Locality-based Security Policies [⋆]

Terkel K. Tolstrup[1], Flemming Nielson[1], and René Rydhof Hansen[2]

[1] Informatics and Mathematical Modelling, Technical University of Denmark
{tkt,nielson}@imm.dtu.dk
[2] Department of Computer Science, University of Copenhagen
rrhansen@diku.dk

Abstract. Information flow security provides a strong notion of end-to-end security in computing systems. However sometimes the policies for information flow security are limited in their expressive power, hence complicating the matter of specifying policies even for simple systems. These limitations often become apparent in contexts where confidential information is released under specific conditions.
We present a novel policy language for expressing permissible information flow under expressive constraints on the execution traces for programs. Based on the policy language we propose a security condition shown to be a generalized intransitive non-interference condition. Furthermore a *flow-logic* based static analysis is presented and shown capable of guaranteeing the security of programs analysed.

1 Introduction

The number of computing devices with built-in networking capability has experienced an explosive growth over the last decade. These devices range from the highly mobile to the deeply embedded and it has become standard for such devices to be "network aware" or even "network dependent" in the sense that these devices can use a wide variety of networking technologies to connect to almost any kind of computer network. Consequently modern software is often expected to utilise resources and services available over the network for added functionality and user collaboration. In such an environment where devices routinely contain highly sensitive or private information and where information flow is complex and often unpredictable it is very challenging to maintain the confidentiality of sensitive information. Predictably it is even more challenging to obtain formal guarantees or to formally verify that a given device or system does not leak confidential information. The problem is further exacerbated by the often complicated and ever changing security requirements of users. Examples include a user's medical records that should be inaccessible unless the user is at a hospital, or personal financial information that may be accessed by a bank or a financial advisor but not by the tax authorities except during a tax audit. The above examples expose one of the major drawbacks of traditional approaches to secure information flow, namely the lack of support for dynamic

[⋆] This work has in part been supported by the EU research project #016004, Software Engineering for Service-Oriented Overlay Computers.

Localities		Nets	
$\ell ::= l$	locality	$N ::= l :: P$	single node
$\mid\ u$	locality variable	$\mid\ l :: \langle et \rangle$	located tuple
		$\mid\ N_1 \parallel N_2$	net composition
Processes		Actions	
$P ::= \mathbf{nil}$	null process	$a ::= \mathbf{out}(t)@\ell$	output
$\mid\ a.P$	action prefixing	$\mid\ \mathbf{in}(T)@\ell$	input
$\mid\ P_1 \mid P_2$	parallel composition	$\mid\ \mathbf{read}(T)@\ell$	read
$\mid\ A$	process invocation	$\mid\ \mathbf{eval}(P)@\ell$	migration
		$\mid\ \mathbf{newloc}(u)$	locality creation
Tuples		Fields	
$T ::= F \mid F, T$	templates	$et ::= ef \mid ef, et$	evaluated tuple
$F ::= f \mid\ !x \mid\ !u$	template fields	$ef ::= V \mid l$	evaluated tuple field
$t ::= f \mid f, t$	tuples	$e ::= V \mid x \mid \ldots$	expressions
$f ::= e \mid l \mid u$	tuple fields		

Fig. 1. Syntax.

and flexible security policies. Even formulating, let alone formalising, an information flow policy for such diverse uses and changing requirements seems to be an insurmountable problem for the traditional approaches where lattice-based policies are formalised and enforced by *non-interference*. This has recently led researchers to look for better and more appropriate ways of specifying information flow policies and their concomitant notions of secure information flow, incorporating concepts such as *downgrading* (or *declassification*), *delimited release*, and *non-disclosure*.

In this paper we develop a novel notion of *locality-based security policies* in conjunction with a strong security condition for such policies: *History-based Release*. The locality-based security policies are powerful and flexible enough to be used in systems with a high degree of network connectivity and network based computing such as those described above. In this paper we model such systems in the μKlaim calculus, which is based on the *tuple space* paradigm, making it uniquely suited for our purposes. In addition we define what we believe to be the first tuple-centric notion of non-interference and show how History-based Release is a strict generalisation. Finally we construct a static analysis for processes modelled in μKlaim and demonstrate how it can be used to formally verify *automatically* that a given system is secure with respect to a given locality-based security policy. Such an analysis is an invaluable tool, both when designing and when implementing a complex system, and can be used to obtain security guarantees that are essential for critical systems.

2 The μKlaim Calculus

The Klaim family of process calculi were designed around the notion of a *tuple space*. In this paradigm systems are composed of a set of *nodes* distributed at

$$\begin{array}{rcl} N_1 \,||\, N_2 & \equiv & N_2 \,||\, N_1 \\ (N_1 \,||\, N_2) \,||\, N_3 & \equiv & N_1 \,||(N_2 \,||\, N_3) \\ l :: P & \equiv & l :: (P \,|\, \mathbf{nil}) \\ l :: A & \equiv & l :: P \qquad \text{if } A \triangleq P \\ l :: (P_1 \,|\, P_2) & \equiv & l :: P_1 \,||\, l :: P_2 \end{array}$$

Fig. 2. Structural congruence.

$$\mathrm{match}(V, V) = \epsilon \quad \mathrm{match}(!x, V) = [V/x]$$
$$\mathrm{match}(l, l) = \epsilon \qquad \mathrm{match}(!u, l') = [l'/u]$$
$$\frac{\mathrm{match}(F, ef) = \sigma_1 \qquad \mathrm{match}(T, et) = \sigma_2}{\mathrm{match}((F, T), (ef, et)) = \sigma_1 \circ \sigma_2}$$

Fig. 3. Tuple matching.

various *localities*. The nodes communicate by sending and receiving *tuples* to and from various tuple spaces based at different localities. Mobility in the Klaim calculi is modelled by remote evaluation of processes. In the standard tuple space model a tuple space is a resource shared among peers and therefore no attempt is made to restrict or control access to these.

The μKlaim calculus [8] comprises three parts: nets, processes, and actions. Nets give the overall structure in which tuple spaces and processes are located. Processes execute by performing actions. The syntax is shown in Fig. 1. Processes execute by performing an action, a, or by "invocation" of a process place-holder variable. The latter is used for iteration and recursion. Processes can be composed in parallel and finally a process can be the **nil**-process representing the inactive process. The following actions can be performed by a process. The **out**-action outputs a tuple into a tuple space at a specific locality; the **in** and **read** actions input a tuple from a specific tuple space and either remove it or leave it in place respectively; the **eval**-action remotely evaluates a process at a specified locality; and **newloc** creates a new locality.

