Adaptive Security Policies

Nielson, Flemming; Hansen, René Rydhof; Nielson, Hanne Riis

Published in:
Leveraging Applications of Formal Methods, Verification and Validation

DOI (link to publication from Publisher):
10.1007/978-3-030-61470-6_17

Publication date:
2020

Document Version
Accepted author manuscript, peer reviewed version

Citation for published version (APA):
Nielson, F., Hansen, R. R., & Nielson, H. R. (2020). Adaptive Security Policies. In T. Margaria, & B. Steffen (Eds.), *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Proceedings* (pp. 280-294). Springer. https://doi.org/10.1007/978-3-030-61470-6_17

Adaptive Security Policies

Flemming Nielson[1], René Rydhof Hansen[2], and Hanne Riis Nielson

[1] Department of Mathematics and Computer Science, Technical University of Denmark, Kgs. Lyngby, Denmark. fnie@dtu.dk
[2] Department of Computer Science, Aalborg University, Aalborg, Denmark. rrh@cs.aau.dk

Abstract. We develop an approach to security of adaptive agents that is based on respecting the local security policies of agents rather than imposing a global security policy on all agents. In short, an agent can be assured, that it will not be able to observe any violation of its own security policy due to the changing presence of other agents in its environment. The development is performed for a version of Dijkstra's Guarded Commands with relocation primitives, channel based communication, and explicit non-determinism. At the technical level a type system enforces local security policies whereas a reference monitor ensures that relocation is permissible with local security of all agents.

1 Introduction

In a traditional IT system, security is usually taken care of by a central security policy enforced on all components of the IT system. Security policies may range from simple discretionary access control policies over mandatory access control policies to decentralised control policies. In keeping with this approach, designs like the ones based on XACML [14] presuppose the existence of a central access control server for mandating whether or not access operations can be permitted throughout the possibly distributed IT system.

To fully support collective adaptive systems it seems overly demanding to enforce that they all adhere to the same security policy. Rather a framework needs to be found, where individual agents can define their own security policy and get some assurance that their own security policy is not compromised due to the changing presence of other agents in its environment. Examples might include visitors with mobile phones entering the premises of a corporation, customers changing their service providers, or a robot entering from one legislative domain to another.

This paper proposes a framework ensuring that agents will never be able to observe that information flows throughout the system in a manner that violates their own security policies. The basic idea being that an agent must accept its environment as long as it cannot observe any violation of its own security policy. We consider this to be a realistic proposal, because in real systems one would never have any guarantees against the internal behaviour of other agents on the system, including providers of social media channels and national intelligence services. As an example, if another agent receives confidential information and decides to make it public on a medium that can also be accessed from the agent in question, then this would constitute a violation of our framework. We consider it acceptable that we cannot detect if information is collected by other agents in a closed group, such as a company like Google or a national intelligence service.

The approach is highly motivated by considerations of *non-interference*; here it is ensured that there is no information flow from data at a higher security level to data at a lower security level. This is usually formulated for a deterministic system as the condition that two executions whose start states agree above a certain 'secret' security level also produce resulting states that agree above that 'secret' security level. (For non-deterministic systems the sets of possible outcomes need to be the same; for probabilistic systems the probability distributions on possible outcomes need to be the same.) This is a global condition focusing on terminating computations, so we will need to rephrase it as a local condition on what can be observed during a possibly non-terminating computation. We therefore aim at a situation where no agent in the collective adaptive system will be able to observe an information flow from data at a higher security level *in its own security lattice* to data at a lower security level *in its own security lattice.*

Motivated by [4] this paper develops the syntax and semantics of a language supporting this development. We then clarify our notion of security at the agent level and at the system level and finally we consider how to make this practical by precomputing the security checks. We conclude with directions for further work and a comparison with the literature.

2 Syntax

We extend Dijkstra's Guarded Commands language [6] with parallelism, communication and security domains and allow agents to dynamically modify their location in the environment. This could be change of physical location or merely change of logical location by changing its contact point in the environment (for example the internet provider); our syntax will be more suggestive of the latter.

