/S096012951100017X

Non-disclosure for distributed mobile code

ANA ALMEIDA MATOS[†] and JAN CEDERQUIST[‡]

SQIG-Instituto de Telecomunicações
and
Instituto Superior Técnico de Lisboa, Portugal
Email: {ana.matos;jan.cederquist}@ist.utl.pt

Received 2 June 2008; revised 23 March 2010

With the emergence of the new possibilities offered by global computing, new security issues follow from the fact that these possibilities can be equally exploited by parties with malicious intentions. Many attacks arise at the application level, and can be tackled by means of programming language techniques. For instance, confidentiality can be violated during the execution of programs that reveal secret information. This kind of program behaviour can be avoided by information flow analyses that detect the encoding of illegal flows.
This paper studies information flows that occur in distributed programs with code mobility from a language-based security perspective. New forms of security leaks that are introduced by code mobility, which we call *migration leaks*, are presented and compared with well-known forms of illegal flow. We propose an information flow property that is adequate for networks consisting of a generalisation of the non-disclosure policy. We design a type and effect system for enforcing it on an expressive distributed calculus, and explain a soundness proof methodology in detail.

1. Introduction

Protecting the confidentiality of data is a concern of particular relevance in a global computing context. When information and programs move throughout networks they become exposed to users with different interests and responsibilities. This motivates the search for practical mechanisms that enforce respect for the confidentiality of information, while minimising the need to rely on mutual trust. Access control deals with an important part of the problem, but not all of it, since it does not deal with how information may flow between the different parts of a system. Surprisingly, very little research has been done on the control of information flow in explicitly distributed networks. In fact, to the best of our knowledge, this work is the first to address the problem in an imperative setting where the mobility of resources plays an explicit role.

This paper is concerned with ensuring confidentiality in networks. More specifically, it is about controlling information flows between subjects that have been given different security clearances, in the context of a distributed setting with code mobility. In such a setting, one cannot assume that resources are accessible to all programs at all times. In

[†] Ana Almeida Matos was partially funded by the Ph.D. scholarship POSI/SFRH/BD/7100/2001 and by the French ACI Project CRISS, and at a later stage by FCT project via KLog PTDC/MAT/68723/2006.
[‡] Jan Cederquist was partially funded by FCT project via KLog PTDC/MAT/68723/2006.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

fact, a network can be viewed as a collection of sites where the conditions for computation to occur are not guaranteed by one site alone. We are led to ask whether failures can be exploited as covert information flow channels. The answer is *Yes*. New security leaks, which we call *migration leaks*, arise from the fact that the execution or suspension of programs now depends on the position of resources over the network, which may in turn depend on secret information.

We take a language based approach (Sabelfeld and Myers 2003), and thus restrict our investigation to information leaks occurring within computations of programs of a given language. These can be statically prevented by means of a type and effect system (Volpano *et al.* 1996; Lucassen and Gifford 1988) by rejecting insecure programs before execution. As is standard, we attribute security levels to the objects of our language (memory addresses), and organise them into a lattice (Denning 1976). Since confidentiality is the issue, these levels indicate to which subjects the contents of an object are allowed to be disclosed. Consequently, during computation, information contained in objects of a 'high' security level (we will call them 'high objects') should never influence objects of lower or incomparable level. This policy has been widely studied and is commonly referred to as *non-interference* (Goguen and Meseguer 1982). In a more general setting, where the security lattice may vary within a program, *non-disclosure* (Almeida Matos and Boudol 2009) can be used instead, thus requiring that each step performed by a thread complies with its current flow policy.

We consider a calculus for mobility where the notion of the location of a program or resource has an impact on computations:

— resources and programs are distributed over computation sites – or *domains* – and can change position during execution;

— a resource can only be accessed by a program that is located at the same site;

— remote accesses (that is, attempts to access resources that are currently not local) are suspended until the resources become available.

The language of local computations is an imperative higher-order λ-calculus with concurrent threads, to which we add a standard migration primitive. We include a flow declaration construct (Almeida Matos and Boudol 2009) giving the programmer the ability to declare local flow policies for concurrent threads, allowing, in particular, the *declassification* of information, that is, explicitly allowing certain information leaks to occur in a controlled way (see Sabelfeld and Sands (2005) for an overview). We show that mobility and declassification can be safely combined provided migrating threads compute according to declared flow policies.

Current security properties were designed for local computations where the notion of locality does not play a crucial role and are not suitable for treating information flows in a distributed setting with code mobility. In fact, since the location of resources in a network can itself be a source of information leaks, the notion of a safe program must take this into account. We therefore propose a new security property, *non-disclosure for networks*, that can detect migration leaks on states that track the positions of programs in a network. Working very much in the spirit of non-disclosure, security insurances consider

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

distributed local flow policies, which are a crucial aspect of the decentralised nature of networks.

1.1. *Contributions*

This paper is based on the conference article Almeida Matos (2005), as well as on the first author's Ph.D. thesis (Almeida Matos 2006). The main contributions are:

— The formalisation of a flexible information flow policy, which directly generalises non-interference and non-disclosure (Almeida Matos and Boudol 2009) to distributed settings with code mobility. We call this property *non-disclosure for networks*.
— The identification of a new kind of security leak, which we call *migration leaks*, that are specific to distributed settings with code mobility. The belief that such leaks are not limited to the particular language presented here is supported by a short discussion about a study of non-interference for Boxed Ambients (Crafa *et al.* 2002).
— The presentation of a new type and effect system for enforcing this property. Besides extending the one in (Almeida Matos and Boudol 2009) to a distributed setting, when stripped of the conditions and parameters that tackle the networking issues, it provides an alternative way of enforcing non-disclosure in general by restricting information leaks so that they only occur within the boundaries of the flow declarations.
— A detailed presentation of the soundness proof supported by remarks that give a *rationale* for the most important steps. The aim is to provide sufficient explanations to enable the same proof mechanism to be used in other settings.

Outline of the paper

The paper is organised as follows. In the next section we provide intuitions for the main questions that we attempt to answer in this paper. In Section 3 we define a distributed calculus with code and resource mobility. In Section 4 we formulate a non-disclosure property that is suitable for a decentralised setting. In Section 5 we develop a type and effect system that only accepts programs satisfying such a property. Finally, we comment on related work in Section 6 and present our conclusions.

2. Motivation

The main question that we aim to answer in this paper is *Why does code mobility pose new challenges regarding the security of information flow?* We provide some intuitions in this section, and discuss briefly what the desirable information flow property to be enforced should be, as well as how declassification can fit into the picture.

2.1. *Information leaks*

We begin by considering a sequential imperative language, such as the one in Volpano *et al.* (1996), where each variable is assigned one of two security levels, *low* (public, L) and *high* (secret, H), meaning that the information they refer to can only be read by subjects

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

with the corresponding security clearance. The variables' security levels are specified using subscripts. Some programs in our language are said to set up *insecure flows* of information, or *interferences*, during their execution. The reason they are considered insecure is that the initial values of high variables may influence the final value of low variables, thus breaking the intended meaning of the security levels.

The simplest case of insecure flow occurs in an assignment of the value of a high variable to a low variable, as in $b_L := a_H$, which encodes a *direct leak* of information. More subtle leaks may be induced by the choice of control paths (*control leaks*), as in the program

$$\text{if } a_H = tt \text{ then } b_L := tt \text{ else } b_L := ff \tag{1}$$

where at the end of execution the value of b_L may contain information about a_H. Other programs may be considered secure provided they appear in a sequential setting: for example, the program

$$(\text{while } a_H = tt \text{ do nil}); b_L := ff \tag{2}$$

is secure in a sequential setting since whenever it terminates it produces the same value ff for b_L. However, in a concurrent setting, this piece of code can be used in the following program, which always terminates, and in which the final value of b_L reflects the initial value of the high variable c_H when a_H and a'_H are initially assigned tt:

$$\begin{array}{c} \text{if } c_H = tt \text{ then } a_H := ff \text{ else } a'_H := ff \;\; || \\ (\text{while } a_H = tt \text{ do nil}); b_L := ff; a'_H := tt \;\; || \\ (\text{while } a'_H = tt \text{ do nil}); b_L := tt; a_H := ff \end{array} \tag{3}$$

The insecure flow that occurs here is usually called a *termination leak*, since it results from the 'termination behaviour' of a portion of the program.

In a sequential setting it makes sense to look only at the output values that a program may give, thus ignoring all its non-terminating computations. In fact, only the output values of terminating computations can be used by other programs that are sequentially composed with that program. Furthermore, if a computation enters a non-terminating loop, it is not possible for other programs to interrupt the loop since they will never get their turn to execute. However, as we have just seen, this is no longer true in a concurrent setting, which indicates that the context in which a program appears is important when considering the secureness of its execution.

2.1.1. *Migration leaks.* Here we are interested in the context of a distributed setting with code mobility, where programs are distributed over computation sites, and the possibility of execution or failure of programs cannot be guaranteed by one domain alone – it might, for instance, depend on their location. In order to understand the information flows that arise in such a setting, we will further enrich our simple concurrent language with the minimum features enabling us to simulate code mobility in a network.

We assume that threads are named, where M^n denotes a thread M named n, and that they are placed in *domains* for execution. The position of each thread in a network is given by a special 'location-variable' (for a thread n it is denoted by $pos(n)$), so that migration

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

is obtained by assigning a new domain name to such variables: $pos(n) := d$ now means that the thread n migrates to domain d (unless the value is already d, which means that it is already there). Furthermore, we assume that the location-variable $pos(n)$ can only be written by thread n (that is, migration is self-inflected, or *subjective*). We will make the simplest assumption, which is that the value of the location-variable $pos(n)$ can only be tested for equality to the location of the thread that tests it, that is, a thread located at d can only test whether $pos(n) = d$. This means that a thread can only determine whether threads are present in the domain it is in.

We can then write the following program, where a form of busy waiting (for the arrival of the thread m) is unblocked, depending on the value of a high variable a (assuming that initially we have $pos(m) = d$, $pos(n_1) = d_1$, and $pos(n_2) = d_2$):

$$\begin{aligned}(\text{if } a_H = tt \text{ then } (pos(m) := d_1) \text{ else } (pos(m) := d_2))^m \parallel \\ \parallel (\text{while } pos(m) \neq d_1 \text{ do nil}); (b_L := tt)^{n_1} \parallel \\ \parallel (\text{while } pos(m) \neq d_2 \text{ do nil}); (b_L := ff)^{n_2}\end{aligned} \tag{4}$$

Then, depending on the value of the high variable a, different low assignments may occur to the low variable b. We call the information leak that is set up by the above program a *migration leak* since it follows from the 'migrating behaviour' of threads.

2.2. *Choosing a calculus for global computing*

Most of the languages used to study information flow have been *local* in the sense that resources are assumed to be accessible to all programs at all times. Such an assumption does not hold in general for networks. In fact, a network can be viewed as a collection of computation sites – domains – where resources are only accessible by local programs, and failures can be generated by attempts to access remote resources. In order to study the problem of whether these forms of failures can give rise to information leaks like the one in Example (4), we will consider a language where the notion of the location of a program or a resource has an impact on computations.

The design of network models is a whole research area in itself, and there is a wide spectrum of calculi that focus on different aspects of mobility (Sekiguchi and Yonezawa 1997; Dal Zilio 2001). Here we are interested in a simple general framework that addresses the unreliable nature of resource access in networks, as well as trust concerns that are raised when computational entities have different security orientations. We will thus adopt the 'take-away type' of Sekiguchi and Yonezawa and allow references to move, together with the threads that own them, by means of a standard migration primitive included in our language. We also assume that a reference can only be accessed by a program that is located at the same site; however, inspired by ULM (Boudol 2004), remote accesses are suspended (without failure) until the references become available.

To illustrate the suspensive nature of reference access in the language studied in the next section, a read access to (that is the dereference of) a reference named a is denoted by '$(?\ a)$' while a write access to a, where a value V is assigned to it, is denoted by '$(a :=^? V)$'. If we consider a to belong to thread m, and that the access is performed at the domain d, then the behaviour of each of the above constructs is similar to preceding

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

the read or write operation by

$$(\text{while } pos(m) \neq d \text{ do nil}). \tag{5}$$

This should give an intuition for how migration leaks like the one in Example (4) can be encoded in our language (see Section 4).

2.3. *Security property*

The *non-interference* property, which states the total absence of insecure flows during the execution of a program, has been studied extensively (Cohen 1977; Goguen and Meseguer 1982; Sabelfeld and Myers 2003), and is still of theoretical interest due to its simple and elegant properties. However, it relies on the notion of a unique static security lattice that holds for the entire program, which is an unreasonable assumption in distributed settings, which are intrinsically decentralised. The applicability of pure non-interference is also impaired by the fact that it rejects programs that deliberately *declassify* information from high security levels to lower ones, thus disabling the use of programs that are very common and may even be unavoidable. A typical example is the ubiquitous password checking procedure, which reveals a little bit of information to any user attempting to 'log-in'.

In order to accept programs that violate non-interference in a controlled way, we must resort to a more flexible security property than non-interference. Among the various proposed mechanisms for allowing declassification (Sabelfeld and Sands 2005), we find the *flow declaration* construct (Almeida Matos and Boudol 2009), which is accompanied by a direct generalisation of non-interference called *non-disclosure*. Because of its flexibility, this language construct gives the programmer complete power to declare dynamically that different flow policies should hold in different parts of the program. Non-disclosure then states that each step performed by the program complies with the security policy that is declared at that moment for the executing thread. This can be used to express the permission allowing declassification within the scope of such declarations at any point of the program, and between any two security levels; in this perspective, information leaks are restricted to occuring within the lexical bounds of the flow declarations. Of possible greater interest is the fact that flow declarations can be used to express decentralised security policies in a natural way.

As is often done for non-interference in concurrent settings (Focardi and Gorrieri 1995; Sabelfeld and Sands 2000; Smith 2001; Boudol and Castellani 2002), non-disclosure is conveniently expressed using bisimulations. Dealing with small-step transitions (as opposed to describing the final result of a computation) is a natural choice for addressing the local nature of the flow declaration construct, since it is enough to label each transition with the flow policy that is declared by the evaluation context in order to know what policy holds for that particular step. For instance,

$$\langle P, S\rangle \underset{F}{\rightarrow} \langle P', S'\rangle \tag{6}$$

means that the program P changes state S into S' under the flow policy F. Then, by means of an appropriate bisimulation, it becomes easy to express the idea that a program

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

is secure if at each step it satisfies non-interference *with respect to the flow policy that holds for each computation step*.

Given the attractive qualities of non-disclosure and the flow declaration construct, amongst which is the possibility of allowing different threads to express different flow policies at the same time, is of particular interest here, we have introduced flow declarations into our language (Section 3). The next step is to explore the generalisability of the property to networks, that is, to distributed settings with code mobility.

2.3.1. *Non-disclosure for networks.* Current non-disclosure and other information flow properties were designed for local computations and are not suitable for treating information flows in a distributed setting with code mobility. Indeed, the notion of a safe program must reflect the fact that the location of references in a network can itself be a source of information leaks. To this end, we extend the notion of state with a mapping T that tracks the position of programs in a network.

Since the visibility of threads is a consequence of the possibility/impossibility of accessing any of its references, we associate with threads a security level, which is a lower bound on the levels of the references that it can own. We then extend the usual indistinguishability relation between memory locations into a more general one between states that track the positions of programs in a network. In this way, it becomes easy to generalise the usual bisimulation to one that is defined on a small-step semantics with transitions of the form

$$\langle P, T, S\rangle \underset{F}{\rightarrow} \langle P', T', S'\rangle. \tag{7}$$

The formalisation of non-disclosure is then based on such bisimulations (Section 4).

3. An imperative mobile λ-calculus

In our computation model, a *network* consists of a number of *domains*, which are places where local computations occur independently. Threads may execute concurrently inside domains, create other threads and *migrate* to another domain. They can own and create a memory space, which is a *store* that associates values to *references*, which are addresses of memory containers. These stores move together with the thread they belong to, which means that threads and their local references are, at all times, located in the same domain. However, a thread need not own a reference in order to access it. Read and write operations on references may be performed if and only if the corresponding memory location is present in the domain (otherwise they are implicitly suspended). We now present the language we will use for studying the security issues introduced by code mobility.

3.1. *Syntax*

The language of expressions is a higher-order λ-calculus that includes the imperative constructs of ML (Milner *et al.* 1997; Wright and Felleisen 1994), conditional branching and booleans values, as well as thread and reference creation. It also includes a

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Principals	p, q	$\in$	$\boldsymbol{Pri}$		
Security Levels	l, j, k	$\subseteq$	$\boldsymbol{Pri}$		
Flow Policies	F, G	$\subseteq$	$\boldsymbol{Pri} \times \boldsymbol{Pri}$		
Thread Identifiers	$\check{m}, \check{n}$	$\in$	$\check{\boldsymbol{Nam}}$		
Effects	s			$::=$	$\langle l, l, l \rangle$
Type Variables	t				
Types	τ, σ, θ	$\in$	$\boldsymbol{Typ}$	$::=$	$t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta\ \mathrm{ref}_{l,\check{m}_j} \mid \tau \xrightarrow[G,\check{m}_j]{s} \sigma$

Fig. 1. Syntax of security annotations and types

Variables	x, y	$\in$	$\boldsymbol{Var}$		
Domain Names	d	$\in$	$\boldsymbol{Dom}$		
Thread Names	m, n	$\in$	$\boldsymbol{Nam}$		
Reference Identifiers	u, v	$\in$	$\boldsymbol{Ref}$		
Reference Names	a, b, c			$::=$	$m_j.u$
Decorated Thread Names				$::=$	m_j
Decorated Reference Names				$::=$	$a_{l,\theta}$
Values	V	$\in$	$\boldsymbol{Val}$	$::=$	$() \mid x \mid a_{l,\theta} \mid (\lambda x.M) \mid tt \mid ff$
Pseudo-values	W	$\in$	$\boldsymbol{Pse}$	$::=$	$V \mid (\varrho x.W)$
Expressions	M, N	$\in$	$\boldsymbol{Exp}$	$::=$	$W \mid (M\ N) \mid (M; N) \mid$ $(\mathrm{ref}_{l,\theta}\ M) \mid (?\ N) \mid (M :=^{?} N) \mid$ $(\text{if } M \text{ then } N_t \text{ else } N_f) \mid$ $(\mathrm{thread}_l\ M) \mid (\text{flow } F \text{ in } M) \mid$ $(\text{goto } d)$

Fig. 2. Syntax of expressions

Threads			$::=$	$M^{m_j} \quad (\in \boldsymbol{Exp} \times \boldsymbol{Nam} \times 2^{\boldsymbol{Pri}})$
Pool of Threads	P	:		$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \rightarrow \boldsymbol{Exp}$
Position-Tracker	T	:		$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \rightarrow \boldsymbol{Dom}$
Store	S	:		$(\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Ref} \times 2^{\boldsymbol{Pri}} \times \boldsymbol{Typ}) \rightarrow \boldsymbol{Val}$
Networks	X, Y		$::=$	$d[\ P, S\ ] \mid X \parallel Y$
Configurations			$::=$	$\langle P, T, S \rangle$

Fig. 3. Syntax of configurations

declassification mechanism, and is enriched with a notion of domain and a basic mobility primitive. We define the syntax of security annotations, types, expressions and networks (configurations) in Figures 1, 2 and 3.

3.1.1. *Security annotations and types.* We will assume a set $\boldsymbol{Pri}$ of all the principals in the system, denoted by p, q. A security level l, j, or k is then a subset of $\boldsymbol{Pri}$ (see Figure 1).

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

They are apparent in the syntax as they are associated with references (and reference creators), as well as threads (and thread creators). The security level of a reference is to be understood as the set of principals that are allowed to read the information contained in that reference.

A flow policy F is a set of pairs of principals, where a pair $(p, q) \in F$, which we will usually write as $p \prec q$, which is to be understood as 'information may flow from principal p to principal q', that is, more precisely, '*everything that principal p is allowed to read may also be read by principal q*'.

Types and effects appear in the syntax of the language (see Figure 1), and will be explained in Section 5. These annotations do not play any role in the operational semantics, but are used for proving type soundness.

3.1.2. *Expressions.* The syntax of expressions is defined in Figure 2. We assume four disjoint countable sets $\boldsymbol{Dom} \neq \emptyset$, ***Nam***, ***Var*** and ***Ref***. *Names* are given to domains ($d \in \boldsymbol{Dom}$), threads ($m, n \in \boldsymbol{Nam}$) and references ($a, b, c$), which we also call *addresses*. We add annotations (subscripts) to names: *decorated thread names* carry the threads security level, while *decorated reference names* carry the references security level and the type of the values that they can hold, as well as the security level of the thread that owns them. So a decorated thread name m_j consists of a pair made up of a thread name m and a security level j, while a decorated reference name $m_j.u_{l,\theta}$ is a 5-tuple made up of a thread name m, its security level j, a reference identifier u, a type θ and a security level l. References are lexically associated with the threads that create them: they are of the form $m_j.u$, where u is an identifier given by the thread. Thread and reference names can be created at runtime. In the following we may omit subscripts whenever they are irrelevant, following the convention that the same name always has the same subscript.

Values, ranged over by $V \in \boldsymbol{Val}$, are special expressions that cannot compute, and include:

— the command () that does nothing;

— the function abstraction $(\lambda x.M)$ with body M and parameter x;

— the boolean values tt and ff.

The construct $(\varrho x.W)$, which binds the occurrences of x in the pseudo-value W, is used to express recursive values – it recursively executes the result of applying $(\lambda x.W)$ to itself.

The set ***Exp*** of *expressions*, ranged over by M, N, includes:

— the application $(M\ N)$, which applies the function that results from computing M to the result of the computation of N;

— the conditional (if M then N_t else N_f), which executes either N_t or N_f depending on whether the computation of M renders tt or ff;

— the sequential composition $(M; N)$, which executes N after the execution of M has terminated;

— the dereferencing operation $(?\ M)$, which, after M has executed and returned a decorated reference name, returns the value that the reference points to;

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

— the assignment ($M :=^{?} N$) of the value returned by the computation of N to the decorated reference name returned by the computation of M[†];
— the thread creator (thread M), which spawns the thread M, which is to be executed concurrently, and returns $()$;
— the migration construct (goto d), where d is a domain name, which triggers the migration of the thread that executes the migration operation to the domain d;
— the flow declaration construct (flow F in M), where F is a flow policy, and M is any expression of the language, which has the same behaviour as M.

Other useful commands can be derived from the above expressions. For example, we can write the let construct (let $x = N$ in M) as $((\lambda x.M)\ N)$. We can write recursive functions as $(\varrho f.(\lambda x.M))$, which is close to (let rec $f = (\lambda x.M)$ in f) written in an ML-like notation. We use loop to denote the expression $(\varrho x.x)$. We may encode while loops in the following standard way:

$$(\text{while } M \text{ do } N) \stackrel{\text{def}}{=} ((\varrho y.(\lambda x.(\text{if } M \text{ then } (N;(y\ x)) \text{ else } x)))\ ()). \tag{8}$$

We do not derive sequential composition, or other imperative features, from the functional part of the language for typing reasons (as in Almeida Matos and Boudol (2009) and Boudol (2005b)).

3.1.3. *Networks and configurations.* We define *stores* S, which map decorated reference names to values, and *threads*, which are named expressions M^{m_j} (the names are decorated). Threads run concurrently in pools P, which are mappings from decorated thread names to expressions (they can also be viewed as sets of threads). Networks are flat juxtapositions of domains, each containing a store and a pool of threads. Thread and domain names are assumed to be distinct; furthermore, references are assumed to be located in the same domain as the thread that owns them (the owner thread's name is a prefix of the reference's name) and to always have the same decorations.

Notice that networks are in fact just collections of threads and owned references that are running in parallel, and whose executions depend on their relative location. To keep track of the locations of threads and references, it suffices to maintain a mapping from thread names to domain names. This is the purpose of T, a *position-tracker*, which is a mapping from a finite set of decorated thread names to domain names. Together with the pool P containing all the threads in the network, and the store S containing all the references in the network, it forms a *configuration* $\langle P, T, S\rangle$, on which the reduction relation is defined in the next subsection. More precisely, given a set $\mathscr{D}$ of domain names in a network, we obtain a configuration of the form $\langle P, T, S\rangle$ from a network $d_1[\ P_1, S_1\ ] \parallel \cdots \parallel d_n[\ P_n, S_n\ ]$, where:

— $T = \{m_j \mapsto d_1 | M^{m_j} \in P_1\} \cup \cdots \cup \{m_j \mapsto d_n | M^{m_j} \in P_n\}$
— $P = P_1 \cup \cdots \cup P_n$
— $S = S_1 \cup \cdots \cup S_n$.

[†] The notation '?' indicates the fact that these operations are potentially suspensive when the reference that is being read or written to is not accessible.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

3.2. *Semantics*

We now define the semantics of the language as a small-step operational semantics on configurations. To this end, we begin by defining some useful notation and conventions. We then describe the transitions on configurations, which are based on evaluation contexts, and state some properties of the semantics.

3.2.1. *Basic sets and functions.* Given a configuration $\langle P, T, S\rangle$, we say the pair (T, S) is the *state* of the configuration. We define $\mathrm{dom}(T)$, $\mathrm{dom}(P)$ and $\mathrm{dom}(S)$ as the sets of decorated names of threads and references that are mapped by T, P and S, respectively. We say that a thread or reference name is fresh in T or S if it does not occur, with any subscript, in $\mathrm{dom}(T)$ or $\mathrm{dom}(S)$, respectively. We use $\mathrm{tn}(P)$ and $\mathrm{rn}(P)$ to denote the set of decorated thread and reference names, respectively, that occur in the expressions of P (this notation is extended in the obvious way to expressions). Furthermore, we overload tn and define, for a set R of reference names, the set $\mathrm{tn}(R)$ of thread names that are prefixes of the names in R.