The semantics for μKlaim, shown in Fig. 4, is a straightforward operational semantics and we shall not go into further detail here but refer instead to [8]. As is common for process calculi the semantics incorporates a structural congruence, see Fig. 2. In Fig. 3 the semantics for tuple matching, as used in the rules for the **in**- and **read**-actions, is shown. We assume the existence of a semantic function for evaluating tuples denoted as $[\![\cdot]\!]$.

Semantically we define an execution trace as the sequence of locations where processes are executed when evaluating the processes. Hence we write $L \vdash N \overset{l}{\rightarrowtail} L' \vdash N'$ when evaluating one step of a process located at the location l. Clearly the execution originates from an action being evaluated at the process space of location l. For the transitive reflexive closure we write $L \vdash N \overset{\omega}{\rightarrowtail}^{*} L' \vdash N'$ where $\omega \in \Omega$ is a string of locations.

3 Policies for Security

To protect confidentiality within a system, it is important to control how information flows so that secret information is prevented from being released on unintended channels. The allowed flow of information in a system is specified in a security policy. In this section we present a policy language based on graphs. Here vertices represent security domains, describing the resources available in the

$$\frac{\text{match}([\![T]\!], et) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle et \rangle \overset{l}{\rightarrowtail} l :: P\sigma \parallel l' :: \mathbf{nil}}$$

$$\frac{[\![t]\!] = et}{l :: \mathbf{out}(t)@l'.P \parallel l' :: P' \overset{l}{\rightarrowtail} l :: P \parallel l' :: P' \parallel l' :: \langle et \rangle}$$

$$\frac{L \vdash N_1 \overset{l}{\rightarrowtail} L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \overset{l}{\rightarrowtail} L' \vdash N_1' \parallel N_2}$$

$$\frac{\text{match}([\![T]\!], et) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle et \rangle \overset{l}{\rightarrowtail} l :: P\sigma \parallel l' :: \langle et \rangle}$$

$$\frac{N \equiv N_1 \qquad L \vdash N_1 \overset{l}{\rightarrowtail} L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \overset{l}{\rightarrowtail} L' \vdash N'}$$

$$l :: \mathbf{eval}(Q)@l'.P \ \parallel\ l' :: P' \overset{l}{\rightarrowtail} l :: P \parallel l' :: P' \parallel l' :: Q$$

$$\frac{l' \notin L \qquad \lfloor l' \rfloor = \lfloor u \rfloor}{L \vdash l :: \mathbf{newloc}(u).P \overset{l}{\rightarrowtail} L \cup \{l'\} \vdash l :: P[l'/u] \parallel l' :: \mathbf{nil}}$$

Fig. 4. Operational semantics for μKlaim.

net. A security domain is related to sets of resources available in the considered system.

This model allows for the granularity of the mapping from resources to security domains to be very fine-grained. For example we can introduce a security domain for each location. For improved precision we could partition the usage of a location into lifetime periods and introduce a domain for each, hence having more than one security domain for each location in the net. This would allow us to abstract from e.g. reuse of limited resources in the implementation. For further discussion of this see [23].

In our setting the tuple spaces are our resources, hence the localities are related to security domains. This allows us to reason about groups of localities together, as well as singling out specific localities and isolate these in their own security domains. The security policies therefore focus on the information flow between intended domains of localities. Consequently we assume that locations can be uniquely mapped to security domains.

Definition 1. *(Security Domains) For a given net we have a mapping from localities L to security domains V*

$$\underline{\cdot} : L \ \rightarrow \ V$$

We write $\underline{l}$ for the security domain of the location l.

Edges specify permitted flows of information. Information flow between resources can be restricted subject to fulfillment of constraints with respect to certain events taking place prior to the flow. Formally we propose the following definition of security polices.

Definition 2. *(Locality-based security policies) A security policy is a labelled graph $G = (V, \lambda)$, consisting of a set of vertices V representing security domains and a total function λ mapping pairs of vertices to labels $\lambda : V \times V \to \Delta$. We define $\mathbb{G}$ to be the set of policies. The structure, Δ, of labels is defined below.*

In a flow graph the set of vertices V represent security domains. A security domain indicates the usage of a resource in the system. The edges in the flow graph describe the allowed information flow between resources. Hence an edge from the vertex for security domain v_1 to the vertex for security domain v_2 indicates that information is allowed to flow between the resources in these domains subject to the label attached to the edge.

The edges in the flow graph are described by the function $\lambda : V \times V \to \Delta$. We write $v_1 \overset{\delta}{\rightsquigarrow} v_2$ for an edge from vertex v_1 to v_2 constrained by $\delta \in \Delta$, i.e. $\lambda(v_1, v_2) = \delta$. Information flow might be constrained by certain obligations that the system must fulfill before the flow can take place. Here we describe a novel constraint language that allows the security policy to be specific about intransitive information flows in the flow graph. Constraints are specified in the following syntax $\delta \in \Delta$:

$$\delta ::= \mathit{true} \mid \mathit{false} \mid v \mid \delta_1 \cdot \delta_2 \mid \delta_1 \wedge \delta_2 \mid \delta_1 \vee \delta_2 \mid \delta^*$$

A constraint may be trivially *true* or *false*. The constraint v enforces that flows occur at specific locations, thus that the flow is only permitted at locations that is part of the security domain v (i.e. $\underline{l} = v$). The constraint $\delta_1 \cdot \delta_2$ enforces that the flow is only allowed to occur if events described in δ_1 precedes events described in δ_2. Common logical operators $\wedge$ and $\vee$ are available to compose constraints. Finally Kleene's star * allows the policies to express cyclic behaviour.

We might omit the constraint, writing $v_1 \rightsquigarrow v_2$ for $v_1 \overset{\mathit{true}}{\rightsquigarrow} v_2$. Similarly we might omit an edge in a flow graph indicating that the constraint can never be fulfilled, i.e. $v_1 \overset{\mathit{false}}{\rightsquigarrow} v_2$.