The main syntactic category of *Adaptive Guarded Commands* is that of systems (denoted S). A system

$$\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1\ \square\ \cdots\ \square\ \boldsymbol{L}_n\, D_n\, C_n\ \texttt{rap}$$

consists of a number of parallel agents, each with their own security lattice ($\boldsymbol{L}_i$) for expressing their local security policy, their own declarations (D_i) of local variables and channels to the environment, and their own command (C_i) for controlling its behaviour. The fundamental idea is that the partial order ($\sqsubseteq_i$ or just $\sqsubseteq$) of the security lattice indicates the direction in which the security level of data may freely change; in approaches based on the Decentralised Label Model [11] this is called 'restriction'. The syntax is summarised in Figure 1 and explained below; for simplicity of presentation we shall assume that any two security lattices ($\boldsymbol{L}_i$ and $\boldsymbol{L}_j$) are either disjoint or equal ($\boldsymbol{L}_i \cap \boldsymbol{L}_j = \emptyset$ or $\boldsymbol{L}_i = \boldsymbol{L}_j$) so that we can use the union of lattices rather than the disjoint union of lattices.

$$
\begin{array}{lll}
\text{systems:} & S ::= \texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1\ \square\ \cdots\ \square\ \boldsymbol{L}_n\, D_n\, C_n\ \texttt{rap} & (n > 0) \\
\text{declarations:} & D ::= D; D \mid x : t\,\ell \mid c? : t\,\ell \mid c! : t\,\ell & \\
\text{commands:} & C ::= x := e \mid c\,!\,e \mid c\,?\,x \mid C_1\,;C_2 & \\
 & \quad \mid\ \texttt{if}\ e_1 \to C_1\ \square \ldots \square\ e_n \to C_n\ \texttt{fi} & (n > 0) \\
 & \quad \mid\ \texttt{do}\ e_1 \to C_1\ \square \ldots \square\ e_n \to C_n\ \texttt{od} & (n > 0) \\
 & \quad \mid\ \texttt{sum}\ C_1\ \square \ldots \square\ C_n\ \texttt{mus} & (n > 0) \\
 & \quad \mid\ \texttt{relocate}(l) & \\
\text{expressions:} & e ::= x \mid \texttt{loc} \mid n \mid e_1 + e_2 \mid \cdots & \\
 & \quad \mid\ \texttt{true} \mid e_1 = e_2 \mid \cdots \mid e_1 \wedge e_2 \mid \cdots & \\
\text{data types:} & t ::= \texttt{int} \mid \texttt{bool} \mid \texttt{data} \mid \cdots & \\
\text{security level:} & \ell\ \in\ \bigcup_i \boldsymbol{L}_i &
\end{array}
$$

Fig. 1. Syntax of Adaptive Guarded Commands.

Each agent will have a number of declarations. It may be a variable (x) of some type (t) and security level (ℓ). It may be a channel (c) used for input accepting data of some type and security level. It may be a channel (c) used to output data of some type and security level.

The commands include those of Dijkstra's Guarded Commands so we have the basic command of assignment ($x := e$) in addition to sequencing ($C_1\,;C_2$) and constructs for conditionals (`if` $e_1 \to C_1\ \square \ldots \square\ e_n \to C_n$ `fi`) and iteration (`do` $e_1 \to C_1\ \square \ldots \square\ e_n \to C_n$ `od`). On top of this we follow [13] and introduce basic commands for output ($c\,!\,e$) and input ($c\,?\,x$) over a channel (c) and a command performing an 'external' non-deterministic choice among commands (`sum` $C_1\ \square \ldots \square\ C_n$ `mus`); although it will typically be the case that each C_i in `sum` $C_1\ \square \ldots \square\ C_n$ `mus` takes the form $c\,!\,e\,;C$ or $c\,?\,x\,;C$ we shall not formally impose this. Finally, we have a relocation construct (`relocate`(l)) for allowing an agent to dynamically relocate to another contact point (l) in its environment.

The details of the expressions (e) are of little interest to us but they are likely to include variables (x), a special token indicating the current location (`loc`), numbers (n), arithmetic operations (e.g. $e_1 + e_2$), truth values (e.g. `true`), relational operations (e.g. $e_1 = e_2$), and logical operations (e.g. $e_1 \wedge e_2$).