We restrict our attention to well-formed configurations $\langle P, T, S\rangle$ that satisfy the following conditions for stores, values stored in stores and thread names:

— $\mathrm{rn}(P) \subseteq \mathrm{dom}(S)$;
— $a_{l,\theta} \in \mathrm{dom}(S)$ implies $\mathrm{rn}(S(a_{l,\theta})) \subseteq \mathrm{dom}(S)$;
— $\mathrm{dom}(P) \subseteq \mathrm{dom}(T)$;
— $\mathrm{tn}(\mathrm{dom}(S)) \subseteq \mathrm{dom}(T)$;
— all threads in a configuration have distinct names; and
— all occurrences of a name in a configuration are decorated in the same way.

We use $\{x \mapsto W\}M$ to denote the capture avoiding substitution of W for the free occurrences of x in M. The operation of adding or updating the image of an object z to z' in a mapping Z is denoted $[z := z']Z$.

3.2.2. *Evaluation contexts and flow contexts.* In order to define the evaluation order, it is convenient to write expressions using evaluation contexts. We write $\mathrm{E}[M]$ to denote an expression where the subexpression M is placed in the evaluation context E obtained by replacing the occurrence of [] in E by M. The evaluation contexts of the language define a call-by-value evaluation order (see Figure 4). Intuitively, the expressions that are placed in such contexts are to be executed first. Evaluation is *not* allowed under threads that have not yet been created.

The analysis of whether the information flows that occur in M are to be allowed will depend on the flow policies that are declared in the evaluation context where M is executed (see next section). We use $\lceil \mathrm{E} \rceil$ to denote the flow policy that is permitted by the evaluation context E. It collects all the flow policies that are declared using flow declaration constructs into one single flow policy using set union. The other evaluation contexts do not affect the flow policy of the context. This leads us to the following definition.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

$$\begin{array}{rcl} \textit{Evaluation Contexts}\ \mathrm{E} & ::= & [] \mid (\mathrm{E}\ N) \mid (V\ \mathrm{E}) \mid (\mathrm{E}; N) \mid \\ & & (\mathrm{ref}_{l,\theta}\ \mathrm{E}) \mid (?\ \mathrm{E}) \mid (\mathrm{E} :=^{?} N) \mid (V :=^{?} \mathrm{E}) \mid \\ & & (\text{if E then } N_t \text{ else } N_f) \mid (\text{flow } F \text{ in E}) \end{array}$$

Fig. 4. Evaluation contexts

Definition 3.1 (Flow policy declared by an evaluation context). The *flow policy declared by the evaluation context* E is given by $\lceil \mathrm{E} \rceil$ where

$$\begin{aligned} \lceil [] \rceil &= \varnothing \\ \lceil (\text{flow } F \text{ in E}) \rceil &= F \cup \lceil \mathrm{E} \rceil \\ \lceil \mathrm{E}'[\mathrm{E}] \rceil &= \lceil \mathrm{E} \rceil \qquad \text{(if E}' \text{ does not contain flow declarations).} \end{aligned}$$

3.2.3. *Small-step semantics.* The transitions of our (small-step) semantics are defined between configurations. The evaluation rules are defined in Figure 5. We start by defining the transitions of a single thread (we omit the set-brackets for pools that are singletons). These are decorated with the thread N^{n_k} that may be spawned during that transition, where $N^{n_k} = ()$ if no thread is created. The last three rules use the information contained in the label to add any spawned threads to the pool of threads. The final rule means that the execution of a pool of threads is compositional.

The transitions are also decorated with the flow policy that is declared by the evaluation context where they are performed – see Figure 5. Note, however, that the transitions do not depend on the flow label F that decorates them. These labels are just annotations, which will be used later during the security analysis. The evaluation of (flow F in M) is just the evaluation of M annotated with a flow policy that comprises (in the sense of set inclusion) F. The lifespan of the flow declaration terminates when the expression M that is being evaluated terminates (that is, M becomes a value); in this final step, the declared flow policy is irrelevant, and therefore discarded. These annotations express the fact that the flows declared in F are allowed to occur during the execution of M. Notice that we can easily declare two concurrent threads to have different flow policies:

$$d[\ \{(\text{flow } F_1 \text{ in } M_1)^{m_1}, (\text{flow } F_2 \text{ in } M_2)^{m_2}\}, S\ ]. \tag{9}$$

The evaluation of the following expressions depends only on the expressions themselves:

— The application of a function with parameter x and body M to a value V returns the substitution of all free occurrences of x in M by V.
— A conditional with a test on a boolean value V and branches N_t and N_f returns N_t if the value is tt and N_f if the value is ff.
— The sequential composition of a value and an expression N returns N (the value is not used by this form of composition, and is therefore discarded).
— The fix point of a pseudo-value W bound by x results in the substitution of the free occurrences of x in W by the expression itself.

The evaluation of some expressions might depend on and change the store:

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\langle P, T, S\rangle \underset{F}{\rightarrow} \langle P', T', S'\rangle \quad \langle P \cup Q, T, S\rangle \textit{ is well formed}}{\langle P \cup Q, T, S\rangle \underset{F}{\rightarrow} \langle P' \cup Q, T', S'\rangle}$$

Fig. 5. Semantics

— The creation of a reference of level l and type θ containing the value V returns a reference with a name that does not occur so far in the store (say a), and adds the pair $((a, l, \theta), V)$ to the store.
— The dereferencing of a reference, if it is not suspended, returns the value to which the reference is mapped.
— The assignment of a value V to a reference $a_{l,\theta}$, if it is not suspended, returns () and updates the store by replacing any occurrence of a pair $((a, l, \theta), V')$ (where V' is the old value of a) by $((a, l, \theta), V)$.

The relation between these operations and the position-tracker is as follows:

— When a reference is created by a thread m, it is named with a fresh name $m.u$ after the parent thread, for some fresh reference identifier u.
— The dereference and assignment of a reference that belongs to a thread named n is only performed by a thread named m if m and n are both located at the same domain.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

— When a thread is created, its new fresh name and position is added to the position-tracker.
— When the (goto d) statement is executed by a thread m, the position of m in the position-tracker is updated to d.

Summing up, the name of the thread is used in the following rules:

— for the creation of a reference, which is named after the parent thread;
— when a new thread is created, and attached to a domain (namely the parent's one); and,
— in accesses (read or write) to references, which can only be performed if the accessing thread and the reference are placed in the same domain, as pointers to the position of the corresponding threads.

When a new thread is created, the flow policy that is permitted by the evaluation context of the parent thread is not kept (unlike the case in Almeida Matos and Boudol (2009)). We thus leave open the option of declaring a more permissive flow policy for the thread that is created.

3.2.4. *Properties of the semantics.* One can prove that the semantics preserves the conditions for well formedness. Furthermore, a configuration with a single expression has at most one transition, up to the choice of new names.

Next we show a simple but crucial property of the semantics, pinpointing the situations in which two computations of the same thread can split, that is, can yield different threads. Apart from the situations in which two distinct fresh references or thread names are created, this can only occur if the expression is about to read a reference that is given different values by the memories in the starting configurations. More precisely, the following result states that if the evaluation of a thread M^{m_j} differs in the context of two distinct states while not creating two distinct reference names or thread names, this is because M^{m_j} is performing a dereferencing operation, which yields syntactically different results depending on the memory.

Lemma 3.2 (Splitting Computations).
Suppose

$$\langle M^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle {M_1}'^{m_j}, T_1', S_1'\rangle$$
$$\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N'^{n_k}} \langle {M_2}'^{m_j}, T_2', S_2'\rangle$$

with

$${M_1}'^{m_j} \neq {M_2}'^{m_j}$$
$$\mathrm{dom}(T_2' - T_2) = \mathrm{dom}(T_1' - T_1)$$
$$\mathrm{dom}(S_2' - S_2) = \mathrm{dom}(S_1' - S_1).$$

Then,

$$N^{n_k} = () = N'^{n_k},$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

and there exist E and $a_{l,\theta}$ such that

$$F = \lceil \mathrm{E} \rceil = F'$$
$$M = \mathrm{E}[(?\ a_{l,\theta})]$$
$$M'_1 = \mathrm{E}[S_1(a_{l,\theta})]$$
$$M'_2 = \mathrm{E}[S_2(a_{l,\theta})]$$

with

$$\langle T'_1, S'_1 \rangle = \langle T_1, S_1 \rangle$$
$$\langle T'_2, S'_2 \rangle = \langle T_2, S_2 \rangle$$
$$S_1(a_{l,\theta}) \neq S_2(a_{l,\theta}).$$

Proof. The proof is by case analysis on the transition

$$\langle M^{m_j}, T_1, S_1 \rangle \xrightarrow[F]{N^{n_k}} \langle {M_1}'^{m_j}, T'_1, S'_1 \rangle.$$

Note that the only rule where the state is used is the one for $\mathrm{E}[(?\ a_{l,\theta})]$. □

Note that the inequalities ${M_1}'^{m_j} \neq {M_2}'^{m_j}$ and $S_1(a_{l,\theta}) \neq S_2(a_{l,\theta})$ are strictly syntactic. The conditions $\mathrm{dom}(T'_2 - T_2) = \mathrm{dom}(T'_1 - T_1)$ and $\mathrm{dom}(S'_2 - S_2) = \mathrm{dom}(S'_1 - S_1)$ allow us to ignore the differences in the programs M'_1 and M'_2 that might result from the non-deterministic choice of new names.

4. The non-disclosure policy for networks

In this section we formally define non-disclosure for networks. We start by defining the security pre-lattices in terms of a flow relation that is parameterised by the context's flow policy, and discuss the meaning of a 'thread flow policy'. We will then exhibit an indistinguishability relation on states and give a bisimulation definition of non-disclosure for networks. Finally, we will justify the security property using examples and give some properties of secure programs.

4.1. *Security (pre-)lattices*

Our examples so far have just considered the simple case where there are only two security levels – high and low. However, our explanations can be extended to a setting with an arbitrary number of security levels. It is standard to let security levels form a lattice (Bell and La Padula 1976; Denning 1976), which are based on partial order relations (reflexive, transitive and anti-symmetric), and have the property that any two of its elements have a (unique) least upper-bound and a (unique) greatest lower-bound, upon which the (unique) meet and join operations are defined. However, the uniqueness property is not crucial here. Instead, we choose to deal with *pre-lattices*, which are analogously based on preorder relations (which are reflexive and transitive, but not necessarily anti-symmetric). We now give the formal definition of the specific pre-lattices that we use in our framework, and of the meet and join operators we adopt.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

4.1.1. *Concrete security pre-lattices.* With confidentiality in mind, we view security levels of objects as a specification of who is authorised to read information contained in them, representing read-access rights to references (as in access control lists). This is apparent in the syntax of our language, where we have seen that security levels j, k, l are sets of principals $p, q \in \boldsymbol{Pri}$. Our aim is then to ensure that information contained in a reference a_{l_1} (omitting the type annotation) does not leak to another reference b_{l_2} that gives a read access to an unauthorised principal p, that is, such that $p \in l_2$ but $p \notin l_1$. From this point of view, given a set ***Pri*** of principals, an object labelled ***Pri*** (also denoted $\bot$) is a most public one, which every principal is allowed to read, whereas the label $\varnothing$ (also denoted $\top$) indicates a secret object, which is so secret that no principal is allowed to read it. One can easily see that given a set ***Pri*** of principals, the pair $(2^{\boldsymbol{Pri}}, \supseteq)$ is a lattice, where the meet and join are set union and intersection, respectively.

So far we have considered the case of a flat structure of principals, assuming no communication between them. However, one might wish to express the fact that whatever principal p can read, principal q can also read. This is done by means of flow policies, which are sets of such statements. As we have seen, a flow policy is represented in our setting as a binary relation over ***Pri***. We let F, G range over such relations, where a pair $(p, q) \in F$, often written $p \prec q$, is to be understood as 'information may flow from principal p to principal q', that is, and more precisely, '*everything that principal p is allowed to read may also be read by principal q*'. As usual, we use F^* to denote the reflexive and transitive closure of F.

We will now see how the above lattice $(2^{\boldsymbol{Pri}}, \supseteq)$ can be customised by means of relations on principals by defining the *preorder on security levels* $\preceq_F$ that is determined by the flow policy F. To this end, we use the notion of an *F-upward closure* of a security level l defined by $l \uparrow_F = \{q \mid \exists p \in l.\ p\ F^*\ q\}$. The F-upward closure of l contains all the principals that are allowed by the policy F to read at level l. We can now derive (as in Myers and Liskov (1998) and Almeida Matos and Boudol (2009)) a more permissive *flow relation* $\preceq_F$ such that

$$l_1 \preceq_F l_2$$

is defined as

$$\forall q \in l_2\ .\ \exists p \in l_1\ .\ (p, q) \in F^* \tag{10}$$

or, equivalently, as

$$(l_1 \uparrow_F) \supseteq (l_2 \uparrow_F), \tag{11}$$

and use it to define the pre-lattice that is determined by a flow policy. Notice that $\preceq_F$ extends $\supseteq$ in the sense that $\preceq_F$ is larger than $\supseteq$ and that $\preceq_\varnothing \ = \ \supseteq$.

Definition 4.1 (Security pre-lattice). Given a set ***Pri*** of *principals* and a flow policy F in $\boldsymbol{Pri} \times \boldsymbol{Pri}$, the pair $(2^{\boldsymbol{Pri}}, \preceq_F)$ is a *security pre-lattice*, where *meet* $(\curlywedge_F)$ and *join* $(\curlyvee_F)$ are given by the union and intersection, respectively, of the F-upward closures with respect to F:

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

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

We will use the mechanism of extending the flow relation with a flow policy F as follows: if G is a *global* flow policy (that is, a static security policy that holds in an entire system), the information flows that are allowed to occur in an expression M placed in a context E[] must satisfy the flow relation $\preceq_{G \cup \lceil \mathrm{E} \rceil}$.

4.1.2. *Imposing a flow policy.* In a decentralised setting, it is not straightforward to imagine what the traditional notion of a global flow policy should be since it is unrealistic to assume that all participants would agree on it. A practical and conservative approach could be to assume the minimum global flow policy $\varnothing$, which clearly all security pre-lattices satisfy. Nevertheless, for the sake of generality, we can admit the existence of a global flow policy G that all participants comply with (for example, in an environment where $F_1, \ldots, F_n$ are the relevant flow policies, G could be defined as any subset of $\bigcap_{1 \leqslant i \leqslant n} F_i^*$), and use it to parameterise the flow relation. Local flow policies can then be declared to extend the global flow policies in a decentralised manner.

4.2. *A bisimulation-based definition*

We now define our security property in terms of the flow relation $\preceq_F$ defined above, where F is the current flow policy.

4.2.1. *Low-equality.* 'Low-equality' is an informal designation of an equality relation that considers as indistinguishable two memories that coincide in their 'low part'. The low part of the memory is defined with respect to a flow relation $\preceq_F$ and to a security level l that is considered to be 'low', and, consequently, so are all levels lower than l with respect to $\preceq_F$ (both l and F are used as parameters).

As we will see towards the end of this section, in a distributed environment, the position of a thread in the network can reveal information about the values in the memory. Furthermore, threads can detect each other's presence by attempting to access the other's references (they succeed if and only if the two are located within the same domain). Therefore, threads that own low references can be viewed as 'low threads', and their locations should be the same in low-equal states. Our notion of low-equality is then extended to states, and their low part is defined pointwise.

Definition 4.2 (Low part of a state). The low part of a state $\langle T, S \rangle$ is composed of the low part of a memory S and of the position-tracker T with respect to a flow policy F and a security level l, which are given by

$$T \restriction^{F,l} \stackrel{\text{def}}{=} \{(n_k, d) \mid (n_k, d) \in T \ \& \ k \preceq_F l\}$$
$$S \restriction^{F,l} \stackrel{\text{def}}{=} \{(a_{k,\theta}, V) \mid (a_{k,\theta}, V) \in S \ \& \ k \preceq_F l\}.$$

We then say that two states are low-equal if they coincide (syntactically) in their low part.

Definition 4.3 (Low-equality). Low-equality between states $\langle T_1, S_1 \rangle$ and $\langle T_2, S_2 \rangle$ with respect to a flow policy F and a security level l is given by the conjunction of the

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

syntactic equalities between the low parts of memories S_1 and S_2 and of position-trackers T_1 and T_2, with respect to the same security level and flow policy:

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

and

$$S_1 \restriction^{F,l} = S_2 \restriction^{F,l}.$$

Summarising, two states are said to be low-equal if they have the same low-domain and they give the same values to low references and the same positions to low threads. This relation is also transitive, reflexive and symmetric. We shall use the following fact.

Remark 4.4. $F \subseteq F'$ and $\langle T_1, S_1\rangle =^{F',l} \langle T_2, S_2\rangle$ implies $\langle T_1, S_1\rangle =^{F,l} \langle T_2, S_2\rangle$.

4.2.2. *The security property.* We will design a bisimulation to express the requirement that two networks are to be related if they show the same behaviour on the low part of any two low-equal states. Then, if a network is shown to be bisimilar to itself, one can conclude that during its computation, the high part of a state cannot interfere with its low part, that is, no security leak can occur. This idea will be used to define secure networks.

We now present a bisimulation for networks, which is defined between sets of threads P with respect to a low security level and, since the notion of 'being low' uses the flow relation $\preceq_G$, the global flow policy G also appears here as a parameter. We shall use $\twoheadrightarrow$ to denote the reflexive closure of the union of the transitions $\underset{F}{\rightarrow}$, for all F.

Definition 4.5 ((G,l)-bisimulation). A (G,l)-*bisimulation* is a symmetric relation $\mathscr{R}$ on sets of threads such that

$$P_1 \mathscr{R} P_2 \;\&\; \langle P_1, T_1, S_1\rangle \underset{F}{\rightarrow} \langle P_1', T_1', S_1'\rangle \;\&\; \langle T_1, S_1\rangle =^{G\cup F,l} \langle T_2, S_2\rangle$$

with

$$\mathrm{dom}(S_1{}') - \mathrm{dom}(S_1) \cap \mathrm{dom}(S_2) = \varnothing$$
$$\mathrm{dom}(T_1{}') - \mathrm{dom}(T_1) \cap \mathrm{dom}(T_2) = \varnothing$$

implies

$$\exists T_2', P_2', S_2' \,.\, \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle \;\&\; \langle T_1', S_1'\rangle =^{G,l} \langle T_2', S_2'\rangle \;\&\; P_1' \mathscr{R} P_2'.$$

The conditions $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \varnothing$ and $\mathrm{dom}(T_1{}' - T_1) \cap \mathrm{dom}(T_2) = \varnothing$ guarantee that any reference or thread name that may be created by P_1 does not conflict with the free names of P_2. The absence of a condition on the flow policy of the matching move for P_2 enables all expressions that do not change the state to be bisimilar, independently of the flow policy that is declared by their evaluation contexts.

When P_1 performs a transition within the scope of the local flow policy F, it is allowed to read low-references from the input state $\langle T_1, S_1\rangle$ according to the current flow policy $G \cup F$. Recall that these references are labelled with a security level l' such that $l' \preceq_{G\cup F} l$. As for the output states, they must be low-equal with respect to the global flow policy G. By starting with pairs of memories that are low-equal 'to a greater extent', that is,

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

that coincide in a larger portion of the memory, the condition on the behaviour of the program P_2 becomes weaker. As a consequence, we can potentially relate more programs, thus accepting more of them as being secure. This is where we generalise classical non-interference, where both low-equality conditions are established with respect to the global flow policy.

The main difference between our non-disclosure definition and that of Almeida Matos and Boudol (2009) is that the position of low threads is treated as low-information. Another difference lies in the requirement that here the creation of new references must be matched by both threads due to the condition $\langle T_1', S_1' \rangle =^{G,l} \langle T_2', S_2' \rangle$, which requires the domains to match. This means that freshly created references are observable, which may be thought of as addressing an unrealistic power of the observer. This option was taken in order to gain some technical simplicity: our low-equality becomes an equivalence relation, which simplifies the soundness proof.

Remark 4.6.

— For any G and l there exists a (G, l)-bisimulation, such as, for instance, the set $\boldsymbol{Val} \times \boldsymbol{Val}$ of pairs of values.
— The union of a family of (G, l)-bisimulations is a (G, l)-bisimulation.

Consequently, there is a largest (G, l)-bisimulation, which is the union of all (G, l)-bisimulations.

Notation 4.7 ($\approx_{G,l}$). The largest (G, l)-bisimulation is denoted by $\approx_{G,l}$.

Note that the relation $\approx_{G,l}$ is not reflexive. For instance, the insecure expression $(v_B :=^{?} (?\, u_A))$ is not bisimilar to itself if $A \not\leq_G B$, since the low-equality assumption between the two states under consideration does not guarantee that u_A has the same value in both. This is, in fact, the whole point of the security relation since as we can see from the following definition, it should only be 'reflexive' with respect to secure programs.

Definition 4.8 (Non-disclosure for networks with respect to G). A pool of threads P satisfies the non-disclosure for networks policy (or is *secure* from the point of view of non-disclosure for networks) with respect to the global flow policy G if it satisfies $P \approx_{G,l} P$ for all security levels l. We then write $P \in \mathcal{NDN}(G)$.

Intuitively, our security property states that, at each computation step performed by some thread in a network, the information flows that occur respect the global flow policy, extended with the flow policy (F) that is declared by the context where the command is executed. In order to motivate our definition, we will now give some examples of insecure programs, focusing on those that set up migration leaks.

4.2.3. *Examples of insecure programs.* In the following examples, we again assume the simplified setting with only two security levels H and L such that $L \preceq H$. As usual, we use a_H or b_L to denote references with security levels $\{H\}$ or $\{L\}$, leaving out the type and the brackets. For simplicity, we will omit the stores, thread names and levels whenever they are irrelevant.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

The standard examples of direct leaks and control leaks (see Subsection 2.1), which in our language are written

$$(a_L :=^? (?\ b_H)) \tag{12}$$

$$(\text{if } (?\ a_H) \text{ then } (b_L :=^? tt) \text{ else } (b_L :=^? ff)) \tag{13}$$

do not satisfy non-disclosure for networks.

Using a bisimulation approach to security allows us to reject termination leaks, such as in

$$((\text{if } (?\ a_H) \text{ then } () \text{ else } \mathsf{loop}); (b_L :=^? tt)) \tag{14}$$

where writing at level L depends on reading at level H. This is because one of the branches that might result from the conditional $(\mathsf{loop}; (b_L :=^? tt))$ cannot simulate the other one $((); (b_L :=^? tt))$ in its change of the low reference b_l. Another example of a termination leak that arises in higher-order settings is

$$((?\ a_H)(); (b_L :=^? tt)). \tag{15}$$

Indeed, the dereferencing of the high reference a_H can be viewed as a 'high test' that might result in a terminating or non-terminating branch – take, for instance, $(\lambda y.(\lambda x.x))$ or $(\lambda y.\mathsf{loop})$ as two possible values of a_H in two low-equal memories. The application of these functions to $()$ unravels two expressions with different behaviours, as in the previous example. Similarly, there is a termination leak in

$$(((\lambda x.(x\ ()))\ (?\ a_H)); (b_L :=^? tt)) \tag{16}$$

since in one step we obtain Example (15).

The bisimulation we use is said to be 'strong'[†] as in Sabelfeld and Sands (2000) since each time a transition is matched, we restart the bisimulation game by comparing the resulting pools of threads in the context of any new low-equal memories, rather than continuing with the resulting configurations. This allows us to detect an illegal flow in

$$d_1[\ (\text{if } (?\ w_X) \text{ then } ((\text{while } (?\ w_X) \text{ do } ()); (v_L :=^? (?\ u_H))) \text{ else } ()), S_1\ ], \tag{17}$$

which, in the case where $S_1(w_X) = tt$ can be unravelled by other programs that execute concurrently in the pool of threads, or even at another site:

$$d_2[\ ((\text{goto } d_1); (w_X :=^? ff)), S_2\ ]. \tag{18}$$

This demanding definition for bisimulations seems appropriate, therefore, for dealing with mobile code scenarios, where the shared memory of a system of threads can be modified by incoming code.

Insecure migrations. We will now give some examples of how information about the position of a thread in a network can be accessed through the references that the thread owns. This suggests that the security level of that information should be a lower bound

[†] Note that this is not the same use of the word 'strong' as in Milner's CCS.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

to the levels of the threads' references. This is how, in the next section, we will obtain the security level associated with the threads' names.

The suspension occurring on an access to an absent reference can be unblocked by other threads. This allows us to write a program that is similar to Example (3), where non-termination is encoded by a suspended access and unblocked by migration:

$$\begin{array}{c} d[\ (\text{if } a_H \text{ then } (\text{goto } d_1) \text{ else } (\text{goto } d_2))^{n_k}\ ] \parallel \\ d_1[\ ((n_k.x_\top :=^? 0);(m_{1j_1}.y_L :=^? 1))^{m_{1j_1}}\ ] \parallel \\ d_2[\ ((n_k.x_\top :=^? 0);(m_{2j_2}.y_L :=^? 2))^{m_{2j_2}}\ ]. \end{array} \tag{19}$$

Then, depending on the value of the high reference a_H, different low assignments would occur to the low references $m_1.y_L$ and $m_2.y_L$. The same example can show a potential leak of information about the positions of the threads m_1 and m_2 via their own low references $m_1.y_L$ and $m_2.y_L$.