Example 1. To illustrate the usage of flow graphs as security policies we here discuss the three examples given in Figure 5. The first flow graph (a) allows a flow from v_1 to v_2 and v_2 to v_3 but not from v_1 to v_3. That is neither directly nor through v_2! In this manner the intransitive and temporal nature of the policies allow us to have constraints on the order of information flows.

The second flow graph (b) allows the flow from v_1 to v_2, v_2 to v_3 and from v_1 to v_3 as well. The flow can be directly from v_1 to v_3 or through v_2. If we wish to restrict the flow to go through v_2 it could be done as in flow graph (c). We assume that $\underline{l}$ and $\underline{l'}$ map to security domains that no other locations are mapped to. Hence the last flow graph (c) restricts the flows between the security domains to certain locations. This ensures that for information to flow from v_1 to v_3 both locations need to participate. ■

To give intuition to the above example policies we relate them back to the personal finance scenario mentioned in the introduction. In the following we let

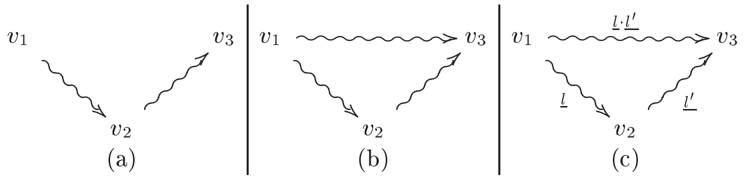

Fig. 5. Examples of flow graphs

v_1, v_2, and v_3 denote the user, the user's financial advisor, and the tax authorities respectively. The first policy (Figure 5(a)) states that the user's financial information may be accessed by the financial advisor but *not* by the tax authorities while still allowing the financial advisor to send (other) information to the tax authorities. The second policy (Figure 5(b)) then states that the user's financial information may be accessed both by the financial advisor and by the tax authorities; this may be necessary during a tax audit. Finally the policy shown in (Figure 5(c)) defines a situation where the user's financial information may be accessed by both the financial advisor and the tax authorities but *only* through the financial advisor; this security policy ensures that the financial advisor can review all relevant information from the user before the tax authorities gain access to it.

A system is given by a net N and a security policy G together with a mapping $\underline{\cdot}$ and might be written N subject to $(G, \underline{\cdot})$. However, as the G and $\underline{\cdot}$ components are clear from context we choose not to incorporate the policies in the syntax of μKlaim.

3.1 Semantics of constraints

In this subsection we present the semantics of our security policy language. The semantics is given as a translation to regular expressions over execution traces. The intuition is that if an execution trace is in the language of the regular expression derived by the semantics, then the constraint is fulfilled.

The main idea behind the locality-based security policies is that they specify constraints that must be fulfilled rather than the total behaviour of the system. This result in succinct policies. Therefore the semantics of constraints is based on an understanding of fulfilment of a constraint whenever an execution trace produces the locations specified. Hence if a trace that fulfills a constraint is preceded or succeeded by other traces the constraint remains fulfilled.

The *true* constraint is always fulfilled and hence is translated to the regular expression L^* accepting all execution traces. Similarly the constraint *false* cannot be fulfilled for any trace, and hence the generated language is $\emptyset$. The constraint v gives the language $L^* \cdot \{l \mid \underline{l} = v\} \cdot L^*$ as we wish to allow the flow only for executions taking place at l. The constraint $\delta_1 \cdot \delta_2$ indicates that the trace ω can be split into ω_1 and ω_2, where ω_1 must be in the language of δ_1, and respectively ω_2 must be in the language of δ_2. The constraints for $\delta_1 \wedge \delta_2$ and $\delta_1 \vee \delta_2$ are straightforward. One obvious definition of $[\![\delta^*]\!]$ is $[\![\delta]\!]^*$; however this choice would invalidate Lemma 1 below. Consequently, it is natural to define $[\![\delta^*]\!] = L^* \cdot [\![\delta]\!]^* \cdot L^*$ as then Lemma 1 continues to hold.

The semantics of the security policy language are given in Figure 6.

$$\begin{aligned}
[\![true]\!] &= L^* \\
[\![false]\!] &= \emptyset \\
[\![v]\!] &= L^* \cdot \{l \mid \underline{l} = v\} \cdot L^* \\
[\![\delta_1 \cdot \delta_2]\!] &= [\![\delta_1]\!] \cdot [\![\delta_2]\!] \\
[\![\delta_1 \wedge \delta_2]\!] &= [\![\delta_1]\!] \cap [\![\delta_2]\!] \\
[\![\delta_1 \vee \delta_2]\!] &= [\![\delta_1]\!] \cup [\![\delta_2]\!] \\
[\![\delta^*]\!] &= L^* \cdot [\![\delta]\!]^* \cdot L^*
\end{aligned}$$

Fig. 6. Semantics of constraints

Lemma 1. *The semantical interpretation of constraint δ does not change by preceding or succeeding it by other traces* $\forall \delta : [\![\delta]\!] = L^* \cdot [\![\delta]\!] \cdot L^*$

We define an ordering of constraints as the relation $\leq_\Delta \subseteq \Delta \times \Delta$, i.e. we say that δ is a restriction of δ' if $(\delta, \delta') \in \leq_\Delta$, normally we write $\delta \leq_\Delta \delta'$.

Definition 3. *We say that a constraint δ is a restriction of δ', written $\delta \leq_\Delta \delta'$ if we have $[\![\delta]\!] \subseteq [\![\delta']\!]$.*

Similarly we define a restriction relation between flow graphs.

Definition 4. *We say that a flow graph $G = (V, \lambda)$ is a restriction of $G' = (V', \lambda')$, written $G \leq G'$ if we have that*

$$V = V' \ \wedge \ \forall v_1, v_2 : \lambda(v_1, v_2) \leq_\Delta \lambda'(v_1, v_2)$$

4 Security Condition

To determine whether a program is secure or not, we need some condition for security. In this section we therefore present our definition of secure nets. The intuition is similar to that of non-interference, however we aim to generalize the traditional non-interference condition to permit release of confidential information, based on constraints on the history of the execution and it's present location. We call this condition *History-based Release.*

4.1 Security Condition

Before we formalize the main security condition we need to formalize what an attacker can observe. Consider an attacker that has access to a subset $\mathcal{V}$ of the tuplespaces that are available in the net under consideration. We formalize the observable part of a net by *nullifying* the tuple spaces that the attacker cannot observe.