We do not need to go deeply into the structure of datatypes (t) but assume that they contain integers (`int`) and booleans (`bool`) and a more interesting type of data (`data`). We shall leave the syntax of security levels (ℓ), channels (c), and locations (l) to the concrete examples.

3 Semantics

Expressions Expressions are evaluated with respect to a memory σ that assigns values to all variables of interest and following [13] the semantic judgement takes the form

$$\sigma \vdash e \rhd v$$

$$\frac{}{\sigma \vdash n \rhd n} \qquad \frac{}{\sigma \vdash \texttt{true} \rhd \mathsf{tt}} \qquad \frac{}{\sigma \vdash \texttt{loc} \rhd \sigma(\texttt{loc})} \qquad \frac{}{\sigma \vdash x \rhd \sigma(x)} \text{ if } \sigma(x) \text{ defined}$$

$$\frac{\sigma \vdash e_1 \rhd v_1 \quad \sigma \vdash e_2 \rhd v_2}{\sigma \vdash e_1 + e_2 \rhd v_1 + v_2} \qquad \frac{\sigma \vdash e_1 \rhd v_1 \quad \sigma \vdash e_2 \rhd v_2}{\sigma \vdash e_1 = e_2 \rhd v_1 = v_2} \qquad \frac{\sigma \vdash e_1 \rhd v_1 \quad \sigma \vdash e_2 \rhd v_2}{\sigma \vdash e_1 \wedge e_2 \rhd v_1 \wedge v_2}$$

Fig. 2. Semantics of expressions.

$$\frac{\sigma \vdash e \rhd v}{(x := e, \sigma) \rightarrow^{\tau} (\surd, \sigma[x \mapsto v])} \text{ if } \sigma(x) \text{ is defined}$$

$$\frac{\sigma \vdash e \rhd v}{(c\,!\,e, \sigma) \rightarrow^{c\,!\,v} (\surd, \sigma)} \qquad \frac{}{(c\,?\,x, \sigma) \rightarrow^{c\,?\,v} (\surd, \sigma[x \mapsto v])} \text{ if } \sigma(x) \text{ is defined}$$

$$\frac{(C_1, \sigma) \rightarrow^{\varphi} (C_1', \sigma')}{(C_1\,; C_2, \sigma) \rightarrow^{\varphi} (C_1'\,; C_2, \sigma')} \text{ if } C_1' \neq \surd \qquad \frac{(C_1, \sigma) \rightarrow^{\varphi} (\surd, \sigma')}{(C_1\,; C_2, \sigma) \rightarrow^{\varphi} (C_2, \sigma')}$$

$$\frac{\sigma \vdash e_i \rhd \mathsf{tt}}{(\texttt{if}\ e_1 \rightarrow C_1 \ \Box\ \cdots\ \Box\ e_n \rightarrow C_n\ \texttt{fi}, \sigma) \rightarrow^{\tau} (C_i, \sigma)}$$

$$\frac{\sigma \vdash e_i \rhd \mathsf{tt}}{(\texttt{do}\ \cdots\ \Box\ e_i \rightarrow C_i\ \Box\ \cdots\ \texttt{od}, \sigma) \rightarrow^{\tau} (C_i\,; \texttt{do}\ \cdots\ \Box\ e_i \rightarrow C_i\ \Box\ \cdots\ \texttt{od}, \sigma)}$$

$$\frac{\sigma \vdash e_1 \rhd \mathsf{ff} \quad \cdots \quad \sigma \vdash e_n \rhd \mathsf{ff}}{(\texttt{do}\ e_1 \rightarrow C_1\ \Box\ \cdots\ \Box\ e_n \rightarrow C_n\ \texttt{od}, \sigma) \rightarrow^{\tau} (\surd, \sigma)}$$

$$\frac{(C_i, \sigma) \rightarrow^{\varphi} (C_i', \sigma')}{(\texttt{sum}\ C_1\ \Box\ \cdots\ \Box\ C_n\ \texttt{mus}, \sigma) \rightarrow^{\varphi} (C_i', \sigma')}$$

$$\frac{}{(\texttt{relocate}(l), \sigma) \rightarrow^{\mathsf{go}\,l} (\surd, \sigma)}$$

Fig. 3. Semantics of commands.

and the details are provided by the axiom schemes and rules of Figure 2 and are mostly straightforward and uninteresting; note that the token `loc` is treated as a variable.