An analogous, but more direct, example shows that the mere arrival of a thread and its references in another domain could trigger a suspended low assignment:

$$\begin{array}{c} d[\ (\text{if } a_H \text{ then } (\text{goto } d_1) \text{ else } (\text{goto } d_2))^{n_k}\ ] \parallel \\ d_1[\ (n_k.y_L :=^? 1)^{m_{1j_1}}\ ] \parallel \\ d_2[\ (n_k.y_L :=^? 2)^{m_{2j_2}}\ ]. \end{array} \tag{20}$$

The previous examples show how the migration of a thread can result in an information leak from a high reference to a lower one via an 'observer' thread. It is the ability of the observer thread to detect the presence of the first thread that allows the leak. However, one must also prevent the thread itself from revealing information about its own position, such as by performing a low assignment following a remote assignment

$$d[\ ((n.u_\top :=^? 0);(b_L :=^? 0))^{m_H}\ ] \tag{21}$$

or a remote dereference

$$d[\ ((?\ n.u_\top);(b_L :=^? 0))^{m_H}\ ], \tag{22}$$

or when the low assignment itself is remote

$$d[\ (b_L :=^? 0)^{m_H}\ ]. \tag{23}$$

4.3. *Properties of secure programs*

Security is compatible with composition by set union.

Proposition 4.9 (Compositionality).

$$P \in \mathcal{NDN}(G)$$

and

$$Q \in \mathcal{NDN}(G)$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

implies

$$(P \cup Q) \in \mathscr{NDN}(G).$$

Proof. If $\mathscr{S}_1$ is a (G,l)-bisimulation such that $P\ \mathscr{S}_1\ P$ and $\mathscr{S}_2$ is a (G,l)-bisimulation such that $Q\ \mathscr{S}_2\ Q$, then $\mathscr{S} = \mathscr{S}_1 \cup \mathscr{S}_2$ is a (G,l)-bisimulation such that $(P \cup Q)\ \mathscr{S}\ (P \cup Q)$. □

4.3.1. *Operationally high threads.* There is a class of threads that have the property of never performing any change in the low part of the state. These are classified as being 'high' according to their behaviour†.

Definition 4.10 (Operationally high threads). A set $\mathscr{H}$ of threads is a set of *operationally* (F,l)*-high threads* if the following holds for all $M^{m_j} \in \mathscr{H}$:

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

implies

$$\langle T, S\rangle =^{F,l} \langle T', S'\rangle$$

and both

$$M'^{m_j}, N^{n_k} \in \mathscr{H}.$$

Even if a thread contains low assignments or low reference creations, it can be considered *operationally high* if these commands are never reached in any execution, or if the assignments do not change the value of any reference. Notice that the low part of the state is considered with respect to the parameter F, and no account is taken of the flow policy of the thread's transitions. When F is the global flow policy, this definition is consistent with the definition of bisimulation, where the 'observation' of the memories that result from a step are taken from the point of view of the global flow policy.

Remark 4.11.

— For any F and l, there exists a set of operationally (F,l)-high threads, such as $\{V^{m_j} \mid V \in \boldsymbol{Val}\}$.
— The union of a family of sets of operationally (F,l)-high threads is a set of operationally (F,l)-high threads.

Therefore, there is a largest set of operationally (F,l)-high threads, which is the union of all sets of operationally (F,l)-high threads:

Notation 4.12. The largest set of (F,l)-high threads is denoted by $\mathscr{H}_{F,l}$.

† The notion of an 'operationally high thread' we define here should not not be confused with the notion of a 'high thread' – the latter refers to the security level associated with a thread; the former to the changes that the thread performs on the state.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

We say that a thread M^{m_j} is an *operationally* (F, l)*-high thread* if $M^{m_j} \in \mathscr{H}_{F,l}$. Note that operationally $(F, \top)$-high threads never modify the state. Furthermore, the flow policy F with respect to which we define operationally $(F, \top)$-high threads can be made more strict.

Remark 4.13. If $F' \subseteq F$, then any operationally (F, l)-high thread is also operationally (F', l)-high.

4.3.2. *Comparison with non-disclosure.* The non-disclosure for networks policy, which is restricted to networks where only one domain exists, is equivalent (up to notational issues) to the non-disclosure policy, if we only consider threads that do not contain migration instructions[†]. To see this, we can rewrite the condition for $\mathscr{R}$ to be a bisimulation in the sense of Almeida Matos and Boudol (2009), but using the language of the current paper (excluding the migration instructions):

$$P_1 \mathscr{R} P_2 \quad \text{and} \quad \langle P_1, T_1, S_1\rangle \underset{F}{\rightarrow} \langle P_1', T_1, S_1'\rangle \quad \text{and} \quad S_1 =^{G\cup F,l} S_2$$

with

$$\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \varnothing$$

$$\mathrm{img}(T_1) = \mathrm{img}(T_2) = \{d\}$$

implies

$$\exists P_2', S_2' : \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2, S_2'\rangle \quad \text{and} \quad S_1' =^{G,l} S_2' \quad \text{and} \quad P_1' \mathscr{R} P_2' \tag{24}$$

For the purposes of this comparison, we shall say that if a pool of threads P satisfies non-disclosure in the above sense, then $P \in \mathscr{ND}(G, d)$.

We say any expression M' that is attainable from M by a (possibly empty) sequence of small-step transitions is a *derivative of an expression* M,.

Definition 4.14 (Derivative of an expression). We say that an expression M' is a derivative of M if and only if $M' = M$, or there exist two states $\langle T_1, S_1\rangle$ and $\langle T_1', S_1'\rangle$ and a derivative M'' of M such that for some F, N^{n_k}, we have

$$\langle M'', T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle M', T_1', S_1'\rangle.$$

Proposition 4.15. Consider a pool of threads P whose expressions do not contain migration instructions. Then, if we consider a network with a single domain d, we have that $P \in \mathscr{NDN}(G)$ if and only if $P \in \mathscr{ND}(G, d)$.

Proof. Suppose $P \in \mathscr{NDN}(G)$. Then, for all security levels l, there exists a relation $\mathscr{S}$ that is a (G, l)-bisimulation according to Definition 4.5, and such that $P \mathscr{S} P$. Then, we have that

$$\mathscr{S}' \stackrel{\text{def}}{=} \{(Q_1, Q_2) \mid Q_1 \mathscr{S} Q_2 \ \&\ Q_1, Q_2 \text{ are derivatives of } P\} \tag{25}$$

[†] For a comparison between non-disclosure and non-interference, see Almeida Matos and Boudol (2009) and Almeida Matos (2006)

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

is also a (G,l)-bisimulation according to Definition 4.5 and $P\ \mathscr{S}'\ P$. Since P does not contain migration instructions, then no derivative of P contains any migration instructions either. Now suppose $P_1\ \mathscr{S}'\ P_2$. Then, if

$$\langle P_1, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle P_1', T_1, S_1'\rangle \tag{26}$$

and $S_1 =^{G\cup F,l} S_2$ and $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \varnothing$, we also clearly have $\langle T_1, S_1\rangle =^{G\cup F,l} \langle T_2, S_2\rangle$ and $\mathrm{img}(T_1) = \mathrm{img}(T_2) = \{d\}$. Therefore, since $\mathscr{S}'$ is a (G,l)-bisimulation according to Definition 4.5,

$$\exists T_2', P_2', S_2' \ :\ \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle \tag{27}$$

such that $\langle T_1', S_1'\rangle =^{G,l} \langle T_2', S_2'\rangle$ and $P_1'\ \mathscr{S}'\ P_2'$. Since P_2 does not contain any migration instructions, $T_2' = T_2$. Clearly, $S_1' =^{G,l} S_2'$. Therefore, $\mathscr{S}'$ is a (G,l)-bisimulation according to condition (24), where $P\ \mathscr{S}'\ P$, and we conclude that $P \in \mathscr{NDN}(G,d)$.

Now suppose $P \in \mathscr{NDN}(G,d)$. Then for all security levels l, there exists a relation $\mathscr{S}$ that is a (G,l)-bisimulation according to condition (24), and such that $P\ \mathscr{S}\ P$. Now suppose $P_1\ \mathscr{S}\ P_2$. So if

$$\langle P_1, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle P_1', T_1, S_1'\rangle \tag{28}$$

and $\langle T_1, S_1\rangle =^{G\cup F,l} \langle T_2, S_2\rangle$ and $\mathrm{dom}(S_1{}' - S_1) \cap \mathrm{dom}(S_2) = \varnothing$, we also clearly have $S_1 =^{G\cup F,l} S_2$ and $\mathrm{img}(T_1) = \mathrm{img}(T_2) = \{d\}$. Therefore, since $\mathscr{S}$ is a (G,l)-bisimulation according to condition (24), we have

$$\exists P_2', S_2' \ :\ \langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2, S_2'\rangle \tag{29}$$

where $S_1' =^{G,l} S_2'$ and $P_1'\ \mathscr{S}'\ P_2'$. It is clear that $\langle T_1', S_1'\rangle =^{G,l} \langle T_2', S_2'\rangle$, so $\mathscr{S}'$ is a (G,l)-bisimulation according to Definition 4.5, where $P\ \mathscr{S}'\ P$, and we can conclude that $P \in \mathscr{NDN}(G)$. □

It is then clear that none of the examples of locally insecure programs, when placed in a single domain, satisfy non-disclosure for networks.

5. Typing non-disclosure for networks

What new features are demanded from a type system to guarantee secure information flow in a distributed setting? In this section we address this question by presenting a type and effect system that only accepts programs satisfying non-disclosure for networks. We will begin by defining the notation used to express the typing judgments, and explain their meanings. We will then comment on the typing conditions used in the typing rules by giving examples of migration leaks that illustrate why each condition is necessary. Finally, we will give some properties of the type system, including Subject Reduction and Soundness theorems.

5.1. *A type and effect system with thread identifiers*

The type and effect system we present here selects secure threads by ensuring compliance of all information flows to the flow relation that rules at each point of the program. As

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

in Almeida Matos and Boudol (2009), it constructively approximates the *effects* of each expression, which include information on the security levels of the references on which termination or non-termination of the computations might depend.

A key observation is that here non-termination of a computation might arise from an attempt to access a *foreign* reference. In order to distinguish the threads that own each expression and reference, we associate unique identifiers $\check{m}, \check{n} \in \check{\boldsymbol{Nam}}$ to names of already existing threads, as well as to the unknown thread name '?' for those threads that are created at runtime.

It should now be clear that information on which the position of a thread n might depend can leak when another thread simply attempts to access one of n's references. For this reason, we interpret the threads' security level – in fact it represents its 'visibility' level – as a lower bound to the references that it can own, since just by owning a low reference, the position of a thread can be detected by 'low observers'. As we will see soon, the threads' security levels are used to reinforce security effects: the writing effect is updated when a thread migrates, while the termination effect is updated when a remote access is attempted.

5.1.1. *The typing judgments.* As defined in Figure 6, the judgments of the type and effect system have the form

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

The above judgment can be read as:

Under the typing context Γ the expression M has type τ and produces the security effect s when placed in a context that declares the flow policy F to extend the global flow policy G, and is part of a thread of level j that is identified by the thread name environment Σ as $\check{m}$.

The meanings of the parameters are as follows:

— The typing context $\boldsymbol{\Gamma}$ assigns types to variables.
— The expression $\boldsymbol{M}$ is a program.
— The thread identifier $\boldsymbol{\check{m}}$ identifies the thread to which the expression M belongs.
— The security level $\boldsymbol{j}$ represents a lower bound to the references that the thread owns and creates. It corresponds to the security level attributed to threads when they are created.
— The security effect $\boldsymbol{s}$ of the form $\langle s.r, s.w, s.t \rangle$, can be understood as follows:
  – $\boldsymbol{s.r}$ is the *reading effect*, an upper-bound on the security levels of the references that are read by M;
  – $\boldsymbol{s.w}$ is the *writing effect*, a lower bound on the references that are written by M;
  – $\boldsymbol{s.t}$ is the *termination effect*, an upper bound on the level of the references on which the termination of expression M might depend.

  According to these intuitions, the reading and termination levels in the type system are composed a covariantly, whereas the writing level is contravariant.
— The type $\boldsymbol{\tau}$ is the type of expression M. The syntax of types, which is given in Figure 1, is repeated here: for any type variable t,

$$\tau, \sigma, \theta \ \in \ \boldsymbol{Typ} \ ::= \ t \mid \mathsf{unit} \mid \mathsf{bool} \mid \theta \ \mathrm{ref}_{l,\check{m}_j} \mid \tau \xrightarrow[G,\check{m}_j]{s} \sigma.$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

$$
\begin{array}{lrcl}
\textit{Thread Name Environment} & \Sigma & \subseteq & ((\boldsymbol{Nam} \cup \{?\}) \times 2^{\boldsymbol{Pri}}) \times (\check{\boldsymbol{Nam}} \times 2^{\boldsymbol{Pri}}) \\
\textit{where} \quad \Sigma \downarrow_{\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}} & : & & (\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}) \to (\check{\boldsymbol{Nam}} \times 2^{\boldsymbol{Pri}}) \\
\textit{Typing Environments} & \Gamma & : & \boldsymbol{Var} \to \boldsymbol{Typ} \\
\textit{Typing Judgments} & & := & \Sigma; \Gamma \vdash^{\check{m}_j}_{G,F} M : s, \tau
\end{array}
$$

Fig. 6. Syntax of typing judgments (see also Figure 1)

The type includes annotations that are used to determine the *effects* of the expression that is being typed. To calculate them, we take into account the level of the references that are accessed and the flow policy of the context. Since we distinguish between local and foreign references, thread identifiers and security levels also appear in the types. Typable expressions that reduce to () have type unit, and those that reduce to booleans have type bool. Typable expressions that reduce to a reference belonging to process m of level j, which points to values of type θ and has security level l have the reference type $\theta\ \mathrm{ref}_{l,\check{m}_j}$ (the security levels l and j are used to determine the effects of expressions that handle references). Expressions that reduce to a function that takes a parameter of type τ and returns an expression of type σ, and with a *latent effect* s (Lucassen and Gifford 1988), where G and $\check{m}_j$, respectively, are the *flow policy* and the thread identifier where the body of the function is to be typed, have the function type $\tau \xrightarrow[G,\check{m}_j]{s} \sigma$.

— The flow policy $\boldsymbol{G}$ is the global flow policy. As we have seen, it is used to parameterise the security pre-lattice, and, in particular, the flow relation and meet/join operators. The flow relation is used in the type system to impose restrictions on the information flows that the typing rules allow, and the meet/join operators are used to construct the security effects of the expressions.

— The flow policy $\boldsymbol{F}$ is the one that is valid in the evaluation context in which M is to be typed, and contributes to the meaning of operations and relations on security levels. It is called the *flow policy of the context*. It is assumed to contain the global flow policy, which is extended with the local flow policies declared by the evaluation context (this assumption implies that in the type system, the relations $\preceq_F$ and $\preceq_{F \cup G}$ are equivalent).

— The thread name environment $\boldsymbol{\Sigma}$ is a binary relation between decorated thread names extended with '$?_k$' (where '?' represents unknown thread names), and the set of decorated thread identifiers. We define dom(Σ) as $\{n_k \mid \exists \check{n}_k \,.\, (n_k, \check{n}_k) \in \Sigma\}$. In fact, the restriction of Σ to the domain $\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}$ (written $\Sigma \downarrow_{\boldsymbol{Nam} \times 2^{\boldsymbol{Pri}}}$) is assumed to be a function, where all thread names n are distinct. The only identifiers that are images of thread names are those that correspond to threads that have already created a reference – and whose name is the prefix of that address. The others are related to $?_k$ for some security level k, which represents the thread names that are created at runtime.

We use the join operation on security effects in some of the typing rules.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

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

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

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

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

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

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

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

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

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

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

$$[\textsc{Thr}]\ \frac{j \preceq_F l \quad \check{n}\ \textit{fresh in}\ \Sigma \quad \Sigma, ?_l : \check{n}_l;\Gamma \vdash^{\check{n}_l}_{\varnothing} M : s,\mathsf{unit}}{\Sigma;\Gamma \vdash^{\check{m}_j}_{F} (\mathrm{thread}_l\ M) : \langle \bot, j \curlyvee s.w, \bot\rangle, \mathsf{unit}}$$

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

Fig. 7. Type and effect system

Definition 5.1.

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

The type and effect system is given in Figure 7, where:

— We write the flow relation with respect to the global flow policy as $\preceq$, meet $\curlywedge$ and join $\curlyvee$, instead of $\preceq_G$, $\curlywedge_G$ and $\curlyvee_G$, respectively.

— We also omit the global flow policy that appears as subscript of $\vdash^{\check{m}_j}_{G,F}$ and simply write $\vdash^{\check{m}_j}_{F}$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

— Whenever we have $\forall F, \check{m}_j \,.\, \Sigma; \Gamma \vdash_F^{\check{m}_j} M : \langle \bot, \top, \bot \rangle, \tau$, we only write $\Sigma; \Gamma \vdash M : \tau$.
— We write $l_1, \ldots, l_m \preceq_G k_1, \ldots, k_n$ instead of $\forall_{1 \leqslant i \leqslant m} \forall_{1 \leqslant j \leqslant n} \,.\, l_i \preceq_G k_j$.

The choice of including sequential composition explicitly in our higher order functional language is now justified by the specialised typing rule that provides more refined typing for that case. The same applies to the operators for reference creation, dereferencing and assignment that may be applied to expressions in general. Notice also that the type system is syntax directed.

5.2. *Typing conditions*

We must now convince ourselves that the type system does indeed only select safe threads, according to the non-disclosure for networks policy, as defined in the previous section. We will give informal justifications for each side condition constraining the typing of expressions and the construction of the security effects. For completeness, we will briefly explain the treatment of direct leaks, control leaks and higher-order leaks, and the use of the termination effect for typing away termination leaks, even though they are not central to this work. We then justify the parts of the rules that reject insecure programs that exhibit migration leaks.

5.2.1. *Direct leaks and control leaks.* The reading and writing effects are introduced by the constructs for dereferencing (see DER) and creating or updating the memory (see the typing rules REF and ASS), respectively.

Cond: The constraint $s.r \preceq s_t.w, s_f.w$ ensures that the branches N_t and N_f only make assignments to references with security level greater than the reading level of M. This prevents control leaks like the one in Example (13). To the same end, we also require the writing level of M to be kept in the effect of (thread M).

Ass: The condition $s'.r \preceq l$ prevents direct flows, as in Example (12). Furthermore, the condition $s.r \preceq l$ rules out assignments to expressions that could return different low references.

App: The condition $s''.r \preceq s'.w$ prevents direct flows from the argument of the function through an assignment occurring in its body.

Ref: The condition $s.r \preceq l$ excludes the creation of a low reference that points to the return value of an expression that performs high reads.

5.2.2. *Termination leaks.* The termination effect is introduced in conditional (COND) and application (APP) constructs. In the conclusion of COND, we add the reading level of the test to the termination level of the whole expression. This is because the conditional might choose branches with different termination behaviour depending on the references that it reads in the predicate. To see why the reading levels of both function and argument are recorded in the termination level of the application (APP), consider Examples (15) and (16), respectively, which show how the application of some argument to a dereferenced value can also unravel expressions with different termination behaviour (and thus depending on the reference that is read). Thread creation expressions (thread M) have no termination

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

effect, since their evaluation always terminates in one step. Furthermore, since a spawned thread executes in parallel with its creating thread, the reference it reads and its termination behaviour cannot influence future computations of the creating thread. Hence, its reading and termination effects are set to $\bot$.

Seq, Ass, App: The conditions $s.t \preceq s'.w$, $s.t \preceq s'.w$ and $s.t \preceq s''.w$ (respectively) prevent termination leaks similar to those in Example (14). Notice that these constraints are not as strict as 'no low write after a high read'.

Cond: The constraint $s.t \preceq s_t.w, s_f.w$ also ensures that the branches only assign to references with security level greater than the termination level of M.

Ass: The conditions $s.t \preceq l$ and $s'.t \preceq l$ prevent termination leaks that result from an assignment to a low reference or of a value (respectively) that are returned following a possibly non-terminating computation.

App: The condition $s''.t \preceq s'.w$ also rejects termination leaks resulting from the argument of the function through an assignment occurring in its body.

Ref: The condition $s.t \preceq l$ excludes termination flows resulting from the creation of a reference that points to a value returned by a possibly non-terminating computation.

5.2.3. *Higher-order leaks.*

App: The condition $s.r \preceq s'.w$ excludes expressions that obtain a function with a low latent write effect from a high reference, and then unravel this low write effect by applying it to some argument.

5.2.4. *Migration leaks.* The management of migration leaks is supported by the features of the type system that keep track of the location of threads and references.

In rule Loc, since the name of the thread that owns the reference is given in the prefix, the corresponding thread identifier is found using Σ. In rule Ref, the reference created belongs to the thread identified by the superscript of the '$\vdash$'. We check that the security level declared for the new reference is greater than the level of the thread.

The body of an abstraction (rule Abs) is executed by the thread that applies it to an argument (see App) in the same flow context of that application. This is why the thread identifier and flow context of its execution are latent.

In rule Thr, a fresh identifier – image of an unknown thread name represented by '?' – is used to type the thread that is created. When a runtime thread is created by another runtime thread, the domain of Σ used to type the nested threads contains more than one entry using ?. The reason the value of ? cannot be overwritten when typing nested thread creations is that we must keep a full record of the image of Σ in order to guarantee that new thread identifiers attributed by the rule Thr are fresh. As we will see soon, these are used mainly to distinguish accesses to local references from accesses to foreign references (which are potentially remote).

We have seen in Section 2.1 that termination leaks appear when a change to the low state depends on the termination of a computation that precedes it, which in turn depends on high information. Suspension of a thread on an access to an absent reference can be viewed as a non-terminating computation that can be unblocked by migration of

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

concurrent threads. Therefore, we can deal with migration leaks in similar ways to our treatment of termination leaks.

Example (19) shows how high information can leak by means of a thread (n) that is located in a domain that is different from where low assignments are performed (by m_1 and m_2). The key point in this example is that the synchronisation between the two threads is made through the migration of n to a domain where m_1 or m_2 is located, at a time where the low assignments that are bound to occur in m_1 and m_2 are blocked by a suspension on an access to one of n's references.

From the point of view of thread n in Example (19), we cannot know whether, in the domains it might migrate to, there are threads that are suspended on its arrival. Assuming the worst case for thread n, that is, that there are indeed other threads suspended on accesses to n's lowest references, the type system updates the writing effect of the migration instruction in rule MIG with the security level of n, which is a lower bound k to the level of all its references. As a consequence, the rule COND of the type system imposes the condition $H \preceq k$ over thread n in a standard manner. In general terms, as long as a thread is typable, one can ensure that the level of the information on which its migration depends is lower than or equal to its own security level.

Now, from the point of view of threads m_1 and m_2 (still in Example (19)), we cannot know whether the arrival of the thread that might unblock their computations depends on high information or not. Again assuming the worst case for threads m_1 and m_2, that is, that the arrival of the owner (of level k) of the foreign references that m_1 and m_2 want to access does indeed carry information of level k, rules DER and ASS of the type system update the termination effect of m_1 and m_2 with k. As a consequence, the rule SEQ of the type system imposes the condition $k \preceq L$ over threads m_1 and m_2 in a standard manner. In general, as long a thread is typable, we can ensure that any blocked low assignments are not lower than the accesses that are causing the suspension.

Example (19) is thus rejected for the reason that conditions $H \preceq k$ and $k \preceq L$ are not compatible. In this way, we prevent migrations of threads that own low references from depending on high information. Similarly, in rule THR, by adding the security level of the thread to the write effect of the migration construct, we prevent the creation of low threads from depending on high information.

In Examples (21) and (22), the low assignment can only occur if the threads m and n are located in the same domain. Therefore, the position of m might also be leaked when the low assignment occurs. This is why we also update the termination level of the assignment (ASS) and the dereference (DER) with m's security level.

Ref: The condition $j \preceq l$ ensures that the references created by a thread respect the security level of the thread, that is, that they are not lower (with respect to the global flow policy) than it.

Ass: The insecure program in Example (23) is rejected by the condition $j \preceq_G l$ in rule ASS. Similarly, the program in Example (20) is rejected if j_1 or $j_2 \not\preceq L$ to prevent the leak of information about the positions of m_1 and m_2. Notice that in the typing rule for the cases where $\check{m} = \check{n}$ (and therefore $j = k$), which correspond to local assignments,

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

the condition $j \preceq_G l$ is always satisfied due to the fact that by assumption $k \preceq l$ always holds for references of type $\theta\ \mathrm{ref}_{l,\check{n}_k}$.

Thr: The condition $j \preceq_F l$ rejects the insecure program

$$d[\ (\mathrm{thread}_L\ M)^{m_H}\ ]. \tag{30}$$

The reason this program is considered insecure is that the presence of the high thread m, which should only be 'visible' at level H, is indicated at the level L at which the created thread is apparent.

5.2.5. *Expressivity of declassification.* There is a subtlety in the allowed usage of flow declarations that differs from what is allowed in Almeida Matos and Boudol (2009). Consider the program

$$(b_L :=^? (\mathrm{flow}\ H \prec L\ \mathrm{in}\ (?\ a_H))), \tag{31}$$

which is safe according to our notion of non-disclosure for networks. This program is not typable because the assignment is not performed inside the flow declaration. Another example is

$$((\mathrm{flow}\ H \prec L\ \mathrm{in}\ (\mathrm{if}\ (?\ a_H)\ \mathrm{then}\ ()\ \mathrm{else}\ \mathsf{loop}));(b_L :=^? 0)), \tag{32}$$

where information about H is allowed to be downgraded to level L, and thus can be transmitted through the termination behaviour of the conditional and possibly stored in the low level reference b. Notice that the above programs would be accepted if the flow declaration encompassed the whole assignment, in the case of Example (31), and the whole sequential composition, in the case of Example (32).