Definition 5. *($\mathcal{V}$-observable) The $\mathcal{V}$-*observable *part of a net N written $N|_\mathcal{V}$ is*

$$\begin{aligned}
(l :: P)|_\mathcal{V} &= l :: P \\
(l :: \langle et \rangle)|_\mathcal{V} &= \begin{cases} l :: \langle et \rangle & \textit{if } l \in \mathcal{V} \\ l :: nil & \textit{otherwise} \end{cases} \\
(N_1 \parallel N_2)|_\mathcal{V} &= N_1|_\mathcal{V} \parallel N_2|_\mathcal{V}
\end{aligned}$$

Furthermore we assume that the attacker has knowledge of all the processes that exist in the net, and hence can reason about the absence of a tuple at a given location. Similar to the probabilistic attacker in [25] it is feasible to assume that two tuple spaces can be compared on all tuples. Thus for two nets N_1 and N_2 an attacker that observes at security domain $\mathcal{V}$ can compare the observable parts of the nets as $N_1|_{\mathcal{V}} \sim N_2|_{\mathcal{V}}$.

Definition 6. *(Observable equivalence) Two nets N_1 and N_2 are observably equivalent $N_1 \sim N_2$ iff*

$$\{\!\!\{(l, \langle et \rangle) \mid N_1 = (\cdots \parallel l :: \langle et \rangle \parallel \cdots)\}\!\!\} = \{\!\!\{(l, \langle et \rangle) \mid N_2 = (\cdots \parallel l :: \langle et \rangle \parallel \cdots)\}\!\!\}$$

where we write $\{\!\!\{\cdot\}\!\!\}$ for a multi-set.

We define the function $\nabla : \mathcal{P}(V) \times \mathbb{G} \times \Omega \to \mathcal{P}(V)$ for extending a set of security domains with the domains that are permitted to interfere with the observed domains due to the fulfillment of constraints by the execution trace ω. The resulting set of security domains describe the permutted information flows during the execution.

Definition 7. *For a security policy G and a execution trace ω an observer at $\mathcal{V}$ can observe the localities*

$$\nabla(\mathcal{V}, G, \omega) = \mathcal{V} \cup \{v_1 \mid v_2 \in \mathcal{V} \wedge \omega \in [\![\lambda(v_1, v_2)]\!]\}$$

A less restrictive policy, ∇ will never reduce the observable part of the net. This allows us to establish the following fact.

Fact 1. *If $G \leq G'$ then $\nabla(\mathcal{V}, G, \omega) \subseteq \nabla(\mathcal{V}, G', \omega)$.*

We consider a program secure if in no execution trace, neither a single step nor a series of steps, will an attacker observing the system at the level of a set of security domains $\mathcal{V}$ be able to observe a difference at any locality, when all resources permitted to interfere with the locality is observably equivalent before the evaluation. We formalize the condition as a bisimulation over execution traces on nets.

Definition 8. *(Bisimulation) A $(G, \mathcal{V})$-bisimulation is a symmetric relation $\mathcal{R}$ on (the process part of) nets whenever*

$$\begin{array}{l} L_1 \vdash N_1 \overset{\omega}{\rightarrowtail}{}^{*} L'_1 \vdash N'_1 \;\wedge \\ L_1 \vdash N_1|_{\emptyset} \;\mathcal{R}\; L_2 \vdash N_2|_{\emptyset} \;\wedge \\ N_1|_{\nabla(\mathcal{V},G,\omega)} \sim N_2|_{\nabla(\mathcal{V},G,\omega)} \end{array}$$

then there exists N'_2, L'_2 and ω' such that

$$\begin{array}{l} L_2 \vdash N_2 \overset{\omega'}{\rightarrowtail}{}^{*} L'_2 \vdash N'_2 \;\wedge \\ L'_1 \vdash N'_1|_{\emptyset} \;\mathcal{R}\; L'_2 \vdash N'_2|_{\emptyset} \;\wedge \\ N'_1|_{\mathcal{V}} \sim N'_2|_{\mathcal{V}} \end{array}$$

We use the fact that the observable part of a net projected on an empty set of security domains $L \vdash N|_\emptyset$ gives the process part of the net. The reason why we define the bisimulation in this way is to focus on the executable part and not the memory part.

The bisimulation follows the approach of Sabelfeld and Sands [22] in utilizing a *resetting of the state* between subtraces. This follows from modelling the attacker's ability to modify tuple spaces concurrently with the execution. Furthermore it accomodates the dynamically changing nature of the security policies due to the fulfillment of constraints, as seen in [13]. The definition is transitive but not reflexive. That the definition is not reflexive follows from observing that the net $l :: \mathbf{in}(!x)@l_H.\mathbf{out}(x)@l_L$ is not self-similar whenever information is not permitted to flow from l_H to l_L.

Fact 2. *If $G \leq G'$ and $\mathcal{R}$ is a $(G, \mathcal{V})$-bisimulation then $\mathcal{R}$ is also a $(G', \mathcal{V})$-bisimulation.*

Definition 9. *A G-bisimulation is a relation $\mathcal{R}$ such that for all $\mathcal{V}$, $\mathcal{R}$ is a $(G, \mathcal{V})$-bisimulation. Denote the largest G-bisimulation $\approx_G$.*

Now we can define the security condition as a net being bisimilar to itself.

Definition 10. *(History-based Release) A net N is secure wrt. the security policy G if and only if we have $N \ \approx_G \ N$.*

Example 2. In the following we will consider a number of example programs that illustrate the strength of History-based Release. First consider the program

$$l :: \mathbf{in}(!x)@l_1.\mathbf{out}(x)@l_2$$

which reads a tuple from l_1 and writes it to l_2. With the policy $\underline{l_1} \overset{l}{\rightsquigarrow} \underline{l_2}$, the program is secure, while changing the policy to $\underline{l_1} \overset{l'}{\rightsquigarrow} \underline{l_2}$ makes the program insecure. The reason that the second program is insecure is because the bisimulation forces us to consider all possible traces, so even if the above program was modified to execute a process on l' concurrently with the one on l, the result would be the same. This corresponds to *intransitive non-interference* [14] (or *lexically scoped flows* according to [2]).