Commands Commands are interpreted relative to a local memory σ for each agent in question and may update it as needed. The semantic judgement takes the form

$$(C, \sigma) \rightarrow^{\varphi} (C', \sigma')$$

where the superscript (φ) indicates whether the action is silent (τ), an input $(c\,?\,v)$, an output $(c\,!\,v)$ or a relocation $(\mathsf{go}\,l)$.

The details are provided in Figure 3 and we use C and C' to range both over the commands of Figure 1 and the special symbol $\surd$ indicating a terminated

$$\frac{(C_i,\sigma_i) \to^{\tau} (C_i',\sigma_i')}{\begin{array}{l}(\texttt{par} \cdots \;\Box\; \boldsymbol{L}_i\, D_i\, C_i \;\Box\; \cdots\ \texttt{rap}, \cdots\sigma_i\cdots) \\ \quad \to (\texttt{par} \cdots \;\Box\; \boldsymbol{L}_i\, D_i\, C_i' \;\Box\; \cdots\ \texttt{rap}, \cdots\sigma_i'\cdots)\end{array}} \quad \text{if } \sigma_i \text{ covers } D_i$$

$$\frac{(C_i,\sigma_i) \to^{c\,!\,v} (C_i',\sigma_i') \qquad (C_j,\sigma_j) \to^{c\,?\,v} (C_j',\sigma_j')}{\begin{array}{l}(\texttt{par} \cdots \boldsymbol{L}_i\, D_i\, C_i \;\Box\; \boldsymbol{L}_j\, D_j\, C_j \cdots \texttt{rap}, \cdots\sigma_i\sigma_j\cdots) \\ \quad \to (\texttt{par} \cdots \boldsymbol{L}_i\, D_i\, C_i' \;\Box\; \boldsymbol{L}_j\, D_j\, C_j' \cdots \texttt{rap}, \cdots\sigma_i'\sigma_j'\cdots)\end{array}} \quad \text{if } \begin{cases}\sigma_i \text{ covers } D_i \\ \sigma_j \text{ covers } D_j \\ i \neq j \\ c? : t\,\ell \text{ in } D_i \\ \ \text{and } \ell \neq \bot_i \\ c! : t'\,\ell' \text{ in } D_j \\ \sigma_i(\texttt{loc}) = \sigma_j(\texttt{loc})\end{cases}$$

$$\frac{(C_i,\sigma_i) \to^{\texttt{go}\ l} (C_i',\sigma_i')}{\begin{array}{l}(\texttt{par} \cdots \;\Box\; \boldsymbol{L}_i\, D_i\, C_i \;\Box\; \cdots\ \texttt{rap}, \cdots\sigma_i\cdots) \\ \quad \to (\texttt{par} \cdots \;\Box\; \boldsymbol{L}_i\, D_i\, C_i' \;\Box\; \cdots\ \texttt{rap}, \cdots\sigma_i''\cdots)\end{array}} \quad \text{if } \begin{cases}\sigma_i \text{ covers } D_i \\ \cdots\end{cases}$$

$$\text{where } \sigma_i''(\texttt{loc}) = l \text{ and } \sigma_i''(x) = \begin{cases}\sigma_i'(x) & \text{if } x : t\,\ell \text{ in } D_i \text{ and } \ell = \bot \\ \texttt{0}_t & \text{if } x : t\,\ell \text{ in } D_i \text{ and } \ell \neq \bot\end{cases}$$

Fig. 4. Semantics of systems (with an incomplete rule for relocation).

configuration. Communication will be taken care of at system level by means of synchronous communication so the rules for output and input merely indicate the action taking place (as a superscript on the arrow). The same approach is taken for the relocation construct. The remaining constructs are in line with [13] and are generally straightforward.