We could have used, as in Almeida Matos and Boudol (2009), a kind of subsumption on the security effect in the FLOW rule to make the above examples be accepted (see Almeida Matos (2006) for a discussion on this point). This would allow us to mimic the declassify (M, l) operator used in some languages (see Myers (1999) and Sabelfeld and Myers (2004) for instance) for downgrading the value of M to lower confidentiality levels. By eliminating subsumption from our type system, we choose here to restrict consideration purely to the concept of a flow declaration that enables declassification operations. In this way, we underline the particular style of declassification introduced by the flow declarations, as opposed to approaches that aim at downgrading values.

Another more technical difference between the type system presented here and the one in Almeida Matos and Boudol (2009) is the fact that in the latter the security effects are built using the extended flow relation, while here they are built with respect to the global flow policy. Using the extended flow relation, the security effects are 'weaker' – in other words, more precise. However, this does not necessarily imply greater refinement of the type system since the restrictions imposed on those security effects are taken with respect to the extended flow relation. In fact, one may conjecture that the weaker security effects of Almeida Matos and Boudol (2009) could be simplified in the same way as here, and without loss of generality.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

5.3. *Properties of typed expressions*

We now state and prove the main properties of the work described in this paper. We will prove the soundness of the type system with respect to our security property, that is that networks of typable processes are secure with respect to the non-disclosure for networks property. In order to prove this, we will verify some intermediate results, including subject reduction. In the detailed proof that follows, we highlight the differences with respect to Almeida Matos and Boudol (2009).

5.3.1. *Meaning of effects.* Unlike the effects given by the type systems in Almeida Matos and Boudol (2009), the reading effect of a typable expression for the type systems of the current paper is not always upward bounded by its termination effect. In this context, the termination of an expression does not just depend on the existence of non-terminating loops (which in turn depend on tested values), but also on the possibility of suspension on accesses to foreign references (which depend on the relative position of threads in the network). We can check that the intuitive meaning of the effects is indeed captured by our type system.

Lemma 5.2 (Update of effects).

(1) If $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(?\ n_k.u_{l,\theta})] : s,\tau$, then $l \preceq s.r$. Also, if $m \neq n$, then $k \curlyvee j \preceq s.t$.
(2) If $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(n_k.u_{l,\theta} :=^? V)] : s,\tau$, then $s.w \preceq l$. Also, if $m \neq n$, then $k \curlyvee j \preceq s.t$.
(3) If $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)] : s,\tau$, then $s.w \preceq l$.
(4) If $\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}[(\mathrm{goto}\ d)] : s,\tau$, then $s.w \preceq j$.

Proof. The proof is by induction on the structure of E. □

5.3.2. *Subject reduction.* In order to establish the soundness of the type system of Figure 7 we need a Subject Reduction result stating that the typing of expressions is preserved by computation. We will follow the usual proof steps (Wright and Felleisen 1994) in detail, but note first that a value, and more generally a pseudo-value, has no effect, and that this is properly reflected in the type system. Moreover, the typing of a pseudo-value does not depend on the thread identifier or current flow policy.

Remark 5.3. If $W \in \boldsymbol{Pse}$ and $\Sigma;\Gamma \vdash_F^{\check{m}_j} W : s,\tau$, then for all thread identifiers $\check{n}_k$ and flow policies F', we have $\Sigma;\Gamma \vdash_{F'}^{\check{n}_k} W : \langle\bot,\top,\bot\rangle,\tau$.

The following lemma establishes some standard weakening and strengthening properties.

Lemma 5.4.

(1) If $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$ and $x \notin \mathrm{dom}(\Gamma)$, then $\Sigma;\Gamma,x:\sigma \vdash_F^{\check{m}_j} M : s,\tau$.
(2) If $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$ and $\check{n}$ fresh in Σ, then $\Sigma,?_k : \check{n}_k;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$.
(3) If $\Sigma;\Gamma,x:\sigma \vdash_F^{\check{m}_j} M : s,\tau$ and $x \notin \mathrm{fv}(M)$, then $\Sigma;\Gamma \vdash_F^{\check{m}_j} M : s,\tau$.
(4) If $\Sigma;\Gamma \vdash_G^{\check{m}_j} M : s,\tau$, then $\Sigma;\Gamma \vdash_{G\cup F}^{\check{m}_j} M : s,\tau$.

Proof. The proof is by induction on the inference of the type judgment. □

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

We now prove two final preliminary lemmas, which state that substitutions and replacements in contexts preserve types.

Lemma 5.5 (Substitution). If $\Sigma; \Gamma, x : \sigma \vdash_F^{\check{m}_j} M : s, \tau$ and $\Sigma; \Gamma \vdash W : \sigma$, then $\Sigma; \Gamma \vdash_F^{\check{m}_j} \{x \mapsto W\}M : s, \tau$.

Proof. The proof is by induction on the inference of $\Sigma; \Gamma, x : \tau \vdash_F^{\check{m}_j} M : s, \sigma$, and, using the previous lemma, by case analysis on the last rule used in this typing proof. We will just present the cases of the THR and FLOW typing rules:

Thr: We have $M = (\text{thread}_k\ \bar{M})$ and for $\check{n}$ fresh in Σ, we have $\Sigma, ?_k : \check{n}_k; \Gamma, x : \sigma \vdash_{\varnothing}^{\check{n}_k} \bar{M} : s, \tau$, with $\tau = \mathsf{unit}$ and $s = \langle \bot, s.w, \bot \rangle$. Using the assumption and Lemma 5.4, we have $\Sigma, ?_k : \check{n}_k; \Gamma \vdash W : \sigma$. By the induction hypothesis, we then have $\Sigma, ?_k : \check{n}_k; \Gamma \vdash_{\varnothing}^{\check{m}_j} \{x \mapsto W\}\bar{M} : s, \tau$. Therefore, by rule THR, we have $\Sigma; \Gamma \vdash_F^{\check{m}_j} (\text{thread}_k\ \{x \mapsto W\}\bar{M}) : s, \tau$.

Flow: We have $M = (\text{flow}\ \bar{F}\ \text{in}\ \bar{M})$ and $\Sigma; \Gamma, x : \sigma \vdash_{F \cup \bar{F}}^{\check{m}_j} \bar{M} : s, \tau$. By the induction hypothesis, $\Sigma; \Gamma \vdash_{F \cup \bar{F}}^{\check{m}_j} \{x \mapsto W\}\bar{M} : s, \tau$. Then, by FLOW, we have $\Sigma; \Gamma \vdash_F^{\check{m}_j} (\text{flow}\ \bar{F}\ \text{in}\ \{x \mapsto W\}\bar{M}) : s, \tau$. □

Lemma 5.6 (Replacement). If $\Sigma; \Gamma \vdash_F^{\check{m}_j} \mathrm{E}[M] : s, \tau$ is a valid judgment, then the proof gives M a typing $\Sigma; \Gamma \vdash_{F \cup \lceil \mathrm{E} \rceil}^{\check{m}_j} M : \bar{s}, \bar{\tau}$ for some $\bar{s}$ and $\bar{\tau}$ such that

$$\bar{s}.r \preceq s.r$$
$$s.w \preceq \bar{s}.w$$
$$\bar{s}.t \preceq s.t.$$

Furthermore, if

$$\Sigma; \Gamma \vdash_{F \cup \lceil \mathrm{E} \rceil}^{\check{m}_j} N : \bar{s}', \bar{\tau}$$

with

$$\bar{s}'.r \preceq \bar{s}.r$$
$$\bar{s}.w \preceq \bar{s}'.w$$
$$\bar{s}'.t \preceq \bar{s}.t,$$

then

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} \mathrm{E}[N] : s', \tau,$$

for some s' such that

$$s'.r \preceq s.r$$
$$s.w \preceq s'.w$$
$$s'.t \preceq s.t.$$

Proof. The proof is by induction on the structure of E. We will just present the case for the flow declaration context. If $\mathrm{E}[M] = (\text{flow}\ F'\ \text{in}\ \bar{\mathrm{E}}[M])$, then, by FLOW, we have

$$\Sigma; \Gamma \vdash_{F \cup F'}^{\check{m}_j} \bar{\mathrm{E}}[M] : s, \tau.$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

By the induction hypothesis, the proof gives M a typing

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

for $\hat{F}, \hat{s}, \hat{\tau}$ with

$$\hat{F} = F \cup F' \cup \lceil \bar{\mathrm{E}} \rceil$$

and $\hat{s}.r \preceq s.r$, $s.w \preceq \hat{s}.w$ and $\hat{s}.t \preceq s.t$.

Also by the induction hypothesis,

$$\Sigma; \Gamma \vdash^{\check{m}_j}_{\hat{F}} \bar{\mathrm{E}}[N] : s', \tau,$$

for some s' such that $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$. Then, again by FLOW, we have

$$\Sigma; \Gamma \vdash^{\check{m}_j}_{F} (\text{flow } F' \text{ in } \bar{\mathrm{E}}[N]) : s', \tau. \qquad \square$$

Finally, we will now prove Subject Reduction, which states that computation preserves the type of threads, and that as the effects of an expression are performed, the security effects of the thread 'weaken'. We assume that the value contained in references that are given type θ do indeed have type θ. The differences with respect to the Subject Reduction in Almeida Matos and Boudol (2009) lie mainly in the fact that we do not have subsumption, and in the treatment of thread names. In particular, we ensure that when a thread is created, it is typable with respect to a fresh thread identifier in an environment where Σ is updated accordingly.

Theorem 5.7 (Subject reduction). If for some $\Sigma, \Gamma, s, \tau, F, m_j$ we have

$$\vdash^{\Sigma(m_j)}_{F} M : s, \tau$$

and

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

where all $a_{l,\theta} \in \mathrm{dom}(S)$ satisfy $\Sigma; \Gamma \vdash S(a_{l,\theta}) : \theta$, then $\exists s'$ such that

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F} M' : s', \tau,$$

where $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$. Furthermore, we have $\exists \check{n}, s''$ such that

$$\Sigma, ?_k : \check{n}_k; \Gamma \vdash^{\check{n}_k}_{\varnothing} N : s'', \mathsf{unit}$$

where $\check{n}$ is fresh in Σ, and $s.w \preceq s''.w$.

Proof. Suppose $M = \bar{\mathrm{E}}[\bar{M}]$ and

$$\langle \bar{M}^{m_j}, T, S \rangle \xrightarrow[\bar{F}]{\bar{N}^{n_k}} \langle \bar{M}'^{m_j}, \bar{T}', \bar{S}' \rangle.$$

We start by observing that this implies

$$F' = \bar{F} \cup \lceil \bar{\mathrm{E}} \rceil$$
$$M' = \bar{\mathrm{E}}[\bar{M}']$$
$$\bar{N}^{n_k} = N^{n_k}$$
$$\langle \bar{T}', \bar{S}' \rangle = \langle T', S' \rangle.$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

We can assume, without loss of generality, that $\bar{M}$ is the smallest in the sense that there is no $\hat{\mathrm{E}}, \hat{M}, \hat{N}$ such that $\hat{\mathrm{E}} \neq []$ and $\hat{\mathrm{E}}[\hat{M}] = \bar{M}$, for which we can write

$$\langle \hat{M}^{m_j}, T, S\rangle \xrightarrow[\hat{F}]{\hat{N}^{n_k}} \langle \hat{M}'^{m_j}, T', S'\rangle.$$

By Lemma 5.6, we have

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F \cup \lceil \bar{\mathrm{E}} \rceil} \bar{M} : \bar{s}, \bar{\tau}$$

in the proof of

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F} \bar{\mathrm{E}}[\bar{M}] : s, \tau,$$

for some $\bar{s}$ and $\bar{\tau}$. We then proceed by case analysis on the transition

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

and prove that

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F \cup \lceil \bar{\mathrm{E}} \rceil} \bar{M}' : \bar{s}', \bar{\tau}$$

for some $\bar{s}'$ such that $\bar{s}'.r \preceq \bar{s}.r$, $\bar{s}.w \preceq \bar{s}'.w$ and $\bar{s}'.t \preceq \bar{s}.t$. We will only consider two representative cases. If $\bar{M} = ((\lambda x.\hat{M})\ V)$ (and similarly if $\bar{M} = (\varrho x.W)$), we use Lemma 5.5. If $\bar{M} = (\text{flow } F' \text{ in } V)$, then, by rule FLOW, we have

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F \cup \lceil \bar{\mathrm{E}} \rceil \cup F'} V : s, \tau,$$

and by Remark 5.3, we have

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F \cup \lceil \bar{\mathrm{E}} \rceil} V : \langle \bot, \top, \bot\rangle, \bar{\tau}.$$

Again by Lemma 5.6, we can now conclude that

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F} \bar{\mathrm{E}}[\bar{M}'] : s', \tau$$

for some s' such that $s'.r \preceq s.r$, $s.w \preceq s'.w$ and $s'.t \preceq s.t$.

Finally, if $N^{n_k} \neq ()$ (N^{n_k} is created), then

$$\exists l, \hat{N} : M = \bar{\mathrm{E}}[(\text{thread}_l\ \hat{N})]$$

and $\bar{N} = \hat{N}$. By Lemma 5.6, we have

$$\Sigma; \Gamma \vdash^{\Sigma(n_k)}_{F \cup \lceil \bar{\mathrm{E}} \rceil} (\text{thread}_l\ \hat{N}) : \hat{s}, \mathsf{unit}$$

in the proof of

$$\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{F} \bar{\mathrm{E}}[(\text{thread}_l\ \hat{N})] : s, \tau,$$

for some $\hat{s}$, and $\hat{\tau}$. By THR, for some $\check{n}$ fresh in Σ we have

$$\Sigma, ?_k : \check{n}_k; \Gamma \vdash^{\check{n}_k}_{\varnothing} \hat{N} : \hat{s}, \mathsf{unit},$$

where $\hat{s} = \langle \bot, s.w, \bot\rangle$. □

5.3.3. *Syntactically high expressions.* Some expressions can be easily classified as 'high' by the type system simply because their code does not contain any instruction that could affect the 'low' part of the state. According to Lemma 5.2, this is the case for expressions

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

with a high writing effect, which can be said to be *syntactically high* with respect to a security level in the context of a flow policy and thread name.

Definition 5.8 (Syntactically 'high' expressions). An expression M is *syntactically* (F, l, m_j)-*high* if there exist Σ, Γ, s, and τ such that

$$\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} M : s, \tau$$

with $s.w \not\preceq_F l$. The expression M is a *syntactically* (F, l, m_j)-*high function* if there exist Σ, Γ, s, τ and σ such that

$$\Sigma; \Gamma \vdash M : \tau \xrightarrow[F,\Sigma(m_j)]{s} \sigma$$

with $s.w \not\preceq_F l$.

We are now able to prove that syntactically high expressions have an operationally high behaviour.

Lemma 5.9 (High expressions). If M is a syntactically (F, l, m_j)-high expression, then M^{m_j} is an operationally (F, l)-high thread.

Proof. We show that if M is syntactically (F, l, m_j)-high, that is, if there exist Σ, Γ, s and τ such that

$$\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} M : s, \tau$$

with $s.w \not\preceq_F l$, and

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

then $S' =^{F,l} S$. This is enough to prove the lemma since, by Theorem 5.7 (Subject reduction), we have both M' is syntactically (F, l, m_j)-high and N is syntactically (F, l, n_k)-high. We proceed by cases on the proof of the transition

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

and show the cases where $\langle T, S\rangle \neq \langle T', S'\rangle$:

$M = \mathbf{E}[(\mathbf{ref}_{\bar{l},\bar{\theta}}\ V)]$:
In this case $S' = \{a_{\bar{l},\bar{\theta}} \mapsto V\} \cup S$ where a is fresh for S, and $s.w \preceq \bar{l}$ by Lemma 5.2. This implies $\bar{l} \not\preceq_F l$, hence $S' =^{F,l} S$.

$M = \mathbf{E}[(a_{\bar{l},\bar{\theta}} :=^? V)]$:
In this case $S' = [a_{\bar{l},\bar{\theta}} := V]S$ and $s.w \preceq \bar{l}$ by Lemma 5.2. This implies $\bar{l} \not\preceq_F l$, hence $S' =^{F,l} S$.

$M = \mathbf{E}[(\mathbf{goto}\ d)]$:
In this case $T' = [m_j := d]T$ and $s.w \preceq j$ by Lemma 5.2. This implies $j \not\preceq_F l$, hence $T' =^{F,l} T$. □

5.4. *Soundness*

In this section we follow a proof methodology that has been applied in a series of increasingly complex concurrent settings (Boudol and Castellani 2002; Almeida Matos

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

and Boudol 2009; Boudol 2005b; Almeida Matos 2006), and which we believe could be used in others as well. We therefore present and explain in detail the proofs, providing a 'Rationale' that summarises the intuition behind the proof for each intermediate result. In particular, the conditions of the typing rules that are used in the proof are pointed out – see Subsection 5.2 for examples justifying the need for these conditions.

We prove soundness of the type system of Figure 7 with respect to the notion of security given by Definition 4.8, that is, that under any global flow policy G, all typable sets of threads P satisfy non-disclosure for networks. Informally, this means that whatever the security level that is chosen to be 'low' (here that security level will be denoted by '*low*'), the set P always presents the same behaviour according to a *weak* bisimulation on low-equal states: if two continuations P_1 and P_2 of P are related, and if P_1 can perform an execution step over a certain state, then P_2 can perform the same low changes to any low-equal state in zero or one step, while the two resulting continuations are still related. It is useful to start by analysing the behaviour of the class of expressions that are typable with a low termination effect, for which we can state a stronger soundness result.

5.4.1. *Behaviour of 'low'-terminating expressions.* Recall that according to the intended meaning of the termination effect, the termination or non-termination of expressions with low termination effect should only depend on the low part of the state. In other words, two computations of the same thread running under two 'low'-equal states should either both terminate or both diverge. In particular, this implies that the termination behaviour of these expressions cannot be used to leak 'high' information when composed with other expressions (through termination leaks).

The ability of a thread to compute depends on whether its position in the network is the same as that of the references that it needs to access. This means that in order to guarantee that a step is performed by a thread in two different states, one must assume that it does not suspend on an access to an absent reference. The following guaranteed-transition result holds for low-equal states where, if the thread is about to access a reference, then either the thread owns that reference, or both the thread and the reference have a low security level.

Lemma 5.10 (Guaranteed transitions). Suppose M is typable for Σ, $\Sigma(m_j)$, F, and that if $M = \mathrm{E}[(n_k.u_{l,\theta} :=^{?} V)]$ or $M = \mathrm{E}[(?\, n_k.u_{l,\theta})]$, then either $j \curlyvee k \preceq_F \mathit{low}$ or $n = m$.

If

$$\langle M^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle$$

such that $\bar{n}_{\bar{k}}$ is fresh for T_2 if $\bar{n}_{\bar{k}} \in \mathrm{dom}(T_1' - T_1)$ and a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$ and that for some F' we have

$$\langle T_1, S_1\rangle =^{F \cup F', \mathit{low}} \langle T_2, S_2\rangle,$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

then there exist M_2', T_2' and S_2' such that

$$\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_k}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$$

with

$$\langle T_1', S_1'\rangle =^{F\cup F', low} \langle T_2', S_2'\rangle.$$

Rationale. When a typable thread m_j is about to perform an assignment or dereference of a reference belonging to a thread n_k, the execution of this operation depends on m_j and n_k being located at the same domain. By assuming that either both j and k are low, or m and n are the same thread, we can conclude that, in low-equal states, m_j and n_k have the same location. Therefore, if M performs a transition in some state, it is able to perform it in a low-equal state as well.

When a thread n_k is created by m_j, we use the condition $j \preceq_F k$ of rule Thr to ensure that either k is high (and therefore its creation does not change the low state) or m_j is a low thread (therefore n_k is created at the same place in two low-equal memories).

Proof. The proof is by case analysis on the proof of

$$\langle M^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{\bar{n}_k}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle.$$

In most cases, this transition does not modify or depend on the state $\langle T_1, S_1\rangle$, and we may let $M_2' = M_1'$ and $\langle T_2', S_2'\rangle = \langle T_2, S_2\rangle$.

$M = \mathrm{E}[(\mathrm{ref}_{l,\theta}\ V)]$:
In this case $M_1' = \mathrm{E}[m_j.u_{l,\theta}]$, $F = \lceil \mathrm{E}\rceil$, $N^{\bar{n}_k} = \emptyset$, $T_1' = T_1$ and $S_1' = S_1 \cup \{m_j.u_{l,\theta} \mapsto V\}$. Since $m_j.u$ is fresh for S_2, we also have

$$\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_k}} \langle {M_1'}^{m_j}, T_2, S_2' \cup \{m_j.u_{l,\theta} \mapsto V\}\rangle.$$

$M = \mathrm{E}[(?\ n_k.u_{l,\theta})]$:
In this case $M_1' = \mathrm{E}[S_1(n_k.u_{l,\theta})]$, $F = \lceil \mathrm{E}\rceil$, $N^{\bar{n}_k} = \emptyset$, and $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$. We have

$$\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_k}} \langle \mathrm{E}[S_2(n_k.u_{l,\theta})]^{m_j}, T_2, S_2\rangle,$$

because $T_1 =^{F\cup F', low} T_2$ and either:

(i) $m = n$, in which case M^{m_j} cannot suspend.

(ii) $m \neq n$ and $j \vee k \preceq_F low$, in which case $T_1(m_j) = T_2(m_j)$ and $T_1(n_k) = T_2(n_k)$. Since $T_1(m_j) = T_1(n_k)$, we have $T_2(m_j) = T_2(n_k)$.

In other words, in T_2 the threads m_j and n_k are also located in the same domain.

$M = \mathrm{E}[(n_k.u_{l,\theta} :=^{?} V)]$:
In this case $M_1' = \mathrm{E}[\emptyset]$, $F = \lceil \mathrm{E}\rceil$, $N^{\bar{n}_k} = \emptyset$, $T_1' = T_1$ and $S_1' = [n_k.u_{l,\theta} := V]S_1$. Analogously to the previous case, the threads m_j and n_k in T_2 are located in the same

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

domain, so

$$\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle \mathrm{E}[\langle\rangle]^{m_j}, T_2, [n_k.u_{l,\theta} := V]S_2\rangle.$$

***M* = E[(thread$_{\bar{k}}$ $\bar{M}$)]:**
In this case $M_1' = \mathrm{E}[\langle\rangle]$, $F = \lceil \mathrm{E} \rceil$, $N^{\bar{n}_{\bar{k}}} = \bar{M}^{\bar{n}_{\bar{k}}}$, $T_1' = T_1 \cup \{\bar{n}_{\bar{k}} \mapsto T_1(m_j)\}$, and $S_1' = S_1$. Since n is fresh for T_2, we have

$$\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle \mathrm{E}[\langle\rangle]^{m_j}, T_2 \cup \{\bar{n}_{\bar{k}} \mapsto T_2(m_j)\}, S_2\rangle.$$

Notice that

$$T_1 \cup \{\bar{n}_{\bar{k}} \mapsto T_1(m_j)\} =^{F\cup F', low} T_2 \cup \{\bar{n}_{\bar{k}} \mapsto T_2(m_j)\}$$

because $T_1 =^{F\cup F', low} T_2$ and if $\bar{k} \preceq_{F\cup F'} low$, then, by the condition $j \preceq_{F\cup F'} \bar{k}$ in rule Thr, we also have $j \preceq_{F\cup F'} low$, in which case $T_1(m_j) = T_2(m_j)$.

***M* = E[(goto *d*′)]:**
In this case $M_1' = \mathrm{E}[\langle\rangle]$, $F = \lceil \mathrm{E} \rceil$, $N^{\bar{n}_{\bar{k}}} = \langle\rangle$, $T_1' = [m_j := d']T_1$ and $S_1' = S_1$. We have

$$\langle M^{m_j}, T_2, S_2\rangle \xrightarrow[F]{N^{\bar{n}_{\bar{k}}}} \langle \mathrm{E}[\langle\rangle]^{m_j}, [m_j := d']T_2, S_2\rangle.$$

□

The hypotheses of the above lemma are fulfilled when the termination effect is low.

Remark 5.11. Suppose $M = \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ or $M = \mathrm{E}[(?\, n_k.u_{l,\theta})]$, and that for some Σ, m_j, F, s and τ we have $\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} M : s, \tau$. Then, $s.t \preceq_F low$ implies that either $j \vee k \preceq_F low$ or $n = m$.

We now prove that any two computations under low-equal states of a typable thread that has a low-termination effect have the same 'length', and, in particular, that they are either both finite or both infinite. To this end, we design a reflexive binary relation on expressions with low-termination effects that is closed under the transitions of Lemma 5.10 (Guaranteed transitions). The definition of $\mathscr{T}^{m_j}_{G,F,low}$, which is abbreviated to $\mathscr{T}^{m_j}_{F,low}$ when the global flow policy is G, is given in Definition 5.12. The flow policy F is assumed to contain G. Notice that it is a symmetric relation. In order to ensure that expressions related by $\mathscr{T}^{m_j}_{F,low}$ perform the same changes to the low memory, its definition requires that the references created or written using (potentially) different values are high.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Definition 5.12 ($\mathscr{T}^{m_j}_{F,low}$). We have

$$M_1 \ \mathscr{T}^{m_j}_{F,low} \ M_2$$

if

$$\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_1 : s_1, \tau$$
$$\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_2 : s_2, \tau$$

for some Σ, Γ, s_1, s_2 and τ with $s_1.t \preceq_F low$ and $s_2.t \preceq_F low$ and one of the following holds:

Clause 1: M_1 and M_2 are both values; or
Clause 2: $M_1 = M_2$; or
Clause 3: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{M}_2$; or
Clause 4: $M_1 = (\mathrm{ref}_{l,\theta} \ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta} \ \bar{M}_2)$ with $\bar{M}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{M}_2$, and $l \not\preceq_F low$; or
Clause 5: $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ with $\bar{M}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{M}_2$; or
Clause 6: $M_1 = (\bar{M}_1 :=^{?} \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^{?} \bar{N}_2)$ with both $\bar{M}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{M}_2$ and $\bar{N}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{N}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta \ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$; or
Clause 7: $M_1 = (\mathrm{flow}\ F' \ \mathrm{in}\ \bar{M}_1)$ and $M_2 = (\mathrm{flow}\ F' \ \mathrm{in}\ \bar{M}_2)$ with $\bar{M}_1 \ \mathscr{T}^{m_j}_{F \cup F', low} \ \bar{M}_2$.