History-based Release goes beyond lexically scoped flows as the policy might constrain the history of a process. This is illustrated by the following example. Consider the security policy in Fig. 5(c) and assume that $\underline{l_1} = v_1$, $\underline{l_2} = v_2$ and $\underline{l_3} = v_3$, for which the program

$$l :: \mathbf{in}(!x)@l_1.\mathbf{out}(x)@l_2 \quad || \quad l' :: \mathbf{in}(!y)@l_2.\mathbf{out}(y)@l_3$$

is secure. On the other hand the program

$$l :: \mathbf{in}(!x)@l_1.\mathbf{out}(x)@l_2 \quad || \quad l' :: \mathbf{in}(!y)@l_1.\mathbf{out}(y)@l_3$$

is insecure because the process at l' might evaluate prior to the process at l. ■

Example 3. Another concern is the handling of indirect flows. Consider the program

$$l :: \mathbf{in}(a)@l_1.\mathbf{in}(b)@l_2$$

and an attacker observing whether the tuple b is removed from location l_2 or not. Based on this the attacker will know if the process was capable of removing the tuple a from location l_1. Therefore History-based Release allows the program for the policy $\underline{l_1} \overset{l}{\rightsquigarrow} \underline{l_2}$, but not for $\underline{l_1} \overset{false}{\rightsquigarrow} \underline{l_2}$. This is due to the fact that information can be observed by the attacker through the absence of a tuple in a tuple space. ■

Example 4. Finally we wish to look at an example program that is insecure in the traditional setting where lattices are used as security policies. Consider the program

$$l :: \mathbf{in}(!x)@l_2.\mathbf{out}(x)@l_3.\mathbf{read}(!y)@l_1.\mathbf{out}(y)@l_2$$

which is secure for the policy in Fig. 5(a) when $\underline{l_1} = v_1$, $\underline{l_2} = v_2$ and $\underline{l_3} = v_3$. This is because evaluating the program does not result in information flowing from l_1 to l_3. ■

4.2 Consistency of History-based Release

In this subsection we argue the consistency of the definition of History-based Release. In particular we will discuss two of the principles presented by Sabelfeld and Sands in [21]. In the following we consider declassification to refer to constraints that are not trivially evaluated to *true* or *false*.

Conservativity: *Security for programs with no declassification is equivalent to non-interference.*

Limiting all constraints on edges in the flow graphs to only being of the simple form *true*, *false* or v gives us intransitive non-interference. Removing all non-trivial constraints (i.e. only having the constraints *true* and *false*) results in traditional non-interference.

Monotonicity of release: *Adding further declassifications to a secure program cannot render it insecure.*

Adding declassifications to a program coresponds to making our security policies less restrictive. Hence we aim to show that a program will be secure for any policy at least as restrictive as the original policy, for which it can be shown secure.

Lemma 2. *(Monotonicity) If $G \leq G'$ then $N_1 \approx_G N_2 \Rightarrow N_1 \approx_{G'} N_2$.*

Proof: It follows from Fact 1 that $(N_1|_{\nabla(\mathcal{V},G',\omega)} \sim N_2|_{\nabla(\mathcal{V},G',\omega)}) \Rightarrow (N_1|_{\nabla(\mathcal{V},G,\omega)} \sim N_2|_{\nabla(\mathcal{V},G,\omega)})$. The Lemma follows from Fact 2 and observing that to show $N_1 \approx_{G'} N_2$ we have either $N_1|_{\nabla(\mathcal{V},G',\omega)} \sim N_2|_{\nabla(\mathcal{V},G',\omega)}$, in which case we can reuse the proof for $N_1 \approx_G N_2$, or otherwise the result holds trivially. ■

$$
\begin{aligned}
E(\mathbf{nil}) &= \emptyset \\
E(P_1 \mid P_2) &= E(P_1) \cup E(P_2) \\
E(A) &= E(P) \quad \text{if } A \triangleq P \\
E(\mathbf{out}^\iota(t)@\ell.P) &= \{\iota\} \\
E(\mathbf{in}^\iota(T)@\ell.P) &= \{\iota\} \\
E(\mathbf{read}^\iota(T)@\ell.P) &= \{\iota\} \\
E(\mathbf{eval}^\iota(Q)@\ell.P) &= \{\iota\} \\
E(\mathbf{newloc}^\iota(u).P) &= \{\iota\}
\end{aligned}
\qquad
\begin{aligned}
\hat{\sigma}[\![V]\!] &= \{V\} \\
\hat{\sigma}[\![x]\!] &= \hat{\sigma}(x) \\
\hat{\sigma}[\![l]\!] &= \{l\} \\
\hat{\sigma}[\![u]\!] &= \hat{\sigma}(u) \\
\hat{\sigma}[\![f, t]\!] &= \hat{\sigma}[\![f]\!] \times \hat{\sigma}[\![t]\!]
\end{aligned}
$$

Fig. 7. (a) Exposed labels in a process. (b) Extension of $\hat{\sigma}$ to templates.

5 Security Analysis

In this section we present an approach for verifying systems fulfillment of confidentiality wrt. History-based Release specified in a security policy. The analyses are given in the *Flow Logic* framework [18]. Hence the security guarantee is static and performed prior to the deployment of the system considered.

The analyses are based on the approach of Kemmerer [12]. Thus we analyse a system in two steps. First in Section 5.1 we identify the local dependencies; this is done by modifying a *control flow analysis* by introducing a novel component for the synchronization of events allowing it to track implicit flows. Second in Section 5.2 we describe a closure condition of the local dependencies to find the global dependencies. These two steps are independent of the security policy given for the considered system, and only related to the program analysed. The final step is the comparison of the security policy and the flow graph extracted from the program and in Section 5.3 we argue that the security enforced by our approach is History-based Release.