Systems The agents of a system have disjoint local memories so they can only exchange values by communicating over the channels. More precisely this means that for each process we will have a local memory assigning values to the variables of interest and we shall be based on synchronous communication. The judgement takes the form

$$\begin{array}{rl} & (\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1 \;\Box\; \cdots \;\Box\; \boldsymbol{L}_n\, D_n\, C_n\ \texttt{rap}, \sigma_1\cdots\sigma_n) \\ \to & (\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1' \;\Box\; \cdots \;\Box\; \boldsymbol{L}_n\, D_n\, C_n'\ \texttt{rap}, \sigma_1'\cdots\sigma_n')\end{array}$$

where once more we allow C and C' to range both over commands and the special symbol $\surd$ indicating a terminated configuration.

The details are provided in Figure 4. The first rule takes care of a constituent agent performing a silent step and we use 'σ_i covers D_i' as a shorthand for the condition that the domain of σ_i contains all variables declared in D_i as well as the token `loc`.

The second rule takes care of synchronous communication between two distinct agents. We require that $i \neq j$ and the rule should *not* be read to suggest that $i+1=j$. On top of ensuring that channels are locally declared in a manner consistent with their use for output and input we also ensure that both agents are located at the same point in their environment. In case a declaration contains multiple declarations of the form $c? : t\,\ell$ we use '$c? : t\,\ell$ in D_i' as a shorthand for the condition that the rightmost occurrence of any declaration of $c? : t'\,\ell'$ is $c? : t\,\ell$; similar considerations apply to $c! : t\,\ell$ (and $x : t\,\ell$ below). The decision not to allow the reception of data at security level $\perp$ will be explained below.

The third rule takes care of relocation. Since the semantics of systems needs to ensure that our overall notion of security is maintained we cannot provide the full details of the rule before having developed our notion of security in the subsequent sections. However, already now we can record that the location information of the agent relocating is being updated. Also, that the agent relocating is only allowed to retain its data at the lowest security level; at higher security levels some constant element $\mathsf{0}_t$ of type t will replace any previous value.

Example 1. The condition in the rule for communication that data cannot be received at the lowest security level, and the condition in the rule for relocation that only data at the lowest security level can be retained, jointly prevent certain information flows due to relocation that are not captured by our security checks developed in the next sections.

To illustrate this point let us suppose we have locations `LOC1` and `LOC2` and a parallel system composed of processes *procA* (initially located at `LOC1` with security lattice $\boldsymbol{L}_1 \colon \mathsf{U} \sqsubseteq_1 \mathsf{C} \sqsubseteq_1 \mathsf{S}$), *procB* (initially located at `LOC1` with security lattice $\boldsymbol{L}_2 \colon \mathsf{L} \sqsubseteq_1 \mathsf{H}$) and *procC* (initially located at `LOC2` with security lattice $\boldsymbol{L}_3 \colon \mathsf{N} \sqsubseteq_1 \mathsf{P}$) defined as follows:

$$\begin{aligned}
procA = {} & \boldsymbol{L}_1\ \ (\mathtt{c}_1! : \mathtt{data}\,\mathsf{S};\ \mathtt{c}_3? : \mathtt{data}\,\mathsf{C};\ \mathtt{tmpS} : \mathtt{data}\,\mathsf{S};\ \mathtt{tmpC} : \mathtt{data}\,\mathsf{C})\\
& \mathtt{c}_1\ \ !\ \ \mathtt{tmpS};\ \mathtt{c}_3\ \ ?\ \ \mathtt{tmpC}\\[1em]
procB = {} & \boldsymbol{L}_2\ \ (\mathtt{c}_1? : \mathtt{data}\,\mathsf{H};\ \mathtt{c}_2! : \mathtt{data}\,\mathsf{H};\ \mathtt{tmpH} : \mathtt{data}\,\mathsf{H})\\
& \mathtt{c}_1\ \ ?\ \ \mathtt{tmpH};\ \mathtt{relocate(LOC2)};\ \mathtt{c}_2\ \ !\ \ \mathtt{tmpH}\\[1em]
procC = {} & \boldsymbol{L}_3\ \ (\mathtt{c}_2? : \mathtt{data}\,\mathsf{P};\ \mathtt{c}_3! : \mathtt{data}\,\mathsf{P};\ \mathtt{tmpP} : \mathtt{data}\,\mathsf{P})\\
& \mathtt{c}_2\ \ ?\ \ \mathtt{tmpP};\ \mathtt{relocate(LOC1)};\ \mathtt{c}_3\ \ !\ \ \mathtt{tmpP}
\end{aligned}$$