Remark 5.13. If for some m_j, F and low we have $M_1 \ \mathscr{T}^{m_j}_{F,low} \ M_2$ and $M_1 \in \boldsymbol{Val}$, then $M_2 \in \boldsymbol{Val}$.

We have seen in Lemma 3.2 (Splitting computations) that two computations of the same expression can split only if the expression is about to read a reference that is given different values by the memories in each of the configurations. Since we will only be interested in the case where the two memories are low-equal, this situation coincides with the case where the reference that is read is high. From the following lemma one can conclude that the relation $\mathscr{T}^{m_j}_{F,low}$ relates the possible outcomes of expressions that are typable with a low termination effect and that perform a high read over low-equal memories.

Lemma 5.14. If there exist Σ, Γ, s, τ such that $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} \mathrm{E}[(?\ a_{l,\theta})] : s, \tau$ with $s.t \preceq_F low$ and $l \not\preceq_{F \cup [\mathrm{E}]} low$, then for any values $V_0, V_1 \in \boldsymbol{Val}$ such that $\Sigma;\Gamma \vdash V_i : \theta$, we have $\mathrm{E}[V_0] \ \mathscr{T}^{m_j}_{F,low} \ \mathrm{E}[V_1]$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Rationale. If a typable expression is about to use a value that results from a high dereference in such a way that it could influence its termination behaviour, then its termination effect cannot be low (contradicting the assumption). The type system enforces this by updating the termination effect of the expression with the reading effect of the dereferencing operation in the cases where the value is used:

— in the predicate of a conditional ($s.r$ in the termination effect of Cond);
— to determine the function of an application ($s.r$ in the termination effect of App);
— to determine the argument of an application ($s''.r$ in the termination effect of App).

The relation $\mathscr{T}$ requires that the references that are created or written using the high dereferenced value are high (see Clauses 4 and 6, respectively). This is guaranteed by conditions of the form '$s.r \preceq_F l$', where s is the security effect of the program that is performing the access, and l is the security level of the reference that is created or written. More precisely, conditions are imposed when the dereferenced value is used to:

— create a reference ($s.r \preceq_F l$ in rule Ref);
— determine a reference that is being assigned to ($s.r \preceq_F l$ in rule Ass);
— determine a value that is being assigned ($s'.r \preceq_F l$ in rule Ass).

Proof. The proof is by induction on the structure of E:

$\mathrm{E}[(?\ a_{l,\theta})] = (?\ a_{l,\theta})$:
We have $V_0 \ \mathscr{T}^{m_j}_{F,low}\ V_1$ by Clause 1.

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathrm{E}_1[(?\ a_{l,\theta})]\ M)$:
By App, we have

$$\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\tau} \xrightarrow[F,\Sigma(m_j)]{\bar{s}'} \bar{\sigma}$$

with $\bar{s}.r \preceq s.t$. By Lemma 5.2, we have $l \preceq \bar{s}.r$. Therefore $l \preceq_F s.t$, which contradicts the assumption that both $s.t \preceq_F low$ and $l \not\preceq_{F \cup \lceil \mathrm{E} \rceil} low$ hold.

$\mathrm{E}[(?\ a_{l,\theta})] = (V\ \mathrm{E}_1[(?\ a_{l,\theta})])$:
By rule App, we have

$$\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}'', \bar{\tau}$$

with $\bar{s}''.r \preceq s.t$. By Lemma 5.2, we have $l \preceq \bar{s}''.r$. Therefore $l \preceq_F s.t$, which contradicts the assumption that both $s.t \preceq_F low$ and $l \not\preceq_{F \cup \lceil \mathrm{E} \rceil} low$ hold.

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathbf{if}\ \mathrm{E}_1[(?\ a_{l,\theta})]\ \mathbf{then}\ M_t\ \mathbf{else}\ M_f)$:
By Cond, we have

$$\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \mathsf{bool}$$

with $\bar{s}.r \preceq s.t$. By Lemma 5.2, we have $l \preceq \bar{s}.r$. Therefore $l \preceq_F s.t$, which contradicts the assumption that both $s.t \preceq_F low$ and $l \not\preceq_{F \cup \lceil \mathrm{E} \rceil} low$ hold.

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathrm{E}_1[(?\ a_{l,\theta})]; M)$:
By Seq, we have

$$\Sigma; \Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\tau}$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

with $\bar{s}.t \preceq_F s.t$. Therefore $\bar{s}.t \preceq_F \mathit{low}$, and since $l \not\preceq_{F\cup\lceil \mathrm{E}\rceil} \mathit{low}$ implies $l \not\preceq_{F\cup \mathrm{E}_1} \mathit{low}$, by the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,\mathit{low}}\ \mathrm{E}_1[V_1]$. Lemma 5.6 and Clause 3 then give the required result.

$\mathbf{E[(?\ a_{l,\theta})] = (\mathrm{ref}_{l',\theta'}\ E_1[(?\ a_{l,\theta})])}$:
By rule Ref, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\tau}$$

with $\bar{s}.r = s.r \preceq_F l'$ and $\bar{s}.t = s.t$. Therefore $\bar{s}.t \preceq_F \mathit{low}$, and since $l \not\preceq_{F\cup\lceil \mathrm{E}\rceil} \mathit{low}$ implies $l \not\preceq_{F\cup \mathrm{E}_1} \mathit{low}$, by the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,\mathit{low}}\ \mathrm{E}_1[V_1]$. By Lemma 5.2, we have $l \preceq s.r$, so $s.r \not\preceq_F \mathit{low}$. Therefore, $l' \not\preceq_F \mathit{low}$, and Lemma 5.6 and Clause 4 then give the required result.

$\mathbf{E[(?\ a_{l,\theta})] = (?\ E_1[(?\ a_{l,\theta})])}$:
By rule Der, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\tau}$$

with $\bar{s}.t \preceq_F s.t$. Therefore $\bar{s}.t \preceq_F \mathit{low}$, and since $l \not\preceq_{F\cup\lceil \mathrm{E}\rceil} \mathit{low}$ implies $l \not\preceq_{F\cup \mathrm{E}_1} \mathit{low}$, by the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,\mathit{low}}\ \mathrm{E}_1[V_1]$, and Lemma 5.6 and Clause 5 then give the required result.

$\mathbf{E[(?\ a_{l,\theta})] = (E_1[(?\ a_{l,\theta})] :=^?\ M)}$:
By rule Ass, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\theta}\ \mathrm{ref}_{\bar{l},\check{n}_k}$$

with $\bar{s}.t \preceq_F s.t$ and $\bar{s}.r \preceq_F \bar{l}$. Therefore $\bar{s}.t \preceq_F \mathit{low}$, and since $l \not\preceq_{F\cup\lceil \mathrm{E}\rceil} \mathit{low}$ implies $l \not\preceq_{F\cup \mathrm{E}_1} \mathit{low}$, by the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,\mathit{low}}\ \mathrm{E}_1[V_1]$. On the other hand, by Clause 2, we have $M\ \mathscr{T}^{m_j}_{F,\mathit{low}}\ M$. By Lemma 5.2, we have $l \preceq \bar{s}.r$, so $l \preceq_F \bar{l}$. So we must have $\bar{l} \not\preceq_F \mathit{low}$, since otherwise $l \preceq_{F\cup\lceil \mathrm{E}\rceil} \mathit{low}$. Lemma 5.6 and Clause 6 then give the required result.

$\mathbf{E[(?\ a_{l,\theta})] = (V :=^?\ E_1[(?\ a_{l,\theta})])}$:
By rule Ass, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} V : \bar{s}, \bar{\theta}\ \mathrm{ref}_{\bar{l},\check{n}_k},$$

and

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}', \theta$$

with $\bar{s}'.t \preceq_F s.t$ and $\bar{s}'.r \preceq_F \bar{l}$. Therefore $\bar{s}'.t \preceq_F \mathit{low}$, and since $l \not\preceq_{F\cup\lceil \mathrm{E}\rceil} \mathit{low}$ implies $l \not\preceq_{F\cup \mathrm{E}_1} \mathit{low}$, by the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,\mathit{low}}\ \mathrm{E}_1[V_1]$. On the other hand, by Clause 2, we have $V\ \mathscr{T}^{m_j}_{F,\mathit{low}}\ V$. By Lemma 5.2, we have $l \preceq \bar{s}'.r$, so $l \preceq_F \bar{l}$. So we must have $\bar{l} \not\preceq_F \mathit{low}$, since otherwise $l \preceq_{F\cup\lceil \mathrm{E}\rceil} \mathit{low}$. Lemma 5.6 and Clause 6 then give the required result.

$\mathbf{E[(?\ a_{l,\theta})] = (\mathrm{flow}\ F'\ \mathrm{in}\ E_1[(?\ a_{l,\theta})])}$:
By rule Flow, we have

$$\Sigma;\Gamma \vdash_{F\cup F'}^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : s, \tau.$$

By the induction hypothesis, $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F\cup F',\mathit{low}}\ \mathrm{E}_1[V_1]$, so Lemma 5.6 and Clause 7 then give the required result. □

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

We can now prove that $\mathscr{T}^{m_j}_{F,low}$ behaves as a kind of 'strong bisimulation'.

Proposition 5.15 (Strong bisimulation for low-terminating threads). If we have $M_1 \; \mathscr{T}^{m_j}_{F,low} \; M_2$ and

$$\langle {M_1}^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle,$$

with

$$\langle T_1, S_1\rangle =^{F\cup F', low} \langle T_2, S_2\rangle$$

such that n is fresh for T_2 if $\bar{n}_{\bar{k}} \in \mathrm{dom}(T_1' - T_1)$ and a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, then there exist T_2', M_2' and S_2' such that

$$\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$$

with $M_1' \; \mathscr{T}^{m_j}_{F,low} \; M_2'$ and $\langle T_1', S_1'\rangle =^{F,low} \langle T_2', S_2'\rangle$.

Rationale. If M_1 and M_2 are equal (related by $\mathscr{T}$ using Clause 2), then since they have a low termination effect we can use Lemma 5.10 (Guaranteed transitions) to conclude that M_2 can also make a step and perform the same changes to low-equal memories. If the result of performing the two steps is different (and thus not falling under Clause 2 again), by Lemma 3.2 (Splitting computations), we conclude that they have performed a high dereference. We saw in Lemma 5.14 that this implies that the resulting expressions are still in the $\mathscr{T}$ relation.

The remaining cases use the fact that M_1 is a value if and only if M_2 is a value to show that if M_1 can perform a computation step, then, as long as suspension cannot occur, so can M_2.

Suspension could only occur when the dereference or assignment operations are imminent (that is, when all arguments have been computed into values). However, the possibility of suspension on an access to some reference that belongs to n_k is excluded by the fact that the threads $M_1^{m_j}$ and $M_2^{m_j}$ are assumed to have low termination effect. In fact, since if $m \neq n$ the termination of the threads depends on levels j and k, then both j and k must be low, which implies that m_j and n_k have the same position in low-equal memories. This is guaranteed by the type system in rules Der and Ass, when the termination effect is updated with $j \vee k$.

Proof. The proof is by induction on the definition of $\mathscr{T}^{m_j}_{F,low}$. In the following, we use Theorem 5.7 (Subject reduction) to guarantee that the termination effect of the expressions resulting from M_1 and M_2 is still low with respect to *low* and F. This, together with typability (with the same type) for m_j, *low* and F, is a requirement for being in the $\mathscr{T}^{m_j}_{F,low}$ relation.

Clause 1: This case cannot occur.

Clause 2: $M_1 = M_2$.

By Lemma 5.10 (Guaranteed transitions) there exist T_2', M_2' and S_2' such that

$$\langle M_2^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle M_2'^{m_j}, T_2', S_2'\rangle$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

with

$$\langle T_1', S_1' \rangle =^{F \cup F', low} \langle T_2', S_2' \rangle.$$

There are two cases to consider:

(i) $\boldsymbol{M_2' = M_1'}$:
In this case we have $M_1' \ \mathscr{T}^{m_j}_{F,low} \ M_2'$, by Clause 2 and Theorem 5.7 (Subject reduction).

(ii) $\boldsymbol{M_2' \neq M_1'}$:
In this case, by Lemma 3.2 (Splitting computations), we have $(N^{\bar{n}_{\bar{k}}} = ())$ and there exist E and $a_{l,\theta}$ such that

$$\begin{aligned} F' &= \lceil \mathrm{E} \rceil \\ M_1' &= \mathrm{E}[S_1(a_{l,\theta})] \\ M_2' &= \mathrm{E}[S_2(a_{l,\theta})] \\ \langle T_1', S_1' \rangle &= \langle T_1, S_1 \rangle \\ \langle T_2', S_2' \rangle &= \langle T_2, S_2 \rangle. \end{aligned}$$

Since $S_1(a_{l,\theta}) \neq S_2(a_{l,\theta})$, we have $l \not\preceq_{F \cup F'} low$. Therefore, $M_1' \ \mathscr{T}^{m_j}_{F,low} \ M_2'$, by Lemma 5.14.

Clause 3: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ where $\bar{M}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{M}_2$.

Then either:

(i) $\boldsymbol{\bar{M}_1}$ **can compute:**
In this case $M_1' = (\bar{M}_1'; \bar{N})$ with

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from the induction hypothesis, Clause 3 and Theorem 5.7 (Subject reduction).

(ii) $\boldsymbol{\bar{M}_1}$ **is a value:**
In this case $M_1' = \bar{N}$ and $F' = \varnothing$, $N^{\bar{n}_{\bar{k}}} = ()$ and $\langle T_1', S_1' \rangle = \langle T_1, S_1 \rangle$. We have $\bar{M}_2 \in \boldsymbol{Val}$ by Remark 5.13, so

$$\langle M_2{}^{m_j}, T_2, S_2 \rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{N}^{m_j}, T_2, S_2 \rangle,$$

and the result then follows from Clause 2 and Theorem 5.7 (Subject reduction).

Clause 4: $M_1 = (\mathrm{ref}_{l,\theta} \ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta} \ \bar{M}_2)$ where $\bar{M}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{M}_2$, and $l \not\preceq_F low$.

There are two cases:

(i) $\boldsymbol{\bar{M}_1}$ **can compute:**
In this case $M_1' = (\mathrm{ref}_{l,\theta} \ \bar{M}_1')$ with

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{\bar{n}_{\bar{k}}}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from the induction hypothesis, Theorem 5.7 (Subject reduction) and Clause 4.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

(ii) $\bar{M}_1$ **is a value:**
In this case $M_1' = a_{l,\theta}$, with a fresh for S_1, $F' = \varnothing$, $N^{\bar{n}_k} = ()$ and $\langle T_1', S_1' \rangle = \langle T_2, S_1 \cup \{a_{l,\theta} \mapsto \bar{M}_1\} \rangle$ (and therefore a is also fresh for S_2). Then $\bar{M}_2 \in \boldsymbol{Val}$ by Remark 5.13, so

$$\langle {M_2}^{m_j}, T_2, S_2 \rangle \xrightarrow[F']{N^{\bar{n}_k}} \langle {a_{l,\theta}}^{m_j}, T_2', S_2 \cup \{a_{l,\theta} \mapsto \bar{M}_2\} \rangle.$$

If we let $S_2' = S_2 \cup \{a_{l,\theta} \mapsto \bar{M}_2\}$, then $\langle T_1', S_1' \rangle =^{F,low} \langle T_2', S_2' \rangle$ since $l \not\leq_F low$. The result then follows from Clause 1 and Theorem 5.7 (Subject reduction).

Clause 5: $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ where $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$.

There are two cases:

(i) $\bar{M}_1$ **can compute:**
In this case

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{\bar{n}_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle,$$

and the result then follows from the induction hypothesis, Theorem 5.7 (Subject reduction) and Clause 5.

(ii) $\bar{M}_1$ **is a value:**
In this case $\bar{M}_1 = n_k.u_{l,\theta}$ and $M_1' \in \boldsymbol{Val}$, $\langle T_1', S_1' \rangle = \langle T_1, S_1 \rangle$, $F' = \varnothing$ and $N^{\bar{n}_k} = ()$. By Remark 5.13, we have $\bar{M}_2 \in \boldsymbol{Val}$, and since M_1 and M_2 have the same type, it must be a reference $n_k.v_{l',\theta}$. Notice also that $T_1(n_k) = T_1(m_j)$. We now further distinguish two sub-cases:

— $\boldsymbol{n \neq m}$:
In this case, by rule DER, we have $j \vee k \preceq s.t$, and therefore $j \vee k \preceq_{F \cup F'} low$. Since $T_1 =^{F \cup F', low} T_2$, we have $T_1(m_j) = T_2(m_j)$ and $T_1(n_k) = T_2(n_k)$.

— $\boldsymbol{n = m}$:
In this case it is immediate that $T_2(m_j) = T_2(n_k)$.

In both the above cases $T_2(n_k) = T_2(m_j)$, so

$$\langle {M_2}^{m_j}, T_2, S_2 \rangle \xrightarrow[F']{N^{\bar{n}_k}} \langle {M_2}'^{m_j}, T_2, S_2 \rangle$$

with $M_2' \in \boldsymbol{Val}$. The result then follows from Clause 1 and Theorem 5.7 (Subject reduction).

Clause 6: $M_1 = (\bar{M}_1 :=^{?} \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^{?} \bar{N}_2)$ with $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$, $\bar{N}_1 \mathscr{T}^{m_j}_{F,low} \bar{N}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\leq_F low$.

There are three cases:

(i) $\bar{M}_1$ **can compute.**
In this case

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{\bar{n}_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from the induction hypothesis, Theorem 5.7 (Subject reduction) and Clause 6.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

(ii) **$\bar{M}_1$ is a value, but $\bar{N}_1$ can compute:**
In this case we have

$$\langle \bar{N}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{\bar{n}_k}} \langle \bar{N}_1'^{m_j}, T_1', S_1'\rangle.$$

By Remark 5.13, we also have $\bar{M}_2 \in \textbf{\textit{Val}}$. The result then follows from the induction hypothesis, Theorem 5.7 (Subject reduction) and Clause 6.

(iii) **$\bar{M}_1$ and $\bar{N}_1$ are values:**
In this case $\bar{M}_1 = n_k.u_{l,\theta}$ and $M_1' = ()$, $\langle T_1', S_1'\rangle = \langle T_1, \{\bar{N}_1 \mapsto \bar{M}_1\}S_1\rangle$, $F' = \varnothing$ and $N^{\bar{n}_k} = ()$. By Remark 5.13, we also have $\bar{M}_2, \bar{N}_2 \in \textbf{\textit{Val}}$, and since $\bar{M}_1$ and $\bar{M}_2$ have the same type, $\bar{M}_2$ must be a reference $n_k.v_{l',\theta'}$. Notice that $T_1(n_k) = T_1(m_j)$. We now further distinguish two sub-cases:

— $\boldsymbol{n \neq m}$. By Ass, we have $j \curlyvee k \preceq s.t$, therefore $j \curlyvee k \preceq_{F\cup F'} low$. Since $T_1 =^{F\cup F', low} T_2$, we have $T_1(m_j) = T_2(m_j)$ and $T_1(n_k) = T_2(n_k)$.

— $\boldsymbol{n = m}$. In this case it is immediate that $T_2(m_j) = T_2(n_k)$.

In both these cases, $T_2(n_k) = T_2(m_j)$, so

$$\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_k}} \langle {M_2}'^{m_j}, T_2, \{\bar{N}_2 \mapsto \bar{M}_2\}S_2\rangle$$

with $\bar{M}_2' \in \textbf{\textit{Val}}$. Since $l \not\preceq_F low$, we have

$$\{\bar{N}_1 \mapsto \bar{M}_1\}S_1 =^{F\cup F', low} \{\bar{N}_2 \mapsto \bar{M}_2\}S_2.$$

Since $M_2' = M_1' = ()$, Clause 2 gives the result.

Clause 7: $M_1 = (\text{flow } \bar{F} \text{ in } \bar{M}_1)$ and $M_2 = (\text{flow } \bar{F} \text{ in } \bar{M}_2)$ and $\bar{M}_1 \; \mathscr{T}^{m_j}_{F\cup\bar{F}, low} \; \bar{M}_2$.

There are two cases:

(i) **$\bar{M}_1$ can compute:**
In this case

$$\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F'']{N^{\bar{n}_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$$

with $F' = \bar{F} \cup F''$. By the induction hypothesis,

$$\langle \bar{M}_2^{m_j}, T_2, S_2\rangle \xrightarrow[F'']{N^{\bar{n}_k}} \langle \bar{M}_2'^{m_j}, T_2', S_2'\rangle,$$

and $\bar{M}_1' \; \mathscr{T}^{m_j}_{F\cup\bar{F}, low} \; \bar{M}_2'$ and $\langle T_1', S_1'\rangle =^{F\cup\bar{F}, low} \langle T_2', S_2'\rangle$. Notice that $\langle T_1', S_1'\rangle =^{F, low} \langle T_2', S_2'\rangle$ by Remark 4.4. The result then follows from Theorem 5.7 (Subject reduction) and Clause 7.

(ii) **$\bar{M}_1$ is a value:**
In this case $M_1' = \bar{M}_1$, $F' = \varnothing$, $N^{\bar{n}_k} = ()$ and $\langle T_1', S_1'\rangle = \langle T_1, S_1\rangle$. Hence $\bar{M}_2 \in \textbf{\textit{Val}}$ by Remark 5.13, so

$$\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{\bar{n}_k}} \langle \bar{M}_2^{m_j}, T_2, S_2\rangle.$$

The result then follows from Clause 1 and Theorem 5.7 (Subject reduction). □

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

From Remark 5.13, we know that when two expressions are related by $\mathscr{T}^{m_j}_{F,low}$ and one of them is a value, the other is also a value. From a semantical point of view, when an expression has reached a value, it has successfully completed its computation. It is easy to check that when two expressions are related by $\mathscr{T}^{m_j}_{F,low}$ and one of them is unable to *resolve* into a value in any sequence of unrelated computation steps, the other is also unable to do so.

Definition 5.16 (Non-resolvable expressions). We say that an expression M is *non-resolvable*, denoted $M\dagger$, if there is no derivative M' of M such that $M' \in \boldsymbol{Val}$.

Lemma 5.17. If $M\ \mathscr{T}^{m_j}_{F,low}\ N$ and $M\dagger$ for some m_j, F and low, then $N\dagger$.

The following lemma deduces the operational 'highness' of threads from that of its subexpressions.

Lemma 5.18 (Composition of high expressions). Suppose M^{m_j} is typable in Σ and F. Then:

(1) If $M = (M_1\ M_2)$ and M_1 is a syntactically (F, low, m_j)-high function and either
   — $M_1\dagger$ and ${M_1}^{m_j} \in \mathscr{H}_{F,low}$ or
   — ${M_1}^{m_j}, {M_2}^{m_j} \in \mathscr{H}_{F,low}$,
   then $M^{m_j} \in \mathscr{H}_{F,low}$.
(2) If $M = (\text{if } M_1 \text{ then } M_t \text{ else } M_f)$ and ${M_1}^{m_j}, {M_t}^{m_j}, {M_f}^{m_j} \in \mathscr{H}_{F,low}$, then $M^{m_j} \in \mathscr{H}_{F,low}$.
(3) If $M = (\text{ref}_{l,\theta}\ M_1)$ and $l \not\preceq_F low$ and ${M_1}^{m_j} \in \mathscr{H}_{F,low}$, then $M^{m_j} \in \mathscr{H}_{F,low}$.
(4) If $M = (M_1; M_2)$ and either
   — $M_1\dagger$ and ${M_1}^{m_j} \in \mathscr{H}_{F,low}$ or
   — ${M_1}^{m_j}, {M_2}^{m_j} \in \mathscr{H}_{F,low}$,
   then $M^{m_j} \in \mathscr{H}_{F,low}$.
(5) If $M = (M_1 :=^? M_2)$ and M_1 has type $\theta\ \text{ref}_{l,\check{n}_k}$ with $l \not\preceq_F low$ and either
   — $M_1\dagger$ and ${M_1}^{m_j} \in \mathscr{H}_{F,low}$ or
   — ${M_1}^{m_j}, {M_2}^{m_j} \in \mathscr{H}_{F,low}$,
   then $M^{m_j} \in \mathscr{H}_{F,low}$.
(6) If $M = (\text{flow } F' \text{ in } M_1)$ and ${M_1}^{m_j} \in \mathscr{H}_{F\cup F',low}$, then $M^{m_j} \in \mathscr{H}_{F,low}$.

Rationale. It is easy to see that a construct that does not introduce low effects and is only composed of operationally high expressions is operationally high: for each of the computation steps that can be performed by any of its derivatives, there is a corresponding one that can be performed by a derivative of one of its components. Since the components are operationally high, the step does not make low changes to the state.

Syntactical highness of a function guarantees that its body, which can be viewed as a subexpression of an application, is operationally high. A reference creation or assignment that is only composed of operationally high expressions is operationally high for the same reasons, provided the created or written reference is high.