5.1 Local Dependencies

The local dependencies are identified by a *control flow analysis*. In fact we modify the analysis presented in [10] by introducing a novel component for capturing synchronizations performed in Klaim processes. Hence we will only briefly describe the other components, before focusing on the extension. The analysis for the net N is handled by judgements of the form

$$(\hat{T}, \hat{\sigma}, \hat{C}) \models N : \hat{G}$$

The component $\hat{T} : L \rightarrow \mathcal{P}(t)$ is an abstract mapping that associates a location or location variable with the set of tuples that might be present at the tuple space. The component $\hat{\sigma} : T \rightarrow \mathcal{P}(t)$ is an abstract mapping holding all possible bindings of a variable or locality variable (or a pattern of these) that might be introduced during execution. Furthermore we introduce the abstract mapping $\hat{C} : \mathit{Lab} \rightarrow \mathcal{P}(L)$ that associates the label of an action to the set of localities that the process has previously synchronized with during its execution. The labels of actions are introduced below. Finally we collect an abstract flow graph in the component $\hat{G} : L \rightarrow \mathcal{P}(L \times L)$ for describing the flow between tuple spaces and the location at which the process was executed. We write $l \overset{l''}{\mapsto} l'$ when a flow from l to l' occurs at l''.

$$\begin{array}{lll}
\hat{\sigma} \models_i \epsilon : \hat{V}_\circ \triangleright \hat{V}_\bullet & \text{iff} & \{\hat{et} \in \hat{V}_\circ \mid |\hat{et}| = i-1\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i V, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \ \wedge\ \{\hat{et} \in \hat{V}_\circ \mid prj_i(\hat{et}) = V\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i l, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \ \wedge\ \{\hat{et} \in \hat{V}_\circ \mid prj_i(\hat{et}) = l\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i x, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \ \wedge\ \{\hat{et} \in \hat{V}_\circ \mid prj_i(\hat{et}) = \hat{\sigma}(x)\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i u, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \ \wedge\ \{\hat{et} \in \hat{V}_\circ \mid prj_i(\hat{et}) = \hat{\sigma}(u)\} \sqsubseteq \hat{V}_\bullet \\
\hat{\sigma} \models_i !x, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \ \wedge\ \hat{V}_\circ \sqsubseteq \hat{V}_\bullet \ \wedge\ prj_i(\hat{W}_\bullet) \sqsubseteq \hat{\sigma}(x) \\
\hat{\sigma} \models_i !u, T : \hat{V}_\circ \triangleright \hat{W}_\bullet & \text{iff} & \hat{\sigma} \models_{i+1} T : \hat{V}_\bullet \triangleright \hat{W}_\bullet \ \wedge\ \hat{V}_\circ \sqsubseteq \hat{V}_\bullet \ \wedge\ prj_i(\hat{W}_\bullet) \sqsubseteq \hat{\sigma}(u)
\end{array}$$

Fig. 8. Abstract tuple matching.

An indirect flow can occur by synchronizing with a tuple space before synchronizing with another, as the attacker might observe the absence of the second synchronization. For tracking these flows we label the actions in a program. We define the function $E : P \to \mathcal{P}(Lab)$ as the fixpoint of the *exposed* set of labels. This allows us to track which actions have been executed prior to the one considered at present. The function is presented in Fig. 7(a).

When analysing an action we must use $\hat{C}$ to find the locations that it synchronizes with. The reason is that these might block further execution, if their templates can not be matched anything available at the location. Hence if the attacker can observe the result of an action that follows the one considered he will learn of the existence (or absence) of the tuples matched. Therefore for all input or read actions the condition $\forall \iota_P \in E(P) : \hat{C}(\iota) \cup \hat{\sigma}(\ell) \subseteq \hat{C}(\iota_P)$ must be fulfilled, where ι is the label of the considered action, $\hat{\sigma}(\ell)$ is the set of locations synchronized with and P is the remaining part of the process.

All local information flows must be found in $\hat{G}$. Hence whenever an action results in a flow of information we check that it is in $\hat{G}$. There are two actions that result in flow of information. Clearly the **out** action will output information to the specified location, and whether or not this happens will give away whether previous synchronizations were successful or not. Similarly the **in** action will remove a tuple from the specified location, hence we ensure that a flow is recorded in $\hat{G}$. We do so by emposing the condition $[\hat{C}(\iota) \overset{l}{\mapsto} \hat{\sigma}(\ell)] \subseteq \hat{G}$, where ι is the label of the action considered and $\hat{\sigma}(\ell)$ is the set of influenced locations.

The rest of the components of the analysis are the same as in [10]. The analysis is specified in Fig. 9. In Fig. 7(b) we extend the component $\hat{\sigma}$ for application on templates and in Fig. 8 the analysis of abstract tuple matching is presented.

5.2 Global Dependencies

The analysis of global dependencies are inspired by the approach of Kemmerer [12]. This approach utilizes a transitive closure of the local dependencies. In our setting the local dependencies were identified by the control flow analysis presented above. However as we wish to take execution traces into account, we need to extend the closure, so that the edges are labelled with regular expressions. The goal of the closure will be to guarantee that the language of regular expressions connected to an edge does indeed accept all the executions traces in which the information flow happens. Therefore a correct closure must guarantee that the resulting labelled graph has an edge from a node n_0 to another node