The system can then be defined as:

$$\mathtt{par}\ procA\ \square\ procB\ \square\ procC\ \mathtt{rap}$$

Here there would be an information flow from the security level S to the security level C if we would allow this program to execute without the two conditions in the rules for communication and relocation. $\square$

In all cases, note that if one of the processes terminates then the corresponding component in the configuration will contain $\surd$ and it will not be able to evolve further.

4 Agent-Level Security

We begin by developing an information flow type system for ensuring that each agent can be assured that it adheres to its own security policy. The development

$$\frac{}{\rho \vdash n : \texttt{int}\,\bot} \quad \frac{}{\rho \vdash \texttt{true} : \texttt{bool}\,\bot} \quad \frac{}{\rho \vdash \texttt{loc} : \texttt{data}\,\bot} \quad \frac{}{\rho \vdash x : t\,\ell} \text{ if } \rho(x) = (t, \ell)$$

$$\frac{\rho \vdash e_1 : \texttt{int}\,\ell_1 \quad \rho \vdash e_2 : \texttt{int}\,\ell_2}{\rho \vdash e_1 + e_2 : \texttt{int}\,(\ell_1 \sqcup \ell_2)} \quad \frac{\rho \vdash e_1 : t\,\ell_1 \quad \rho \vdash e_2 : t\,\ell_2}{\rho \vdash e_1 = e_2 : \texttt{data}\,(\ell_1 \sqcup \ell_2)} \quad \frac{\rho \vdash e_1 : \texttt{bool}\,\ell_1 \quad \rho \vdash e_2 : \texttt{bool}\,\ell_2}{\rho \vdash e_1 \wedge e_2 : \texttt{bool}\,(\ell_1 \sqcup \ell_2)}$$

Fig. 5. Types and security levels for expressions.

borrows from that of [13] and is inspired by traditional approaches such as those of [15, 16] but are extended to deal with parallelism and non-determinism.

Well-typed Expressions For expressions the judgement takes the form

$$\rho \vdash e : t\,\ell$$

where ℓ is intended to indicate the 'highest' security level of a variable used in the expression (but we need to be a bit more precise for security lattices that are not totally ordered).

The details are provided by the axiom schemes and rules of Figure 5 and will be explained below. The judgement makes use of a type environment ρ that assigns types and security levels to all variables; if the expression e occurs in some agent then it will become clear shortly that ρ is constructed from the declarations local to that agent and that the only security levels considered are the local ones. The overall idea is that $\rho \vdash e : t\,\ell$ should ensure that the type of the expression e is t and that the security level is $\ell = \bigsqcup_i \rho(x_i)_2$ where x_i ranges over all free variables of e and $\rho(x)_2 = \ell$ whenever $\rho(x) = (t, \ell)$. This is in line with the development in [15, 16].

Well-typed Commands For commands the typing judgement takes the form

$$\rho \vdash C : L$$

where $L = [\ell_1, \ell_2]$ is intended to be a pair of security levels: ℓ_1 is the 'lowest' security level of a variable assigned in the command and ℓ_2 is the 'highest' security level a variable assigned in the command (but we need to be a bit more precise for security lattices that are not totally ordered). This tells us all we will need about the set of security levels for variables assigned in the command, as it will allow us to demand that the set contains exactly one element by imposing $\ell_1 = \ell_2$, and that it contains at most one element by imposing $\ell_1 \sqsupseteq \ell_2$; we shall do the latter to prevent information flows due to non-determinism.