When a non-resolvable expression M_1 is composed with an expression M_2, as in $(M_1\ M_2)$, $(M_1; M_2)$ or $(M_1 :=^? M_2)$, it is enough to require that M_1 is operationally high.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

In fact, for all the computation steps that can be performed by any of these expressions' derivatives, there is a corresponding one that can be performed by a derivative of M_1 – that is, the expression M_2 never gets to be evaluated.

Proof. We give the proof for Case 1 (the other cases are analogous or simpler). We therefore assume that $M = (M_1\ M_2)$ and M_1 is a syntactically (F, low, m_j)-high function. There are two main possibilities to consider:

$M_1\dagger$ and ${M_1}^{m_j} \in \mathscr{H}_{F,low}$:
Let $\mathscr{F}$ be a set of threads that includes $\mathscr{H}_{F,low}$ and contains the threads $(M_1\ M_2)^{m_j}$ provided they are typable in F and satisfy $M_1 \notin \textbf{\textit{Val}}$ and $M_1^{m_j} \in \mathscr{F}$, and M_1 is a (F, low, m_j)-high function. Assume that an application $M = (M_1\ M_2)$ such that $M_1\dagger$ and ${M_1}^{m_j} \in \mathscr{H}_{F,low}$ performs the transition

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

We show that this implies ${M'}^{m_j}, N^{n_k} \in \mathscr{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$.
Since M_1 is non-resolvable, M_1 cannot be a value, and since M can compute, M_1 can also compute. We then have $M' = (M_1'\ M_2)$ with

$$\langle {M_1}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M_1'}^{m_j}, T', S'\rangle.$$

Since ${M_1}^{m_j} \in \mathscr{H}_{F,low}$, we also have ${M_1'}^{m_j}, N^{n_k} \in \mathscr{H}_{F,low}$, so both ${M_1'}^{m_j}, N^{n_k} \in \mathscr{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$. By Theorem 5.7 (Subject reduction), M_1' is an (F, low)-high function, and since $M_1\dagger$, we have $M_1' \notin \textbf{\textit{Val}}$. Hence ${M'}^{m_j} \in \mathscr{F}$.

${M_1}^{m_j}, {M_2}^{m_j} \in \mathscr{H}_{F,low}$:
Let $\mathscr{F}$ be a set of pools of threads that includes $\mathscr{H}_{F,low}$ and contains the threads $(M_1\ M_2)^{m_j}$ provided they are typable in F and satisfy $M_1^{m_j}, M_2^{m_j} \in \mathscr{F}$, and M_1 is a (F, low, m_j)-high function. Assume that such an application $M = (M_1\ M_2)$ performs the transition

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

We show that this implies ${M'}^{m_j}, N^{n_k} \in \mathscr{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$. There are three sub-cases to consider:

— **M_1 and M_2 are values**:
So $M_1 = (\lambda x.\bar{M}_1)$, $M' = \{x \mapsto M_2\}\bar{M}_1$, $N' = \emptyset$ and $\langle T', S'\rangle = \langle T, S\rangle$. Since M_1 is an (F, low, m_j)-high function, by Abs, we have $\bar{M}_1$ is syntactically (F, low, m_j)-high, and by Lemma 5.5 (Substitution), M' is also syntactically (F, low, m_j)-high. Therefore, by Lemma 5.9 (High expressions), ${M'}^{m_j} \in \mathscr{H}_{F,low}$.

— **M_1 can compute**:
We have $M' = (M_1'\ M_2)$ with

$$\langle {M_1}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M_1'}^{m_j}, T', S'\rangle.$$

Since ${M_1}^{m_j} \in \mathscr{H}_{F,low}$, we also have ${M_1'}^{m_j}, N^{n_k} \in \mathscr{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$. By Theorem 5.7 (Subject reduction) M_1' is an (F, low)-high function. Hence $M' \in \mathscr{F}$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

— ***M_1 is a value but M_2 can compute***:
We have $M' = (M_1\ M_2')$ with

$$\langle {M_2}^{m_j}, T, S\rangle \xrightarrow[F']{N^{n_k}} \langle {M_2'}^{m_j}, T', S'\rangle.$$

Since ${M_2}^{m_j}, N^{n_k} \in \mathscr{H}_{F,low}$, we also have ${M_2'}^{m_j}, N^{n_k} \in \mathscr{F}$ and $\langle T', S'\rangle =^{F,low} \langle T', S'\rangle$. Hence $M' \in \mathscr{F}$. □

Lemma 5.19. If for some m_j, F and low we have $M_1\ \mathscr{T}^{m_j}_{F,low}\ M_2$ and ${M_1}^{m_j} \in \mathscr{H}_{F,low}$, then ${M_2}^{m_j} \in \mathscr{H}_{F,low}$.

Rationale. The proof relies on the fact that if an expression M_1 of the form $(\bar{M}_1; \bar{N}_1)$ or $(\bar{M}_1 :=^? \bar{N}_1)$ is operationally high, in spite of $\bar{N}_1$ not being operationally high, then $\bar{M}_1$ is non-resolvable. To see this, note that if M_1 were not non-resolvable, we would have, for some value V, that $(V; \bar{N}_1)$ or $(V :=^? \bar{N}_1)$ are derivatives of M_1. We can then see that, for all the computation steps that can be performed by any of $\bar{N}_1$'s derivatives, there is a corresponding one that can be performed by a derivative of M_1. Since $\bar{N}_1$ is not operationally high, M would not be operationally high either.

We can easily conclude from the fact that an expression is operationally high that the first subexpression to be evaluated is also operationally high. Clauses 3 and 6 do not require their second subexpression $\bar{N}_1$ to be operationally high. However, by the above observation and Lemma 5.17, this implies that $\bar{M}_2$ is non-resolvable. We can then argue that the expressions in the $\mathscr{T}$ relation have the same 'composition', and conclude that they are operationally high using Lemma 5.18 (Composition of high expressions).

Proof. The proof is by induction on the definition of $M_1\ \mathscr{T}^{m_j}_{F,low}\ M_2$. Clauses 1 and 2 are direct.

Clause 3: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1\ \mathscr{T}^{m_j}_{F,low}\ \bar{M}_2$.

Clearly, we have $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. We distinguish two sub-cases:

(i) $\bar{N}^{m_j} \in \mathscr{H}_{F,low}$:
We have $\bar{M}_2^{m_j}, \bar{N}^{m_j} \in \mathscr{H}_{F,low}$. Therefore, by Lemma 5.18 (Composition of high expressions), we have ${M_2}^{m_j} \in \mathscr{H}_{F,low}$.

(ii) $\bar{N}^{m_j} \notin \mathscr{H}_{F,low}$:
We have $\bar{M}_1\dagger$, and by Lemma 5.17, we also have $\bar{M}_2\dagger$. Therefore, by Lemma 5.18 (Composition of high expressions), we have ${M_2}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 4: $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ where $\bar{M}_1\ \mathscr{T}^{m_j}_{F,low}\ \bar{M}_2$, and $l \not\leq_F low$.

Clearly, we have $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so by the induction hypothesis, we must also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Therefore, by Lemma 5.18 (Composition of high expressions), we have ${M_2}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 5: $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ where $\bar{M}_1\ \mathscr{T}^{m_j}_{F,low}\ \bar{M}_2$.

Clearly, we have $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. This implies that ${M_2}^{m_j} \in \mathscr{H}_{F,low}$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Clause 6: $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$, and $\bar{N}_1 \mathscr{T}^{m_j}_{F,low} \bar{N}_2$.

Clearly, we have $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. We distinguish two sub-cases:

(i) $\bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$:
We have $\bar{M}_2^{m_j}, \bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$ where $\bar{M}_2$ has type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$. Therefore, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

(ii) $\bar{N}_2^{m_j} \notin \mathscr{H}_{F,low}$:
We have $\bar{M}_1\dagger$, and by Lemma 5.17, we also have $\bar{M}_2\dagger$. Therefore, since $\bar{M}_2$ has type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 7: $M_1 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_1)$ and $M_2 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_2)$ with $\bar{M}_1 \mathscr{T}^{m_j}_{F \cup F',low} \bar{M}_2$.

Clearly, we have $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Therefore, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$. □

5.4.2. *Behaviour of typable low expressions.* In this second phase of the proof we consider the general case of threads that are typable with any termination level. As in the previous subsection, we show that a typable expression behaves as a strong bisimulation provided it is operationally low. To do this, we make use of the properties identified for the class of low-terminating expressions by only allowing members of this class to be followed by low-writes. Conversely, high-terminating expressions can only be followed by high-expressions (see Definitions 4.10 and 5.8).

Since we are considering the general case where threads do not necessarily have a low termination effect, we cannot, as we did in the previous section, state a guaranteed-transition result. However, from Lemma 5.10 (Guaranteed transitions) and Remark 5.11, we can guarantee transitions in the cases $M \neq \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ and $M \neq \mathrm{E}[(?\ n_k.u_{l,\theta})]$, as well as for these two cases when M is low-terminating. The following lemma covers the remaining cases by asserting that if $M = \mathrm{E}[(?\ a_{l,\theta})]$ when M is not low-terminating, then M is operationally high (therefore it cannot perform low changes on the state).

Lemma 5.20 (Potentially suspensive transitions). Suppose M^{m_j} is typable in Σ and F, and consider the two cases where $M = \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ and $M = \mathrm{E}[(?\ n_k.u_{l,\theta})]$ with $j \vee k \not\preceq_F low$ and $n \neq m$. Then $M^{m_j} \in \mathscr{H}_{F,low}$.

Rationale. By Remark 5.11, the assumptions $j \vee k \not\preceq_F low$ and $n \neq m$ indicate that, as far as the type system is concerned, accesses performed by a thread m_j to a reference that belongs to a thread n_k under low-equal memories are potentially suspensive. This means that, according to the principle that no low-writes can follow high-terminating portions of the program, a thread that is performing such an access must be high.

The above-mentioned principle is guaranteed by the type system using conditions of the form '$s.t \preceq_F$' on the effects and security levels representing writes that are to be performed

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

after the execution of a subprogram with security effect s. More precisely, conditions are imposed when the foreign access is used to:

— create a reference ($s.t \preceq_F l$ in rule Ref);
— determine a reference that is being assigned to ($s.t \preceq_F s'.w$ and $s.t \preceq_F l$ in rule Ass);
— determine a value that is being assigned ($s'.t \preceq_F l$ in rule Ass);
— determine the predicate of a conditional ($s.t \preceq_F s_t.w, s_f.w$ in rule Cond);
— determine a function that is being applied ($s.t \preceq_F s'.w$ and $s.t \preceq_F s''.w$ in rule App);
— determine an argument to which a function is being applied ($s''.t \preceq_F s'.w$ in rule App);
— evaluate the first component of a sequential composition ($s.t \preceq_F s'.w$ in rule Seq).

Proof. The proof is by induction on the structure of E. Consider $M = \mathrm{E}[M_0]$, where either $M_0 = (n_k.u_{l,\theta} :=^? V)$ or $M_0 = (?\ n_k.u_{l,\theta})$. The case where $\mathrm{E}[M_0] = M_0$ is direct.

$\mathrm{E}[M_0] = (\mathrm{E}_1[M_0]\ M_1)$:
By rule App, we have

$$\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \tau_1 \xrightarrow[F,\check{m}_j]{s_1'} \sigma_1$$

and

$$\Sigma;\Gamma \vdash_F^{\check{m}_j} M_1 : s_1'', \tau_1$$

with $s_1.t \preceq_F s_1''.w$ and $s_1.t \preceq_F s_1'.w$. By Remark 5.11, we have $s_1.t \not\preceq_F low$. Therefore, $s_1'.w \not\preceq_F low$ and $s_1''.w \not\preceq_F low$, which means that $\mathrm{E}_1[M_0]$ is a syntactically (F, low, m_j)-high function and M_1 is (F, low, m_j)-high. By Lemma 5.9 (High expressions), we have ${M_1}^{m_j} \in \mathscr{H}_{F,low}$. By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,low}$. Hence, by Lemma 5.18 (Composition of high expressions), $M^{m_j} \in \mathscr{H}_{F,low}$.

$\mathrm{E}[M_0] = (V\ \mathrm{E}_1[M_0])$:
By rule App, we have

$$\Sigma;\Gamma \vdash_F^{\check{m}_j} V : s_1, \tau_1 \xrightarrow[F,\check{m}_j]{s_1'} \sigma_1$$

and

$$\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1'', \tau_1$$

with $s_1''.t \preceq_F s_1'.w$. By Remark 5.11, we have $s_1''.t \not\preceq_F low$. Therefore, $s_1'.w \not\preceq_F low$ and $s_1''.w \not\preceq_F low$, which means that V is a syntactically (F, low, m_j)-high function and $\mathrm{E}_1[M_0]$ is (F, low, m_j)-high. By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,low}$. Hence, by Lemma 5.18 (Composition of high expressions), $M^{m_j} \in \mathscr{H}_{F,low}$.

$\mathrm{E}[M_0] = (\mathbf{if}\ \mathrm{E}_1[M_0]\ \mathbf{then}\ M_t\ \mathbf{else}\ M_f)$:
By rule Cond, we have

$$\Sigma;\Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \mathsf{bool},$$

and

$$\Sigma;\Gamma \vdash_F^{\check{m}_j} M_t : s_1', \tau_1$$

and

$$\Sigma;\Gamma \vdash_F^{\check{m}_j} M_f : s_1'', \tau_1$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

with $s_1.t \preceq_F s_1'.w, s_1'.w$. By Remark 5.11, we have $s_1.t \not\preceq_F \mathit{low}$, so $s_1'.w, s_1'.w \not\preceq_F \mathit{low}$. By Lemma 5.9 (High expressions), we have ${M_t}^{m_j}, {M_t}^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. Hence, by Lemma 5.18 (Composition of high expressions), $M^{m_j} \in \mathscr{H}_{F,\mathit{low}}$.

$\mathbf{E}[M_0] = (\mathbf{E}_1[M_0]; M_1)$:

By rule Seq, we have

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \tau_1$$

and

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} M_1 : s_1', \tau_1'$$

with $s_1.t \preceq_F s_1'.w$. By Remark 5.11, we have $s_1.t \not\preceq_F \mathit{low}$, so $s_1'.w \not\preceq_F \mathit{low}$. By Lemma 5.9 (High expressions), we have ${M_1}^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. Hence, by Lemma 5.18 (Composition of high expressions), $M^{m_j} \in \mathscr{H}_{F,\mathit{low}}$.

$\mathbf{E}[M_0] = (\mathbf{ref}_{l,\theta}\ \mathbf{E}_1[M_0])$:

By rule Ref, we have

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \theta$$

with $s_1.t \preceq_F l$. By Remark 5.11, we have $s_1.t \not\preceq_F \mathit{low}$, so $l \not\preceq_F \mathit{low}$. By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. Hence, by Lemma 5.18 (Composition of high expressions), $M^{m_j} \in \mathscr{H}_{F,\mathit{low}}$.

$\mathbf{E}[M_0] = (?\ \mathbf{E}_1[M_0])$:

This cases follows easily from the induction hypothesis.

$\mathbf{E}[M_0] = (\mathbf{E}_1[M_0] :=^? M_1)$:

By rule Ass, we have

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1, \theta\ \mathrm{ref}_{\bar{l},\check{n}_{\bar{k}}}$$

and

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} M_1 : s_1', \tau_1$$

with $s_1.t \preceq_F s_1'.w$ and $s_1.t \preceq_F \bar{l}$. By Remark 5.11, we have $s_1.t \not\preceq_F \mathit{low}$, so $\bar{l} \not\preceq_F \mathit{low}$ and $s_1'.w \not\preceq_F \mathit{low}$. Hence, by Lemma 5.9 (High expressions), we have ${M_1}^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. Hence, by Lemma 5.18 (Composition of high expressions), $M^{m_j} \in \mathscr{H}_{F,\mathit{low}}$.

$\mathbf{E}[M_0] = (V :=^? \mathbf{E}_1[M_0])$:

By rule Ass, we have

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} V : s_1, \theta\ \mathrm{ref}_{\bar{l},\check{n}_{\bar{k}}}$$

and

$$\Sigma; \Gamma \vdash_F^{\check{m}_j} \mathrm{E}_1[M_0] : s_1', \tau_1$$

with $s_1'.t \preceq_F \bar{l}$. By Remark 5.11, we have $s_1'.t \not\preceq_F \mathit{low}$, so $\bar{l} \not\preceq_F \mathit{low}$. By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,\mathit{low}}$. Hence, by Lemma 5.18 (Composition of high expressions), $M^{m_j} \in \mathscr{H}_{F,\mathit{low}}$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

$\mathrm{E}[M_0] = (\text{flow } F' \text{ in } \mathrm{E}_1[M_0])$:
By rule FLOW, we have

$$\Sigma;\Gamma \vdash^{\check{m}_j}_{F\cup F'} \mathrm{E}_1[M_0] : s_1, \tau_1.$$

By the induction hypothesis, $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F\cup F',low}$, which implies $\mathrm{E}_1[M_0]^{m_j} \in \mathscr{H}_{F,low}$ by Remark 4.13. Hence, by Lemma 5.18 (Composition of high expressions), we have $M^{m_j} \in \mathscr{H}_{F,low}$. □

We will now design a binary relation on expressions that uses $\mathscr{T}^{m_j}_{F,low}$ to ensure that high-terminating expressions are always followed by operationally high ones. The definition of $\mathscr{R}^{m_j}_{G,F,low}$, which is abbreviated to $\mathscr{R}^{m_j}_{F,low}$ when the global flow policy is G, is given in Definition 5.21. The flow policy F is assumed to contain G. Notice that it is a symmetric relation. In order to ensure that expressions that are related by $\mathscr{R}^{m_j}_{F,low}$ perform the same changes to the low memory, its definition requires that the references that are created or written using (potentially) different values are high, and that the body of the functions that are applied are syntactically high.

Definition 5.21 ($\mathscr{R}^{m_j}_{F,low}$). We have $M_1 \mathscr{R}^{m_j}_{F,low} M_2$ if

$$\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_1 : s_1, \tau$$

and

$$\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F} M_2 : s_2, \tau$$

for some Σ, Γ, s_1, s_2 and τ and one of the following holds:

Clause 1′: ${M_1}^{m_j}, {M_2}^{m_j} \in \mathscr{H}_{F,low}$, or
Clause 2′: $M_1 = M_2$, or
Clause 3′: $M_1 = (\text{if } \bar{M}_1 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ and $M_2 = (\text{if } \bar{M}_2 \text{ then } \bar{N}_t \text{ else } \bar{N}_f)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$, and ${\bar{N}_t}^{m_j}, {\bar{M}_f}^{m_j} \in \mathscr{H}_{F,low}$, or
Clause 4′: $M_1 = (\bar{M}_1\ \bar{N}_1)$ and $M_2 = (\bar{M}_2\ \bar{N}_2)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$, and $\bar{M}_1, \bar{M}_2$ are syntactically (F, low, m_j)-high functions, or
Clause 5′: $M_1 = (\bar{M}_1\ \bar{N}_1)$ and $M_2 = (\bar{M}_2\ \bar{N}_2)$ with $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$, and $\bar{N}_1 \mathscr{R}^{m_j}_{F,low} \bar{N}_2$, and $\bar{M}_1, \bar{M}_2$ are syntactically (F, low, m_j)-high functions, or
Clause 6′: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$, and $\bar{N}^{m_j} \in \mathscr{H}_{F,low}$, or
Clause 7′: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$, or
Clause 8′: $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$, and $l \not\leq_F low$, or
Clause 9′: $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$, or
Clause 10′: $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\leq_F low$, or
Clause 11′: $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ with $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$, and $\bar{N}_1 \mathscr{R}^{m_j}_{F,low} \bar{N}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\leq_F low$, or
Clause 12′: $M_1 = (\text{flow } F' \text{ in } \bar{M}_1)$ and $M_2 = (\text{flow } F' \text{ in } \bar{M}_2)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F\cup F',low} \bar{M}_2$.

Remark 5.22. If $M_1 \mathscr{T}^{m_j}_{F,low} M_2$, then $M_1 \mathscr{R}^{m_j}_{F,low} M_2$.

This remark is used to prove the following lemma.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Lemma 5.23. If for some m_j, F and *low* we have $M_1 \mathscr{R}^{m_j}_{F,low} M_2$ and $M_1^{m_j} \in \mathscr{H}_{F,low}$, then $M_2^{m_j} \in \mathscr{H}_{F,low}$.

Rationale. As with Lemma 5.19, the proof rests on the fact that if an expression M_1 of the form $(\bar{M}_1\ \bar{N}_1)$, $(\bar{M}_1; \bar{N}_1)$ or $(\bar{M}_1 :=^? \bar{N}_1)$ is operationally high, in spite of $\bar{N}_1$ not being operationally high, then $\bar{M}_1$ is non-resolvable.

Clauses 5′, 7′ and 11′ do not require $\bar{N}_1$ to be operationally high. However, by the above observation and Lemma 5.17, this implies that $\bar{M}_2$ is non-resolvable. Therefore, this is sufficient to conclude that $\bar{M}_2$ is operationally high.

Proof. The proof is by induction on the definition of $M_1 \mathscr{R}^{m_j}_{F,low} M_2$. Clauses 1′ and 2′ are direct.

Clause 3′: $M_1 = (\text{if } \bar{M}_1 \text{ then } \bar{M}_t \text{ else } \bar{M}_f)$ and $M_2 = (\text{if } \bar{M}_2 \text{ then } \bar{M}_t \text{ else } \bar{M}_f)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$ and $\bar{M}_t^{m_j}, \bar{M}_f^{m_j} \in \mathscr{H}_{F,low}$.

It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 4′: $M_1 = (\bar{M}_1\ \bar{N}_1)$ and $M_2 = (\bar{M}_2\ \bar{N}_2)$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$.

It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 5′ $M_1 = (\bar{M}_1\ \bar{N}_1)$ and $M_2 = (\bar{M}_2\ \bar{N}_2)$ with $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1 \mathscr{R}^{m_j}_{F,low} \bar{N}_2$.

It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by Lemma 5.19, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. We distinguish two sub-cases:

(i) $\boldsymbol{\bar{N}_1^{m_j} \in \mathscr{H}_{F,low}}$:
By the induction hypothesis, we also have $\bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

(ii) $\boldsymbol{\bar{N}_1^{m_j} \notin \mathscr{H}_{F,low}}$:
We have $\bar{M}_1\dagger$, and by Lemma 5.17, we also have $\bar{M}_2\dagger$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 6′: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1 \mathscr{R}^{m_j}_{F,low} \bar{M}_2$ and $\bar{N}^{m_j} \in \mathscr{H}_{F,low}$.

It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 7′ $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1 \mathscr{T}^{m_j}_{F,low} \bar{M}_2$.

It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by Lemma 5.19, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. We distinguish two sub-cases:

(i) $\boldsymbol{\bar{N}^{m_j} \in \mathscr{H}_{F,low}}$:
By Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

(ii) $\bar{N}^{m_j} \notin \mathscr{H}_{F,low}$:
So $\bar{M}_1\dagger$, and by Lemma 5.17, we also have $\bar{M}_2\dagger$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 8′ $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ with $\bar{M}_1\ \mathscr{R}^{m_j}_{F,low}\ \bar{M}_2$, and $l \not\preceq_F low$.
It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 9′ $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ where $\bar{M}_1\ \mathscr{R}^{m_j}_{F,low}\ \bar{M}_2$.
It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. This implies that $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 10′ $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1\ \mathscr{R}^{m_j}_{F,low}\ \bar{M}_2$, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$.
It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 11′ $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ with $\bar{M}_1\ \mathscr{T}^{m_j}_{F,low}\ \bar{M}_2$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\preceq_F low$, and $\bar{N}_1\ \mathscr{R}^{m_j}_{F,low}\ \bar{N}_2$.
It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by Lemma 5.19, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. We distinguish two sub-cases:

(i) $\bar{N}^{m_j} \in \mathscr{H}_{F,low}$:
Then, $\bar{M}_2^{m_j}, \bar{N}^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

(ii) $\bar{N}^{m_j} \notin \mathscr{H}_{F,low}$:
Then $\bar{M}_1\dagger$, and by Lemma 5.17 also $\bar{M}_2\dagger$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$.

Clause 12′ $M_1 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_1)$ and $M_2 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_2)$ with $\bar{M}_1\ \mathscr{R}^{m_j}_{F\cup F',low}\ \bar{M}_2$.
It is clear that $\bar{M}_1^{m_j} \in \mathscr{H}_{F,low}$, so, by the induction hypothesis, we also have $\bar{M}_2^{m_j} \in \mathscr{H}_{F,low}$. Thus, by Lemma 5.18 (Composition of high expressions), we have $M_2{}^{m_j} \in \mathscr{H}_{F,low}$. □

We have seen in Lemma 3.2 (Splitting computations) that two computations of the same expression can split only if the expression is about to read a reference that is given different values by the memories in which they compute. In Lemma 5.24 we saw that the relation $\mathscr{T}^{m_j}_{F,low}$ relates the possible outcomes of expressions that are typable with a low termination effect. Finally, we can conclude from the following lemma that the above relation $\mathscr{R}^{m_j}_{F,low}$ relates the possible outcomes of typable expressions in general.