$$
\begin{array}{lll}
(\hat{T},\hat{\sigma},\hat{C}) \models l :: P : \hat{G} & \text{iff} & (\hat{T},\hat{\sigma},\hat{C}) \models^{\lfloor l \rfloor} P : \hat{G} \\
(\hat{T},\hat{\sigma},\hat{C}) \models l :: \langle et \rangle : \hat{G} & \text{iff} & \langle et \rangle \in \hat{T}(\lfloor l \rfloor) \\
(\hat{T},\hat{\sigma},\hat{C}) \models (N_1 \parallel N_2) : \hat{G} & \text{iff} & (\hat{T},\hat{\sigma},\hat{C}) \models N_1 : \hat{G} \wedge \\
 & & (\hat{T},\hat{\sigma},\hat{C}) \models N_2 : \hat{G} \\
\hline
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} \mathbf{nil} : \hat{G} & \text{iff} & \mathit{true} \\
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} P_1 \mid P_2 : \hat{G} & \text{iff} & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P_1 : \hat{G} \wedge \\
 & & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P_2 : \hat{G} \\
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} A : \hat{G} & \text{iff} & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P : \hat{G} \wedge \\
 & & A \triangleq P \\
\hline
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} \mathbf{out}^{\iota}(t)@\ell.P : \hat{G} & \text{iff} & \forall l' \in \hat{\sigma}(\ell) : \hat{\sigma}[\![t]\!] \subseteq \hat{T}(l') \wedge \\
 & & [\hat{C}(\iota) \overset{l}{\mapsto} \hat{\sigma}(\ell)] \subseteq \hat{G} \wedge \\
 & & \forall \iota_P \in E(P) : \hat{C}(\iota) \subseteq \hat{C}(\iota_P) \wedge \\
 & & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P : \hat{G} \\
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} \mathbf{in}^{\iota}(T)@\ell.P : \hat{G} & \text{iff} & \forall l' \in \hat{\sigma}(\ell) : \hat{\sigma} \vdash_1 T : \hat{T}(l') \triangleright \hat{W}_{\bullet} \wedge \\
 & & [\hat{C}(\iota) \overset{l}{\mapsto} \hat{\sigma}(\ell)] \subseteq \hat{G} \wedge \\
 & & \forall \iota_P \in E(P) : \hat{C}(\iota) \cup \hat{\sigma}(\ell) \subseteq \hat{C}(\iota_P) \wedge \\
 & & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P : \hat{G} \\
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} \mathbf{read}^{\iota}(T)@\ell.P : \hat{G} & \text{iff} & \forall l' \in \hat{\sigma}(\ell) : \hat{\sigma} \vdash_1 T : \hat{T}(l') \triangleright \hat{W}_{\bullet} \wedge \\
 & & \forall \iota_P \in E(P) : \hat{C}(\iota) \cup \hat{\sigma}(\ell) \subseteq \hat{C}(\iota_P) \wedge \\
 & & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P : \hat{G} \\
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} \mathbf{eval}^{\iota}(Q)@\ell.P : \hat{G} & \text{iff} & \forall \iota_Q \in E(Q) : \hat{C}(\iota) \subseteq \hat{C}(\iota_Q) \wedge \\
 & & \forall l' \in \hat{\sigma}(\ell) : (\hat{T},\hat{\sigma},\hat{C}) \models^{l'} Q : \hat{G} \wedge \\
 & & \forall \iota_P \in E(P) : \hat{C}(\iota) \subseteq \hat{C}(\iota_P) \wedge \\
 & & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P : \hat{G} \\
(\hat{T},\hat{\sigma},\hat{C}) \models^{l} \mathbf{newloc}^{\iota}(u).P : \hat{G} & \text{iff} & \{\lfloor u \rfloor\} \subseteq \hat{\sigma}(\lfloor u \rfloor) \wedge \\
 & & \forall \iota_P \in E(P) : \hat{C}(\iota) \subseteq \hat{C}(\iota_P) \wedge \\
 & & (\hat{T},\hat{\sigma},\hat{C}) \models^{l} P : \hat{G}
\end{array}
$$

Fig. 9. Flow analysis

n_k, whenever there exists a path possibly through other nodes in the local flow graph. Furthermore the labels in the resulting graph must accept a language that is a superset of all the languages in the local flow graphs.

Definition 11. *A correct closure $\hat{\mathcal{H}}$ of the flow graph $\hat{G}$, written $\hat{G} \trianglelefteq \hat{\mathcal{H}}$, is defined as $\hat{G} \trianglelefteq \hat{\mathcal{H}}$ iff*

$$\forall n_0 \overset{l_1}{\mapsto} n_1 \overset{l_2}{\mapsto} \cdots \overset{l_k}{\mapsto} n_k \in \hat{G} : \exists \delta : l_1 \cdots l_k \in [\![\delta]\!] \wedge \; n_0 \overset{\delta}{\rightsquigarrow} n_k \; \in \hat{\mathcal{H}}$$

where $n_0 \overset{l_1}{\mapsto} n_1 \overset{l_2}{\mapsto} \cdots \overset{l_k}{\mapsto} n_k \in \hat{G}$ *means* $\forall i : n_{i-1} \overset{l_i}{\mapsto} n_i \in \hat{G}$.

One algorithm that can be used to compute the least $\hat{\mathcal{H}}$ such that $\hat{G} \trianglelefteq \hat{\mathcal{H}}$ is the *Pigeonhole Principle* presented in [11].

5.3 Static Security

We are confident that the static analyses presented above compute a flow graph $\hat{\mathcal{H}}$ for which the analyzed net comply with History-based Release.

Conjecture 1. *If* $(\hat{T}, \hat{\sigma}, \hat{C}) \models N : \hat{G}$ *and* $\hat{G} \trianglelefteq \hat{\mathcal{H}}$ *then* $N \ \approx_{\hat{\mathcal{H}}} \ N$*.*

In fact the analyses ensure that the analyzed net comply with History-based Release for any policy that is at least as restrictive as $\hat{\mathcal{H}}$.

Corollary 1. *If* $(\hat{T}, \hat{\sigma}, \hat{C}) \models N : \hat{G}$*,* $\hat{G} \trianglelefteq \hat{\mathcal{H}}$ *and* $\hat{\mathcal{H}} \leq G$ *then* $N \ \approx_G \ N$*.*
Proof: Follows from Lemma 2 and Conjecture 1. ∎

6 Related Work

Traditionally policies for information flow security have been of the form of security lattices [1, 6] where an underlying hierarchical structure on the principals in the system is assumed and reflected in the security lattice. Hence the principals are tied to security levels and an ordering of security levels indicate what information is observable to a principal. Security lattices have found a widespread usage in language-based approaches to information flow security, see e.g. [24, 20].

In this paper we base our security policies on labelled graphs, i.e. without assigning an underlying ordering. Due to the lack of underlying ordering the expressiveness of the policies is increased, allowing for simplified specification of security policies for systems. One example is systems that let a principal act on resources in different security domains without causing a flow of information in between. The expressiveness gained is due to the transitive nature of the ordering in a lattice. Graphs have previously been used as information flow policies in [23]. Furthermore these policies relate back to resource matrices applied for e.g. covert channel identification [12, 15].

Clearly the translation of a policy specified as a lattice to a labelled graph is straightforward. For each security level a security domain is introduced. Edges (labelled *true*) are added between security domains according to the ordering of the corresponding security levels.

The *Decentralized Label Model* by Myers and Liskov [17, 16] is a framework for security policies that allows owners of information to specify who is permitted to read that information. The basis model is still a lattice and does not provide expressiveness similar to what is presented in this paper.

6.1 Semantical security conditions

The goal of specifying whether a system complies with what is stated in an information flow policy has been formally stated as *non-interference* [5, 7]. Informally non-interference states that for all input to a system, that varies only on information not observable to the attacker, the resulting output will only vary on information not observable to the attacker. We showed in Section 4.2 that History-based Release generalises non-interference.