Lemma 5.24. If there exist Σ, Γ, s, τ such that $\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}[(?\ a_{l,\theta})] : s, \tau$ with $l \not\preceq_{F\cup\lceil \mathrm{E}\rceil} low$, then for any values $V_0, V_1 \in \boldsymbol{Val}$ such that $\Sigma;\Gamma \vdash V_i : \theta$, we have $\mathrm{E}[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Rationale. If a typable expression is about to use a value that results from a high dereference, the following situations can occur:

— If the termination effect is low, that is, if the value cannot influence the termination behaviour of the dereference, then by Lemma 5.14 any two possible outcomes are in the relation $\mathscr{T}$ (see Clauses 5′, 7′ and 11′) and hence in $\mathscr{R}$.
— Otherwise, if the terminating effect is not low, then the type system must ensure that no low writes follow the high dereference (see Clauses 4′, 6′ and 10′). This is partly guaranteed by conditions of the form '$s.t \preceq_F s'.w$', where s is the security effect of a subprogram that is performed before another subprogram whose security effect is s'. More precisely, conditions are imposed when the dereferenced value is used to:
  – determine a reference that is being assigned to ($s.t \preceq_F s'.w$ in rule Ass);
  – determine a function that is being applied ($s.t \preceq_F s''.w$ in rule App);
  – evaluate the first component of a sequential composition ($s.t \preceq_F s'.w$ in rule Seq).

The relation $\mathscr{R}$ requires that the references that are created or written using the high dereferenced value are high (see Clauses 8′, 10′ and 11′), and that function applications that use the high dereferenced value are syntactically high. This is partly guaranteed by conditions of the form '$s.t \preceq_F l$', where s is the security effect of the subprogram that performs the high dereference, and l is the security level of the reference that is created or written. More precisely, conditions are imposed when the dereferenced value is used to:

— create a reference ($s.r \preceq_F l$ in rule Ref);
— determine a reference that is being assigned to ($s.r \preceq_F l$ in rule Ass);
— determine a value that is being assigned ($s'.r \preceq_F l$ in rule Ass);
— determine a function that is being applied ($s.r \preceq_F s'.w$ in rule App);
— determine an argument to which a function is being applied ($s''.r \preceq_F s'.w$ in rule App).

When the high dereferenced value is used in the predicate of a conditional, the branches should be operationally high (see Clause 3′). This is guaranteed by the type system with the condition $s.r \preceq_F s_t.w, s_f.w$ in rule Cond.

Proof. The proof is by induction on the structure of E:

$\mathrm{E}[(?\ a_{l,\theta})] = (?\ a_{l,\theta})$:

We have $V_0\ \mathscr{R}^{m_j}_{F,low}\ V_1$ by Clause 1′.

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathrm{E}_1[(?\ a_{l,\theta})]\ M)$:

By rule App, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\tau} \xrightarrow[F,\Sigma(m_j)]{\bar{s}'} \bar{\sigma}$$

and

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} M : \bar{s}'', \bar{\tau}$$

with $\bar{s}.r \preceq_F \bar{s}'.w$ and $\bar{s}.t \preceq_F \bar{s}''.w$. By Lemma 5.2, we have $l \preceq \bar{s}.r$. Therefore $l \preceq_F \bar{s}'.w$. Since, by hypothesis, $l \not\preceq_{F \cup \lceil \mathrm{E}_1 \rceil} low$ (and thus $l \not\preceq_F low$), we have $\bar{s}'.w \not\preceq_F low$, that is, $\mathrm{E}_1[(?\ a_{l,\theta})]$ is a syntactically (F, low, m_j)-high function. By Lemma 5.6, the

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

same holds for $\mathrm{E}_1[V_0]$ and $\mathrm{E}_1[V_1]$. By the induction hypothesis, we conclude that $\mathrm{E}_1[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. There are two cases to consider:

(i) $\bar{s}.t \not\preceq_F low$:
We have $\bar{s}''.w \not\preceq_F low$ (and also $\bar{s}''.w \not\preceq low$), so by Lemma 5.9 (High expressions), we have $M^{m_j} \in \mathscr{H}_{F,low}$. So we have $\mathrm{E}[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 4′ and Lemma 5.6.

(ii) $\bar{s}.t \preceq_F low$:
By Lemma 5.14, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore, since $M\ \mathscr{R}^{m_j}_{F,low}\ M$ by Clause 2′, we have $\mathrm{E}[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 5′ and Lemma 5.6.

$\mathrm{E}[(?\ a_{l,\theta})] = (V\ \mathrm{E}_1[(?\ a_{l,\theta})])$:

By App, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} V : \bar{s},\bar{\tau} \xrightarrow[F,\Sigma(m_j)]{\bar{s}'} \bar{\sigma}$$

and

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}'',\bar{\tau}$$

with $\bar{s}''.r \preceq_F \bar{s}'.w$. By Lemma 5.2, we have $l \preceq \bar{s}''.r$, so $l \preceq_F \bar{s}'.w$. Since, by hypothesis, $l \not\preceq_{F\cup\lceil \mathrm{E}_1\rceil} low$ (and thus $l \not\preceq_F low$), we have $\bar{s}'.w \not\preceq_F low$, that is, V is a syntactically (F, low, m_j)-high function. By Clause 1, we have $V\ \mathscr{T}^{m_j}_{F,low}\ V$. By the induction hypothesis, $\mathrm{E}_1[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore we have $\mathrm{E}[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 5′ and Lemma 5.6.

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathbf{if}\ \mathrm{E}_1[(?\ a_{l,\theta})]\ \mathbf{then}\ M_t\ \mathbf{else}\ M_f)$:

By Cond, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\mathsf{bool},$$

and

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} M_t : \bar{s}_t,\bar{\tau}$$

and

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} M_f : \bar{s}_f,\bar{\tau}$$

with $\bar{s}.r \preceq_F \bar{s}_t.w, \bar{s}_f.w$. By Lemma 5.2, we have $l \preceq \bar{s}.r$, so $l \preceq_F \bar{s}_t.w, \bar{s}_f.w$. Since, by hypothesis, $l \not\preceq_{F\cup\lceil \mathrm{E}_1\rceil} low$ (and thus $l \not\preceq_F low$), we have $\bar{s}_t.w \not\preceq_F low$ and $\bar{s}_f.w \not\preceq_F low$. This implies that ${M_t}^{m_j}, {M_f}^{m_j} \in \mathscr{H}_{F,low}$. By the induction hypothesis, $\mathrm{E}_1[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. Therefore we have $\mathrm{E}[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 3′ and Lemma 5.6.

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathrm{E}_1[(?\ a_{l,\theta})]; M)$:

By Seq, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s},\bar{\tau}$$

and

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} M : \bar{s}',\bar{\tau}'$$

with $\bar{s}.t \preceq_F \bar{s}'.w$. There are two cases:

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

(i) $\bar{s}.t \not\preceq_F$ ***low***:
We have $\bar{s}'.w \not\preceq_F low$, so, by Lemma 5.9 (High expressions), we have $M^{m_j} \in \mathscr{H}_{F,low}$. By the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. We then have $\mathrm{E}[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 6′ and Lemma 5.6.

(ii) $\bar{s}.t \preceq_F$ ***low***:
By Lemma 5.14, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$, and the result then follows from Clause 7′ and Lemma 5.6.

$\mathbf{E[(?\ a_{l,\theta})] = (\mathrm{ref}_{\bar{l},\bar{\theta}}\ E_1[(?\ a_{l,\theta})])$:
By Ref, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\tau}$$

with $\bar{s}.r = s.r \preceq_F \bar{l}$ and $\bar{s}.t = s.t$. Therefore, since $l \not\preceq_{F\cup \mathrm{E}} low$ implies $l \not\preceq_{F\cup \mathrm{E}_1} low$, by the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. By Lemma 5.2, we have $l \preceq s.r$, so $s.r \not\preceq_F low$. Therefore, $\bar{l} \not\preceq_F low$, and the result follows from Lemma 5.6 and Clause 8′.

$\mathbf{E[(?\ a_{l,\theta})] = (?\ E_1[(?\ a_{l,\theta})])$:
By rule Der, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[(?\ a_{l,\theta})] : \bar{s}, \bar{\tau}.$$

By the induction hypothesis, $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$, and the result then follows from Lemma 5.6 and Clause 9′.

$\mathbf{E[(?\ a_{l,\theta})] = (E_1[(?\ a_{l,\theta})] :=^{?} M)$:
By rule Ass, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[a_{l,\theta}] : \bar{s}, \bar{\theta}\ \mathrm{ref}_{\bar{l},\check{n}_k}$$

with $\bar{s}.r \preceq_F \bar{l}$ and $\bar{s}.t \preceq_F \bar{s}'.w$. By Lemma 5.2, we have $l \preceq s.r$, so $s.r \not\preceq_F low$ so $\bar{l} \not\preceq_F low$. There are two cases to consider:

(i) $\bar{s}.t \not\preceq_F$ ***low***:
We have $\bar{s}'.w \not\preceq_F low$, so, by Lemma 5.9 (High expressions), we have that $M^{m_j} \in \mathscr{H}_{F,low}$. By the induction hypothesis, $\mathrm{E}_1[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. We then have $\mathrm{E}[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}[V_1]$ by Clause 10′ and Lemma 5.6.

(ii) $\bar{s}.t \preceq_F$ ***low***:
By Lemma 5.14, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$, and the result then follows from Lemma 5.6, Clause 11′ and Clause 2′ (regarding M).

$\mathbf{E[(?\ a_{l,\theta})] = (V :=^{?} E_1[(?\ a_{l,\theta})])$:
By rule Ass, we have

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} V : \bar{s}, \bar{\theta}\ \mathrm{ref}_{\bar{l},\check{n}_k}$$

and

$$\Sigma;\Gamma \vdash_F^{\Sigma(m_j)} \mathrm{E}_1[a_{l,\theta}] : \bar{s}', \theta$$

with $\bar{s}'.r \preceq_F \bar{l}$. By Lemma 5.2, we have $l \preceq \bar{s}'.r$, so $l \preceq_F \bar{l}$. So we must have $\bar{l} \not\preceq_F low$, since otherwise $l \preceq_{F\cup \mathrm{E}} low$. By Clause 1, we have $V\ \mathscr{T}^{m_j}_{F,low}\ V$, and by the induction hypothesis, $\mathrm{E}_1[V_0]\ \mathscr{R}^{m_j}_{F,low}\ \mathrm{E}_1[V_1]$. The result follows from Lemma 5.6 and Clause 11′.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

$\mathrm{E}[(?\ a_{l,\theta})] = (\mathbf{flow}\ F'\ \mathbf{in}\ \mathrm{E}_1[(?\ a_{l,\theta})])$:

By rule FLOW, we have $\Sigma;\Gamma \vdash^{\Sigma(m_j)}_{F\cup F'} V : s,\tau$. By the induction hypothesis, we have $\mathrm{E}_1[V_0]\ \mathscr{T}^{m_j}_{F\cup F',low}\ \mathrm{E}_1[V_1]$, and the result follows from Lemma 5.6 and Clause 12′. □

We can now state a crucial result of the paper: the relation $\mathscr{R}^{m_j}_{F,low}$ is a sort of 'strong bisimulation'.

Proposition 5.25 (Strong bisimulation for typable low threads).
If $M_1\ \mathscr{R}^{m_j}_{F,low}\ M_2$ and $M_1 \notin \mathscr{H}_{F,low}$ and

$$\langle {M_1}^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle {M_1'}^{m_j}, T_1', S_1'\rangle,$$

with

$$\langle T_1, S_1\rangle =^{F\cup F',low} \langle T_2, S_2\rangle$$

such that n is fresh for T_2 if $n \in \mathrm{dom}(T_1' - T_1)$ and a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, then there exist T_2', M_2' and S_2' such that

$$\langle {M_2}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{n_k}} \langle {M_2'}^{m_j}, T_2', S_2'\rangle$$

with $M_1'\ \mathscr{R}^{m_j}_{F,low}\ M_2'$ and

$$\langle T_1', S_1'\rangle =^{F,low} \langle T_2', S_2'\rangle.$$

Rationale. Assuming that M_1 (and M_2) are not operationally high allows us to conclude in many cases that a certain subexpression can compute, using Lemma 5.18 (Composition of high expressions). This applies, in particular, to potentially suspensive expressions.
If $M_1^{m_j}$ and $M_2^{m_j}$ are equal (related by $\mathscr{T}$ using Clause 2), we can reject the case where m_j is accessing a high remote reference, since by Lemma 5.20 (Potentially suspensive transitions) we would have $M_1^{m_j}$ operationally high. We can then proceed as in Proposition 5.15 (Strong bisimulation for low-terminating threads).

Proof. The proof is by induction on the definition of $\mathscr{R}^{m_j}_{F,low}$. We use Theorem 5.7 (Subject reduction) to guarantee typability (with the same type) for m_j, *low* and F, which is a requirement for being in the $\mathscr{R}^{m_j}_{F,low}$ relation. We also use Lemma 5.15 (Strong bisimulation for low terminating threads).

Clause 1′:

This case is excluded by assumption.

Clause 2′: $M_1 = M_2$.

If $M_1 = \mathrm{E}[(n_k.u_{l,\theta} :=^? V)]$ or $M_1 = \mathrm{E}[(?\ n_k.u_{l,\theta})]$ with $j \vee k \not\leq_F low$ and $n \neq m$, by Lemma 5.20 (Potentially suspensive transitions), we have $M^{m_j} \in \mathscr{H}_{F,low}$, which is rejected by assumption. Otherwise, the proof is analogous to the corresponding case in Lemma 5.15 (Strong bisimulation for low-typable threads). By Lemma 5.10 (Guaranteed transitions), there exist T_2', M_2' and S_2' such that

$$\langle M_2^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{n_k}} \langle M_2'^{m_j}, T_2', S_2'\rangle$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

with

$$\langle T_1', S_1' \rangle =^{F \cup F', low} \langle T_2', S_2' \rangle.$$

There are two cases to consider:

(i) $\boldsymbol{M_2' = M_1'}$**:**
We have $M_1' \ \mathscr{R}^{m_j}_{F,low} \ M_2'$ by Clause 2′ and Theorem 5.7 (Subject reduction).

(ii) $\boldsymbol{M_2' \neq M_1'}$**:**
By Lemma 3.2 (Splitting computations), we have $N^{n_k} = ()$ and there exist E and $a_{l,\theta}$ such that

$$\begin{aligned} F' &= \lceil \mathrm{E} \rceil \\ M_1' &= \mathrm{E}[S_1(a_{l,\theta})] \\ M_2' &= \mathrm{E}[S_2(a_{l,\theta})] \\ \langle T_1', S_1' \rangle &= \langle T_1, S_1 \rangle \\ \langle T_2', S_2' \rangle &= \langle T_2, S_2 \rangle. \end{aligned}$$

Since $S_1(a_{l,\theta}) \neq S_2(a_{l,\theta})$, we have $l \not\leq_{F \cup F'} low$. Therefore, $M_1' \ \mathscr{R}^{m_j}_{F,low} \ M_2'$, by Lemma 5.24.

Clause 3′: $M_1 =$ (if $\bar{M}_1$ then $\bar{M}_t$ else $\bar{M}_f$) and $M_2 =$ (if $\bar{M}_2$ then $\bar{M}_t$ else $\bar{M}_f$) with $\bar{M}_1 \ \mathscr{R}^{m_j}_{F,low} \ \bar{M}_2$ and $\bar{M}_t^{m_j}, \bar{M}_f^{m_j} \in \mathscr{H}_{F,low}$.

We can assume that $\bar{M}_1^{m_j} \notin \mathscr{H}_{F,low}$, since otherwise $M_1^{m_j} \in \mathscr{H}_{F,low}$ by Lemma 5.18 (Composition of high expressions). Therefore, $M_1' =$ (if $\bar{M}_1'$ then $\bar{M}_t$ else $\bar{M}_f$) with

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from the induction hypothesis, Clause 3′ and Theorem 5.7 (Subject reduction).

Clause 4′: $M_1 = (\bar{M}_1 \ \bar{N}_1)$ and $M_2 = (\bar{M}_2 \ \bar{N}_2)$ with $\bar{M}_1 \ \mathscr{R}^{m_j}_{F,low} \ \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$.

We can assume that $\bar{M}_1$ can compute, since otherwise $M_1^{m_j} \in \mathscr{H}_{F,low}$ by Lemma 5.18 (Composition of high expressions). Therefore, $M_1' = (\bar{M}_1' \ \bar{N}_1)$ with

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from the induction hypothesis, Clause 4′ and Theorem 5.7 (Subject reduction).

Clause 5′: $M_1 = (\bar{M}_1 \ \bar{N}_1)$ and $M_2 = (\bar{M}_2 \ \bar{N}_2)$ with $\bar{M}_1 \ \mathscr{T}^{m_j}_{F,low} \ \bar{M}_2$, $\bar{M}_1$ and $\bar{M}_2$ are syntactically (F, low, m_j)-high functions, and $\bar{N}_1 \ \mathscr{R}^{m_j}_{F,low} \ \bar{N}_2$.

We distinguish two sub-cases:

(i) $\boldsymbol{\bar{M}_1}$ **can compute:**
In this case there exists $\bar{M}_1'$ such that

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

The result then follows from Lemma 5.15, Theorem 5.7 (Subject reduction) and Clause 5′.

(ii) $\bar{M}_1$ **is a value:**
By Remark 5.13, we have $\bar{M}_2 \in \boldsymbol{Val}$. We can assume that $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \notin \mathscr{H}_{F,low}$, since otherwise $M_1^{m_j} \in \mathscr{H}_{F,low}$ by Lemma 5.18 (Composition of high expressions). So $\bar{N}_1$ can compute, so there exist $\bar{N}_1'$ such that

$$\langle \bar{N}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{N}_1'^{m_j}, T_1', S_1' \rangle$$

with $M_1' = (\bar{M}_1\ \bar{N}_1')$. The result then follows from the induction hypothesis, Clause 5′ and Theorem 5.7 (Subject reduction).

Clause 6′: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1\ \mathscr{R}_{F,low}^{m_j}\ \bar{M}_2$ and $\bar{N}^{m_j} \in \mathscr{H}_{F,low}$.

We can assume that $\bar{M}_1^{m_j} \notin \mathscr{H}_{F,low}$, since otherwise $M_1^{m_j} \in \mathscr{H}_{F,low}$ by Lemma 5.18 (Composition of high expressions). Therefore, we have $M_1' = (\bar{M}_1'; \bar{N})$ with

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from the induction hypothesis, Clause 6′ and Theorem 5.7 (Subject reduction).

Clause 7′: $M_1 = (\bar{M}_1; \bar{N})$ and $M_2 = (\bar{M}_2; \bar{N})$ with $\bar{M}_1\ \mathscr{T}_{F,low}^{m_j}\ \bar{M}_2$.

We distinguish two sub-cases:

(i) $\bar{M}_1$ **can compute:**
In this case there exists $\bar{M}_1'$ such that

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from Lemma 5.15, Theorem 5.7 (Subject reduction) and Clause 7′.

(ii) $\bar{M}_1$ **is a value:**
So $M_1' = \bar{N}$, $F = \varnothing$, $N^{n_k} = ()$ and $\langle T_1', S_1' \rangle = \langle T_1, S_1 \rangle$. By Remark 5.13, $\bar{M}_2 \in \boldsymbol{Val}$. So

$$\langle M_2^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{N}^{m_j}, T_1', S_1' \rangle.$$

The result then follows from Lemma 5.15 and Clause 2′.

Clause 8′: $M_1 = (\mathrm{ref}_{l,\theta}\ \bar{M}_1)$ and $M_2 = (\mathrm{ref}_{l,\theta}\ \bar{M}_2)$ with $\bar{M}_1\ \mathscr{R}_{F,low}^{m_j}\ \bar{M}_2$, and $l \not\preceq_F low$.

We can assume that $\bar{M}_1^{m_j} \notin \mathscr{H}_{F,low}$, since otherwise $M_1^{m_j} \in \mathscr{H}_{F,low}$ by Lemma 5.18 (Composition of high expressions). So $\bar{M}_1$ can compute, and $M_1' = (\mathrm{ref}_{l,\theta}\ \bar{M}_1')$ with

$$\langle \bar{M}_1^{m_j}, T_1, S_1 \rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1' \rangle.$$

The result then follows from the induction hypothesis, Theorem 5.7 (Subject reduction) and Clause 8′.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Clause 9′: $M_1 = (?\ \bar{M}_1)$ and $M_2 = (?\ \bar{M}_2)$ where $\bar{M}_1\ \mathscr{R}^{m_j}_{F,low}\ \bar{M}_2$.

We know that $\bar{M}_1$ can compute, since otherwise ${M_1}^{m_j} \in \mathscr{H}_{F,low}$. So we have

$$\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle.$$

The result then follows from the induction hypothesis, Theorem 5.7 (Subject reduction) and Clause 9′.

Clause 10′: $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ where $\bar{M}_1\ \mathscr{R}^{m_j}_{F,low}\ \bar{M}_2$, and $\bar{N}_1^{m_j}, \bar{N}_2^{m_j} \in \mathscr{H}_{F,low}$, and $\bar{M}_1, \bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\leq_F low$.

We can assume that $\bar{M}_1$ can compute, since otherwise $M_1^{m_j} \in \mathscr{H}_{F,low}$ by Lemma 5.18 (Composition of high expressions). Therefore, $M_1' = (\bar{M}_1' :=^? \bar{N}_1)$ with

$$\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle.$$

The result then follows from the induction hypothesis, Clause 10′ and Theorem 5.7 (Subject reduction).

Clause 11′: $M_1 = (\bar{M}_1 :=^? \bar{N}_1)$ and $M_2 = (\bar{M}_2 :=^? \bar{N}_2)$ with $\bar{M}_1\ \mathscr{T}^{m_j}_{F,low}\ \bar{M}_2$, and $\bar{M}_1$, $\bar{M}_2$ both have type $\theta\ \mathrm{ref}_{l,\check{n}_k}$ for some θ and l such that $l \not\leq_F low$, and $\bar{N}_1\ \mathscr{R}^{m_j}_{F,low}\ \bar{N}_2$.

We can assume that M_1 cannot be a redex with $\bar{M}_1, \bar{N}_1 \in \boldsymbol{Val}$ since otherwise $M_1^{m_j} \in \mathscr{H}_{F,low}$ by Lemma 5.18 (Composition of high expressions). There are two cases to consider:

(i) **$\bar{M}_1$ can compute:**
We have

$$\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle.$$

The result then follows from Lemma 5.15, Clause 11′ and Theorem 5.7 (Subject reduction).

(ii) **$\bar{M}_1$ is a value but $\bar{N}_1$ can compute:**
By Remark 5.13, $\bar{M}_2 \in \boldsymbol{Val}$, so

$$\langle \bar{N}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F']{N^{n_k}} \langle \bar{N}_1'^{m_j}, T_1', S_1'\rangle.$$

The result then follows from the induction hypothesis, Clause 11′ and Theorem 5.7 (Subject reduction).

Clause 12′: $M_1 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_1)$ and $M_2 = (\mathrm{flow}\ F'\ \mathrm{in}\ \bar{M}_2)$ with $\bar{M}_1\ \mathscr{R}^{m_j}_{F\cup F',low}\ \bar{M}_2$.

We can assume that $\bar{M}_1^{m_j} \notin \mathscr{H}_{F\cup F',low}$, since otherwise $\bar{M}_1^{m_j} \notin \mathscr{H}_{F,low}$ and by Lemma 5.18 (Composition of high expressions) $M_1^{m_j} \in \mathscr{H}_{F,low}$. Therefore

$$\langle \bar{M}_1^{m_j}, T_1, S_1\rangle \xrightarrow[F'']{N^{n_k}} \langle \bar{M}_1'^{m_j}, T_1', S_1'\rangle$$

with $F' = \bar{F} \cup F''$. By the induction hypothesis, we have

$$\langle \bar{M}_2^{m_j}, T_2, S_2\rangle \xrightarrow[F'']{N^{n_k}} \langle \bar{M}_2'^{m_j}, T_2', S_2'\rangle$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

and

$$M_1' \ \mathscr{R}^{m_j}_{F\cup\bar{F},low} \ M_2'$$

and

$$\langle T_1', S_1'\rangle =^{F\cup\bar{F},low} \langle T_2', S_2'\rangle.$$

Noting that

$$\langle T_1', S_1'\rangle =^{F,low} \langle T_2', S_2'\rangle$$

by Remark 4.4, the result then follows from Theorem 5.7 (Subject reduction) and Clause 12′. □

5.4.3. *Behaviour of sets of typable threads.* To conclude the proof of the Soundness Theorem, we still need to exhibit an appropriate bisimulation on pools of threads.

Definition 5.26 ($\mathscr{R}^{\star}_{G,low}$). The relation $\mathscr{R}^{\star}_{G,low}$ is defined inductively as follows:

$$\text{(a)} \quad \frac{M^{m_j} \in \mathscr{H}_{G,low}}{\{M^{m_j}\} \ \mathscr{R}^{\star}_{G,low} \ \varnothing}$$

$$\text{(b)} \quad \frac{M^{m_j} \in \mathscr{H}_{G,low}}{\varnothing \ \mathscr{R}^{\star}_{G,low} \ \{M^{m_j}\}}$$

$$\text{(c)} \quad \frac{M_1 \ \mathscr{R}^{m_j}_{G,low} \ M_2}{\{{M_1}^{m_j}\} \ \mathscr{R}^{\star}_{G,low} \ \{{M_2}^{m_j}\}}$$

$$\text{(d)} \quad \frac{P_1 \ \mathscr{R}^{\star}_{G,low} \ P_2 \quad Q_1 \ \mathscr{R}^{\star}_{G,low} \ Q_2}{P_1 \cup Q_1 \ \mathscr{R}^{\star}_{G,low} \ P_2 \cup Q_2}.$$

Proposition 5.27. The relation $\mathscr{R}^{\star}_{G,low}$ is a (G, low)-bisimulation.