Non-Disclosure by Matos and Boudol [14] proposes extending the syntax of a ML-like calculus with specific constructs for loosening the security policy. These constructs have the form

$$\mathsf{flow}\ A \prec B\ \mathsf{in}\ M$$

where M is an expression and A and B are security levels. The construct extends the security relation to permit information to flow from A to B in M and thereby permits *disclosure* of confidential information in lexically scoped parts of programs. The policies presented in this paper allow for flows to be scoped within a specified location, i.e. locations tied with a security domain. Clearly by introducing a security domain tied to a *fresh* location for each flow construct and constraining the information flow to only happen in execution traces containing the security domain we get scoped flows. Finally due to the transitive nature of underlying lattice structure in [14], we need to perform a transitive closure on the resulting graph to achieve the same effect in our policies. In this manner the *Non-Disclosure* property can be seen as a specialisation of the History-based Release property. Obviously the *Non-Disclosure* property does not have the expressiveness to handle constraints on execution traces.

Intransitive non-interference. Goguen and Meseguer [7] generalised non-interference to *intransitive non-interference* while observing that information flows might be permitted when properly filtered at specified security levels. The property was further investigated in [9, 19] and adapted to a language-based setting by Mantel and Sands [13]. Mantel and Sands [13] formalise intransitive non-interference so that two goals are achieved. First the place in the program where information flow is limited through a syntactical extension of the language. Second the security level where information flows through is specified through an extension of the security lattice by an intransitive component.

History-based Release incorporates these concerns. The place in the program where information flow is guaranteed in the same way as described above for the non-disclosure property. Furthermore the locality-based security policies are intransitive due to being based on graphs rather than lattices.

Non-interference until by Chong and Myers [3, 4] propose adding conditions to security policies based on lattices. This is done by introducing a special annotated flow into the security policies of the form $\ell_0 \overset{c_1}{\rightsquigarrow} \cdots \overset{c_k}{\rightsquigarrow} \ell_k$ which states that information can be gradually downgraded along with the fulfilment of the conditions $c_1, \ldots, c_k$. It is straightforward to represent the downgrading condition with History-based Release.

However, observe that once the conditions are fulfilled, information can flow directly from ℓ_0 to any of the security levels $\ell_1, \ldots, \ell_k$. Therefore *non-interference until* does not provide the intransitive guarantees of History-based Release. Another point is the temporal constraints that History-based Release enforce on execution traces. *Non-interference until* provides simple temporal guarantees, namely that conditions are fulfilled prior to downgrading, however neither the order of the fulfilment nor the conditions allow for temporal constraints.

Flow Locks. Recently Broberg and Sands [2] introduced the novel concept of *flow locks* as a framework for dynamic information flow policies. The policies are specified in syntactical constructs introduced in an ML-like calculus. The constructs are utilised in the opening and closing of flow locks, these locks are used in constraining the information flows in the policies. The policies have the form $\{\sigma_1 \Rightarrow A_1; \ldots; \sigma_n \Rightarrow A_n\}$ and are annotated to declarations of variables in the programs. These policies correspond to ours, where a policy needs to specify where information might flow globally during execution and is hence not transitively closed. The major difference is that our policies can include temporal constraints which cannot be expressed in the flow locks policies.

Another major difference between [2] and the present paper is the intuition behind the security condition. In the flow lock setting information can flow to security levels as specified in the policies, as long as necessary locks are opened beforehand. This differs from our definition in not being tied to the actual flow of information. E.g. once a lock is open information can flow from several levels and several times. Furthermore flow locks have no way of observing if a lock has been closed and opened again between two flows. In our setting the constraints on the execution trace must be fulfilled for every single flow, hence it is not sufficient that another process is executed at the right location, just before or after considered flow.

7 Conclusion

We have presented a novel concept of locality-based security policies for information flow security. These policies are based on labelled graphs and we have illustrated how this allows for a simpler specification of certain policies. Furthermore we have presented the History-based Release condition that formalise how temporal conditions and intransitive information flow are captured in the security policies.

A static analysis is presented as a mechanism for verification of systems. The analysis is divided into three parts. Since the first part is the only syntax-directed part and as it is independent of the security policy for the given system it can freely be exchanged. Hence allowing us to analyse other process calculi or even programming languages. Future investigation might consider possibilities of adapting History-based Release to hardware description languages where locations could be mapped to blocks in structural specifications.

References

1. D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.
2. Niklas Broberg and David Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Proc. European Symposium on Programming*, pages 180–196, 2006.
3. Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 198–209, New York, NY, USA, 2004. ACM Press.

4. Stephen Chong and Andrew C. Myers. Language-based information erasure. In *CSFW*, pages 241–254, 2005.
5. E. S. Cohen. Information transmission in computational systems. *ACM SIGOPS Operating Systems Review*, 11(5):133–139, 1977.
6. D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.
7. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
8. Daniele Gorla and Rosario Pugliese. Resource access and mobility control with dynamic privileges acquisition. In *ICALP*, pages 119–132, 2003.
9. J. T. Haigh and W. D. Young. Extending the Non-Interference Version of MLS for SAT. In *IEEE Symposium on Security and Privacy*, pages 232–239. IEEE Computer Society Press, 1986.
10. R. R. Hansen, C. W. Probst, and F. Nielson. Sandboxing in myklaim. In *Proc. ARES'06*, 2006.
11. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 2nd edition*. Addison Wesley, 2001.
12. Richard A. Kemmerer. A practical approach to identifying storage and timing channels. In *IEEE Symposium on Security and Privacy*, pages 66–73, 1982.
13. Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *APLAS*, pages 129–145, 2004.
14. Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. In *Proc. IEEE Computer Security Foundations Workshop*, pages 226–240, 2005.
15. J. McHugh. Covert Channel Analysis. *Handbook for the Computer Security Certification of Trusted Systems*, 1995.
16. Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
17. Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, 1997.
18. Hanne Riis Nielson and Flemming Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In *The Essence of Computation*, pages 223–244, 2002.
19. J. Rushby. Noninterference, Transitivity, and Channel-Control Security Policies. Technical Report CSL-92-02, SRI International, December 1992.
20. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
21. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2005.
22. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *CSFW*, pages 200–214, 2000.
23. T. K. Tolstrup, F. Nielson, and H. Riis Nielson. Information Flow Analysis for VHDL. In *Proc. Eighth International Conference on Parallel Computing Technologies*, LNCS. Springer-Verlag, 2005.
24. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.
25. Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(1), 1999.