Rationale. Operationally high threads can be added to any pool of threads without affecting the possibility of it being bisimilar to another pool of threads. This results from the fact that threads in the set $\mathscr{H}$ can only generate threads that are in $\mathscr{H}$, and none of them can perform changes to the low memory. Therefore, any step that is performed by an operationally high thread can be simulated by any pool of threads by doing nothing.

For each of the pairs of threads that are related by $\mathscr{R}^{\star}$, we use Proposition 5.25 (Strong bisimulation for typable low threads) and Clause 2′ to prove that the expressions that are related by $\mathscr{R}^{m_j}_{G,low}$ can simulate each other's steps, and that any threads that they eventually create are related by $\mathscr{R}^{\star}$.

Proof. First, it is easy to see by induction on the definition of $\mathscr{R}^{\star}_{G,low}$ that this relation is symmetric. We now show by induction on the definition of $\mathscr{R}^{\star}_{G,low}$ that if $P_1 \ \mathscr{R}^{\star}_{G,low} \ P_2$ and $\langle P_1, T_1, S_1\rangle \underset{F}{\rightarrow} \langle P_1', T_1', S_1'\rangle$, n is fresh for T_2 if $n \in \mathrm{dom}(T_1' - T_1)$ and a is fresh for S_2 if $a_{l,\theta} \in \mathrm{dom}(S_1' - S_1)$, and if $\langle T_1, S_1\rangle =^{F\cup G,low} \langle T_2, S_2\rangle$, then there exist T_2', P_2' and S_2' such that $\langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2', T_2', S_2'\rangle$ and $P_1' \ \mathscr{R}^{\star}_{G,low} \ P_2'$ and $\langle T_1', S_1'\rangle =^{F\cup G,low} \langle T_2', S_2'\rangle$:

Rule (a): $P_1 = \{M^{m_j}\}$, $P_2 = \varnothing$, and $M^{m_j} \in \mathscr{H}_{G,low}$.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

In this case we have

$$\langle M^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle M'^{m_j}, T_1', S_1'\rangle,$$

with $P_1' = \{M'^{m_j}, N^{n_k}\}$, where we have $M_1'^{m_j}, N^{n_k} \in \mathscr{H}_{G,low}$ and $\langle T_1', S_1'\rangle =^{G,low} \langle T_1, S_1\rangle$. We have $\langle P_2, T_2, S_2\rangle \twoheadrightarrow \langle P_2, T_2, S_2\rangle$ and, by transitivity, $\langle T_1', S_1'\rangle =^{G,low} \langle T_2, S_2\rangle$. By Rule (a) we have $\{M_1'^{m_j}\}\ \mathscr{R}^{\star}_{G,low}\ \varnothing$ and $\{N^{n_k}\}\ \mathscr{R}^{\star}_{G,low}\ \varnothing$. Therefore, by Rule (d), we have $P_1'\ \mathscr{R}^{\star}_{G,low}\ \varnothing$.

Rule (c): $P_1 = \{M_1{}^{m_j}\}$ and $P_2 = \{M_2{}^{m_j}\}$, and we have $M_1\ \mathscr{R}^{m_j}_{G,low}\ M_2$.

By the case for Rule (a), we have $P_1'\ \mathscr{R}^{\star}_{G,low}\ \varnothing$ and

$$\langle T_1', S_1'\rangle =^{F\cup G,low} \langle T_2', S_2'\rangle.$$

If $M_1^{m_j} \in \mathscr{H}_{G,low}$, then, by Lemma 5.23, we also have $M_2^{m_j} \in \mathscr{H}_{G,low}$, so by Rule (b), $\varnothing\ \mathscr{R}^{\star}_{G,low}\ P_2$. Then, by Rule (d), we have $P_1'\ \mathscr{R}^{\star}_{G,low}\ P_2$. If $M_1^{m_j} \notin \mathscr{H}_{G,low}$, there are two cases to consider:

(i) $\boldsymbol{P_1' = \{M_1'^{m_j}\}}$:
We have

$$\langle M_1{}^{m_j}, T_1, S_1\rangle \xrightarrow[F]{\emptyset} \langle M_1'{}^{m_j}, T_1', S_1'\rangle,$$

so by Proposition 5.25 (Strong bisimulation for typable low threads), there exist T_2', M_2' and S_2' such that

$$\langle M_2{}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{\emptyset} \langle M_2'{}^{m_j}, T_2', S_2'\rangle$$

with $M_1'\ \mathscr{R}^{m_j}_{G,low}\ M_2'$ and $\langle T_1', S_1'\rangle =^{G,low} \langle T_2', S_2'\rangle$. Hence, by Rule (c), we have $\{M_1{}^{m_j}\}\ \mathscr{R}^{\star}_{G,low}\ \{M_2{}^{m_j}\}$.

(ii) $\boldsymbol{P_1' = \{M_1'^{m_j}, N^{n_k}\}}$:
We have

$$\langle M_1{}^{m_j}, T_1, S_1\rangle \xrightarrow[F]{N^{n_k}} \langle M_1'{}^{m_j}, T_1', S_1'\rangle,$$

and again by Proposition 5.25 (Strong bisimulation for typable low threads), there exist T_2', M_2' and S_2' such that

$$\langle M_2{}^{m_j}, T_2, S_2\rangle \xrightarrow[F']{N^{n_k}} \langle M_2'{}^{m_j}, T_2', S_2'\rangle$$

with $M_1'\ \mathscr{R}^{m_j}_{G,low}\ M_2'$ and $\langle T_1', S_1'\rangle =^{G,low} \langle T_2', S_2'\rangle$. Hence, by Rule (c), we have $\{M_1{}^{m_j}\}\ \mathscr{R}^{\star}_{G,low}\ \{M_2{}^{m_j}\}$. By Theorem 5.7 (Subject reduction), Lemma 5.4 and Clause 2′, we have $N\ \mathscr{R}^{n_k}_{G,low}\ N$, so by Rule (c), we have $\{N^{n_k}\}\ \mathscr{R}^{\star}_{G,low}\ \{N^{n_k}\}$. Therefore, by Rule (d), we have $\{M_1{}^{m_j}, N^{n_k}\}\ \mathscr{R}^{\star}_{G,low}\ \{M_2{}^{m_j}, N^{n_k}\}$.

Rule (d): $P_1 = \bar{P}_1 \cup \bar{Q}_1$ and $P_2 = \bar{P}_2 \cup \bar{Q}_2$, with $\bar{P}_1\ \mathscr{R}^{\star}_{G,low}\ \bar{P}_2$ and $\bar{Q}_1\ \mathscr{R}^{\star}_{G,low}\ \bar{Q}_2$.

Suppose

$$\langle \bar{P}_1, T_1, S_1\rangle \underset{F}{\rightarrow} \langle \bar{P}_1', T_1', S_1'\rangle$$

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

– the case where $\bar{Q}_1$ reduces is analogous. By the induction hypothesis, there exist T_2', $\bar{P}_2'$ and S_2' such that

$$\langle \bar{P}_2, T_2, S_2 \rangle \twoheadrightarrow \langle \bar{P}_2', T_2', S_2' \rangle$$

with $\bar{P}_1' \; \mathscr{R}^{\star}_{G,low} \; \bar{P}_2'$ and $\langle T_1', S_1' \rangle =^{G,low} \langle T_2', S_2' \rangle$. So

$$\langle \bar{P}_2 \cup \bar{Q}_2, T_2, S_2 \rangle \twoheadrightarrow \langle \bar{P}_2' \cup \bar{Q}_2, T_2', S_2' \rangle,$$

and by Rule (d), we have $\bar{P}_1' \cup \bar{Q}_1 \; \mathscr{R}^{\star}_{G,low} \; \bar{P}_2' \cup \bar{Q}_2$. □

We can now state the main result of this paper.

Theorem 5.28 (Soundness for non-disclosure for networks). Consider a pool of threads P and a global flow policy G. If for all $M^{m_j} \in P$ there exist Σ, Γ, s and τ such that $\Sigma; \Gamma \vdash^{\Sigma(m_j)}_{G,G} M : s, \tau$, then P satisfies the non-disclosure for networks policy with respect to G.

Proof. By Clause 2′ of Definition 5.21, for all choices of security levels *low*, we have $M \; \mathscr{R}^{m_j}_{G,low} \; M$. By Rule (c) of Definition 5.26, we then have $\{M^{m_j}\} \; \mathscr{R}^{\star}_{G,low} \; \{M^{m_j}\}$. Since this is true for all $M^{m_j} \in P$, by Rule (d), we have $P \; \mathscr{R}^{\star}_{G,low} \; P$. By Proposition 5.27, we can then conclude that $P \approx_{G,low} P$. □

This result is compositional in the sense that it is enough to verify the typability of each thread separately to ensure non-disclosure for the whole network. The global flow policy G can be taken as the 'intersection' of the flow policies of all the threads in the network. As we observed earlier, this operation seems too costly and complex to be used in a general case, but the result can be conveniently approximated by the empty flow relation, which gives the minimum flow security pre-lattice that all threads must satisfy.

6. Related work

To the best of our knowledge, this is the first study on the security of information flows that are introduced by mobility in the context of a distributed language with states. Moreover, it seems to be the first to consider the usage of declassification in a distributed scenario. This discussion of related work will focus on type-based approaches for enforcing information flow policies in settings with distribution and mobility, giving particular attention to Crafa *et al.* (2002), which is the work closest to ours.

Our work follows in a line of study that approaches similar problems in the context of simpler concurrent settings. When studying non-interference for an imperative multi-threaded language, Smith and Volpano (1998) recognised *termination leaks* as an issue specific to concurrent settings, and provided a type system that rejects them. This was followed by a series of studies that considered increasingly expressive languages and more refined type systems (Smith 2001; Boudol and Castellani 2002; Honda and Yoshida 2002; Almeida Matos and Boudol 2009; Boudol 2005b). A particular kind of termination leak, known as a *suspension leak*, was studied and handled in Sabelfeld (2001) and Almeida Matos *et al.* (2007) in the presence of different forms of synchronisation.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

In a distributed setting, but where interaction between domains is restricted to the exchange of values (no code mobility), Mantel and Sabelfeld provided a type system for preserving confidentiality for different kinds of channels established over a publicly observable medium (Mantel and Sabelfeld 2003; Sabelfeld and Mantel 2002). Though sharing our underlying aim of studying the distribution of code under information flow policies, Zdancewic *et al.* (2002) posed the problem in a very different manner: they considered a distributed system of potentially corrupted hosts with principals that have different levels of trust on these hosts, and proposed a way of partitioning and distributing a program over that setting. This work is supported on the Decentralised Label Model (Myers and Liskov 2000), where data is owned by sets of principals, each of which can affect the security label that is associated with that piece of data. More recently, in a work by the first author (Almeida Matos 2009), the problem of declassification control was studied for a distributed language with 'allowed flow policies' associated with each site, but in a simpler memory model where resources are globally shared.

Progressing along rather different lines, there is a field of work on mobile calculi that are purely functional concurrent languages. To mention a few representative papers on process calculi, there is Honda *et al.* (2000) for π-calculus, and Hennessy and Riely (2002) for the security π-calculus. In Kırlı (2000), the mobility of functions as values is studied for a deterministic language with only two sites.

Non-interference for Boxed Ambients.

Castagna, Bugliesi and Crafa seem to have been the first to approach the study of non-interference for a language with both distribution and mobility (Crafa *et al.* 2002). We will conclude this review of related work with a short discussion of their work, which was done for Boxed Ambients (Bugliesi *et al.* 2001) (abbreviated to BA), which form a purely functional process calculus (that is, without side-effects) derived from Mobile Ambients (Cardelli and Gordon 2000), which the authors had previously used as a framework for distributed resource access security (Bugliesi *et al.* 2004). Non-interference is stated by means of a contextual equivalence, and a sound type system is presented. However, they used a unique lattice representing the flow policy, and no declassification mechanisms were contemplated.

Distribution in BA is hierarchical, with mobility consisting of *ambients* entering or exiting the boundaries of neighbouring or parent ambients. Communication can occur locally via an unnamed channel, or across boundaries between parent and child using a channel with the child's name. The execution of both the communication and migration instructions depends on the presence of ambients at certain (neighbouring) positions, and is otherwise suspended (in the same sense that our dereference and assignment operations suspend in the absence of the reference they want to access).

Since ambient names correspond at the same time to places of computation, migrating entities and resources for passing values, and because it is purely functional, it is hard to establish a correspondence between BA and our language. However, some analogies can be drawn. Roughly speaking, security levels are associated with ambient names, as they are here with references and threads. As in our work here, a knowledge of the

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

position of an ambient of level l is considered as l-level information. Message passing between parent and child involves a synchronisation that respects the position of those two domains, as happens in our case for accesses to foreign references. Migration is also identified as a way of revealing the position of 'high-ambients' to lower levels, though the dangerous usages of migration are rejected in a rather different way. Some examples that pinpoint similarities between the two studies can be found in the first author's Ph.D. thesis (Almeida Matos 2006).

7. Conclusion

In this section we summarise the main technical contributions of our work, and conclude with some suggestions for future work.

Security policies We have addressed the issue of what it means for a program to be secure in a network from the point of view of confidentiality in information flow. To this end, we have proposed and studied the non-disclosure property for networks, which determines the absence of information flows that are insecure according to a dynamically chosen ordering of security levels in a setting where the location of processes plays a crucial role. The formalisation of non-disclosure for networks is largely independent of the particular language considered here, and should be easily adaptable to other imperative distributed models with a flat structure of computation domains.

Computation models We have considered a distributed setting with thread migration, where threads execute in different domains, and the relative location of threads and resources determines the circumstances in which they can execute. We found that new forms of security leaks – the migration leaks – can be encoded. Similarities to information flow issues that appear in ambient-like networks (Crafa *et al.* 2002) seem to indicate that this problem is not confined to our particular model. This point has been further confirmed in a recent work by the first author (Almeida Matos 2009) in which migration leaks are shown to appear in a setting with a globally shared memory, and where distribution resides in the concept of a site's 'allowed flow policy'. This is a first step in the direction of introducing membrane computation (Boudol 2005a) as a prerequisite for a thread to enter a domain. It would be interesting to explore further how new ways of controlling information leaks can be obtained from considering more complex models of global computing.

Language features The language we have proposed in this study is simple but expressive. It results from adding a flow declaration construct, a notion of domain and a migration instruction to an imperative higher-order lambda-calculus with thread and reference creation. The flow declaration construct allows a dynamic customisation of the security ordering and the migration instruction changes the position of threads and their references. By including flow declarations in our study of information flow control for networks, we have shown their robustness when used in new computation settings.

Enforcement Mechanisms To enforce our security policy, we have presented a new type and effect system, which can be used to enforce non-disclosure for networks in a

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

decentralised manner. It also differs from Almeida Matos and Boudol (2009) by restricting declassification to occur by means of declassification operations contained within a flow declaration. We have thus highlighted the distinction between our new declassification paradigm and the more common declassification by value downgrading. The type soundness proofs, which we have explained in detail, result from the generalisation of a proof mechanism that was first used in Boudol and Castellani (2002), and later pursued in Almeida Matos and Boudol (2009), Boudol (2005b) and Almeida Matos (2009). We have thus given further evidence that our proof mechanism can be adapted to other settings, and hope that the explanations given in this paper can provide good support in that direction.

The topics of declassification and mobility in information flow are more or less independent problems, so it is perhaps not surprising that the two can be combined with little technical effort. However, it should be noted that this facility is rooted in the highly decentralised nature of the flow declarations. No global agreement is assumed about the flow policies for declassification. Moreover, the changes to the flow policy that are dynamically performed by programs have a local scope and do not affect the whole system.

The potential dangers opened up by allowing declassification in a mobile setting might seem more striking than the advantages it offers. One can imagine the example of a migrating thread that declares a very permissive flow policy for its own execution, and once it arrives at a domain where another thread that owns secret references is computing, it can declassify that information. This could be encoded in our language as follows:

$$d_1[\ (\text{goto}\ d_2); (\text{flow}\ H \prec L\ \text{in}\ (m.b_L :=^{?} (?\ n.a_H)))^{m}\ ]\ \ \|\ \ d_2[\ N^{n}\ ]. \tag{33}$$

According to non-disclosure for networks, the above program is secure – in fact, thread m complies with the declared flow policy when copying the value of the reference $n.a_H$ to $m.b_L$. This is not surprising since flow declarations are a means of extending the flexibility of what is allowed to be done, and not a way of restricting it; furthermore, in this setting we do not consider any access control features that take into account the ownership of information. Indeed, the control of the usage of declassification operations such as the flow declarations is beyond the scope of non-disclosure, which focuses purely on the compliance of a program with the declared flow policies. However, the location in the program where declassifications might occur, and the security levels between which information can flow are delimited in the program, which could enable other forms of security analysis.

One can see the potential for formulating other security properties concerned with the context in which declassification is performed. Enforcement of such properties could be obtained by designing language constructs that constrain the execution of programs to comply with more strict flow policies, or even by setting up fire-wall-like conditions that control the migration of mobile threads. This direction is pursued in a recent paper by the the first author (Almeida Matos 2009), where a new security policy called 'Confinement' is studied in a setting where different 'allowed' flow policies are associated with computation sites. In that setting, confinement is formulated as a property requiring

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

flow declarations to comply with the flow policies allowed by the locations where they are executed. Almeida Matos (2009) also studies language-based solutions for controlling the migration of threads according to the flow declarations that are performed, and for enabling a program to offer alternative behaviours to be followed in contexts where flow declarations are forbidden.

Dual to the above example is the example of a mobile thread that brings its own data to some site where it should perform private computations. Then the possibility of a migrating thread declaring its own flow policies turns into an advantage. For instance, we could write the program

$$d_1[\,(\text{goto } d_2);(\text{flow } H \prec L \text{ in } (n.b_L :=^{?} (?\ m.a_H)))^{m}\,] \;\parallel\; d_2[\,N^{n}\,]. \tag{34}$$

These two examples point to the potential benefits of taking into account the ownership of information when considering the control of declassification in distributed settings.

There has been little practical experience in using mobile computing systems, which makes it hard to evaluate the relevance of allowing declassification in a mobile computing setting. Nevertheless, declassification seems to be a crucial feature in any language that is subject to information flow control, which *a fortiori* justifies the option of including it in our mobile language. We believe that the mobile language presented in this paper, being simple yet expressive, is a good starting point for building more complex frameworks that could bring new views on how to tackle issues that are raised by declassification, as well as to the study of secure information flow in networks in general.

Acknowledgments

The first author would like to thank her supervisors Gérard Boudol and Ilaria Castellani for their guidance, since a significant part of this work was performed during her doctoral studies. Thanks are also due to Matthew Hennessy, Tamara Rezk and the anonymous referees for insightful comments at different stages of this work.

References

Almeida Matos, A. (2005) Non-disclosure for distributed mobile code. In: Ramanujam, R. and Sen, S. (eds.) FSTTCS'05: 25th International Conference on Foundations of Software Technology and Theoretical Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **3821** 177–188.

Almeida Matos, A. (2006) *Typing secure information flow: declassification and mobility*, Ph.D. thesis, École Nationale Supérieure des Mines de Paris.

Almeida Matos, A. (2009) Flow-policy awareness for distributed mobile code. In: Bravetti, M. and Zavattaro, G. (eds.) Proceedings of CONCUR 2009 – Concurrency Theory 20th International Conference. *Springer-Verlag Lecture Notes in Computer Science* **5710** 53–68.

Almeida Matos, A. and Boudol, G. (2009) On declassification and the non-disclosure policy. *Journal of Computer Security* **17** (5) 549–597.

Almeida Matos, A., Boudol, G. and Castellani, I. (2007) Typing noninterference for reactive programs. *The Journal of Logic and Algebraic Programming* **72** (2) 124–156. (Special Issue on Programming Language Interference and Dependence.)

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Bell, D. E. and La Padula, L. J. (1976) Secure computer system: Unified exposition and multics interpretation, Technical Report MTR-2997, The MITRE Corporation.

Boudol, G. (2004) ULM: A core programming model for global computing. In: Schmidt, D. A. (ed.) ESOP'04: 13th European Symposium on Programming. *Springer-Verlag Lecture Notes in Computer Science* **2986** 234–248.

Boudol, G. (2005*a*) A generic membrane model. In: Priami, C. and Quaglia, P. (eds.) GC'04: IST/FET International Workshop on Global Computing. *Springer-Verlag Lecture Notes in Computer Science* **3267** 208–222.

Boudol, G. (2005*b*) On typing information flow. In: Hung, D. V. and Wirsing, M. (eds.) ICTAC'05: Second International Colloquium on Theoretical Aspects of Computing. *Springer-Verlag Lecture Notes in Computer Science* **3722** 366–380.

Boudol, G. and Castellani, I. (2002) Noninterference for concurrent programs and thread systems. *Theoretical Computer Science* **281**(1-2) 109–130.

Bugliesi, M., Castagna, G. and Crafa, S. (2001) Boxed ambients. In: Kobayashi, N. and Pierce, B. C. (eds.) TACS'01: 4th International Symposium on Theoretical Aspects of Computer Software. *Springer-Verlag Lecture Notes in Computer Science* **2215** 38–63.

Bugliesi, M., Castagna, G. and Crafa, S. (2004) Access control for mobile agents: The calculus of boxed ambients. *ACM Trans. Program. Lang. Syst.* **26** (1) 57–124.

Cardelli, L. and Gordon, A. D. (2000) Mobile ambients. *Theoretical Computer Science* **240** (1) 177–213.

Cohen, E. (1977) Information transmission in computational systems. In: *SOSP'77: sixth ACM Symposium on Operating Systems Principles*, ACM Press 133–139.

Crafa, S., Bugliesia, M. and Castagna, G. (2002) Information flow security for boxed ambients. In: Sassone, V. (ed.) F-WAN'02: Workshop on Foundations of Wide Area Network Computing. *Electronic Notes in Theoretical Computer Science* **66** 76–97.

Dal Zilio, S. (2001) Mobile processes: A commented bibliography. In: Cassez, F., Jard, C., Rozoy, B. and Ryan, M. D. (eds.) MOVEP'00: 4th Summer School on Modeling and Verification of Parallel Processes. *Springer-Verlag Lecture Notes in Computer Science* **2067**.

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

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

Goguen, J. A. and Meseguer, J. (1982) Security policies and security models. In: *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, IEEE Computer Society 11–20.

Hennessy, M. and Riely, J. (2002) Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems* **24** (5) 566–591.

Honda, K., Vasconcelos, V. T. and Yoshida, N. (2000) Secure information flow as typed process behaviour. In: Smolka, G. (ed.) ESOP'00: 9th European Symposium on Programming. *Springer-Verlag Lecture Notes in Computer Science* **1782** 180–199.

Honda, K. and Yoshida, N. (2002) A uniform type structure for secure information flow. In: *POPL'02: 29th ACM Symposium on Principles of Programming Languages*, ACM Press 81–92.

Kırlı, D. (2000) Mobile functions and secure information flow. In: Degano, P. (ed.) WITS'00: Workshop on Issues in the Theory of Security.

Lucassen, J. M. and Gifford, D. K. (1988) Polymorphic effect systems. In: *POPL'88: 15th ACM symposium on Principles of programming languages*, ACM Press 47–57.

Mantel, H. and Sabelfeld, A. (2003) A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security* **11** (4) 615–676.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997) *The definition of Standard ML*, revised edition, MIT Press.

Myers, A. C. (1999) JFlow: Practical mostly-static information flow control. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM Press 228–241.

Myers, A. C. and Liskov, B. (1998) Complete, safe information flow with decentralized labels. In: *19th IEEE Computer Society Symposium on Security and Privacy*, IEEE Computer Society 186–197.

Myers, A. C. and Liskov, B. (2000) Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology* **9** (4) 410–442.

Sabelfeld, A. (2001) The impact of synchronisation on secure information flow in concurrent programs. In: Bjørner, D., Broy, M. and A. V. Zamulin, (eds.) PSI'01: 4th International Andrei Ershov Memorial Conference on Perspectives of System Informatics. *Springer-Verlag Lecture Notes in Computer Science* **2244** 225–239.

Sabelfeld, A. and Mantel, H. (2002) Securing communication in a concurrent language. In: Hermenegildo, M. V. and Puebla, G. (eds.) SAS'02: 9th International Symposium on Static Analysis. *Springer-Verlag Lecture Notes in Computer Science* **2477** 376–394.

Sabelfeld, A. and Myers, A. (2004) A model for delimited information release. In: International Symposium on Software Security (ISSS'03). *Springer-Verlag Lecture Notes in Computer Science* **3233**.

Sabelfeld, A. and Myers, A. C. (2003) Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* **21** (1) 5–19.

Sabelfeld, A. and Sands, D. (2000) Probabilistic noninterference for multi-threaded programs. In: *CSFW'00: 13th IEEE Computer Security Foundations Workshop*, IEEE Computer Society 200–215.

Sabelfeld, A. and Sands, D. (2005) Dimensions and principles of declassification. In: *CSFW'05: 18th IEEE Computer Security Foundations Workshop*, IEEE Computer Society 255–269.

Sekiguchi, T. and Yonezawa, A. (1997) A calculus with code mobility. In: *FMOODS'97: IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, Chapman and Hall 21–36.

Smith, G. (2001) A new type system for secure information flow. In: *CSFW'01: 14th IEEE Computer Security Foundations Workshop*, IEEE Computer Society 115–125.

Volpano, D. M., Smith, G. and Irvine, C. E. (1996) A sound type system for secure flow analysis. *Journal of Computer Security* **4** (2-3) 167–188.

Wright, A. K. and Felleisen, M. (1994) A syntactic approach to type soundness. *Information and Computation* **115** (1) 38–94.

https://doi.org/10.1017/S096012951100017X Published online by Cambridge University Press