The typing judgement is defined by the axiom schemes and rules of Figure 6 to be explained shortly. We shall allow to write $\ell \sqsubseteq [\ell_1, \ell_2]$ for $\ell \sqsubseteq \ell_1$ and define

$$[\ell_1, \ell_2] \sqcap [\ell_1', \ell_2'] = [\ell_1 \sqcap \ell_1', \ell_2 \sqcup \ell_2']$$

$$\frac{\rho \vdash e : t\,\ell}{\rho \vdash x := e : [\ell', \ell']} \text{ if } \begin{cases} \rho(x) = (t, \ell') \\ \ell \sqsubseteq \ell' \end{cases} \qquad \frac{\rho \vdash C_1 : L_1 \quad \rho \vdash C_2 : L_2}{\rho \vdash C_1 \,;\, C_2 : L_1 \sqcap L_2}$$

$$\frac{\rho \vdash e : t\,\ell}{\rho \vdash c\,!\,e : [\ell', \ell']} \text{ if } \begin{cases} \rho(c!) = (t, \ell') \\ \ell \sqsubseteq \ell' \end{cases} \qquad \frac{}{\rho \vdash c\,?\,x : [\ell', \ell']} \text{ if } \begin{cases} \rho(c?) = (t, \ell) \\ \rho(x) = (t, \ell') \\ \ell \sqsubseteq \ell' \end{cases}$$

$$\frac{\bigwedge_i \rho \vdash e_i : \texttt{bool}\,\ell_i \quad \bigwedge_i \rho \vdash C_i : L_i}{\rho \vdash \texttt{if } e_1 \to C_1 \,\square\, \cdots \,\square\, e_n \to C_n \texttt{ fi} : L_1 \sqcap \cdots \sqcap L_n} \text{ if } \begin{cases} \bigwedge_i \ell_i \sqsubseteq L_i \\ \bigwedge_{(i,j) \in \texttt{cosat}} \ell_j \sqsubseteq L_i \\ \bigwedge_{(i,j) \in \texttt{cosat}} \texttt{uniq}(L_i) \end{cases}$$

$$\frac{\bigwedge_i \rho \vdash e_i : \texttt{bool}\,\ell_i \quad \bigwedge_i \rho \vdash C_i : L_i}{\rho \vdash \texttt{do } e_1 \to C_1 \,\square\, \cdots \,\square\, e_n \to C_n \texttt{ od} : L_1 \sqcap \cdots \sqcap L_n} \text{ if } \begin{cases} \bigwedge_i \ell_i \sqsubseteq L_i \\ \bigwedge_{(i,j) \in \texttt{cosat}} \ell_j \sqsubseteq L_i \\ \bigwedge_{(i,j) \in \texttt{cosat}} \texttt{uniq}(L_i) \end{cases}$$

$$\frac{\bigwedge_i \rho \vdash C_i : L_i}{\rho \vdash \texttt{sum } C_1 \,\square\, \cdots \,\square\, C_n \texttt{ mus} : L_1 \sqcap \cdots \sqcap L_n} \text{ if } \bigwedge_i \texttt{uniq}(L_i)$$

$$\frac{}{\rho \vdash \texttt{relocate}(l) : [\bot, \bot]}$$

Fig. 6. Types and pairs of security levels for commands.

(which is the greatest lower bound operation with respect to a partial order $\sqsubseteq'$ defined by $[\ell_1, \ell_2] \sqsubseteq' [\ell'_1, \ell'_2]$ whenever $\ell_1 \sqsubseteq \ell'_1$ and $\ell_2 \sqsupseteq \ell'_2$). We shall write $\texttt{uniq}([\ell_1, \ell_2])$ for the condition that $\ell_1 \sqsupseteq \ell_2$ and we shall write $L_{\mathsf{null}} = [\top, \bot]$. The intuition is, that if $\mathcal{L}$ is the set of security levels of variables modified in C then $L = [\bigsqcap \mathcal{L}, \bigsqcup \mathcal{L}]$; it follows that $\texttt{uniq}(L)$ holds whenever $\mathcal{L}$ contains at most one element, and $L = L_{\mathsf{null}}$ whenever $\mathcal{L}$ is empty (as would be the case for any `skip` statement we might add to the language). The first component of a security label $L = [\ell_1, \ell_2]$ is in line with the development in [15, 16] whereas the second component is responsible for dealing with non-determinism [13].

The rule for assignment records the security level of the variable modified and checks that the explicit information flow is admissible. The rule for sequencing is straightforward given our explanation of $\rho \vdash C : L$ and the operation $L_1 \sqcap L_2$. The rule for output and the axiom scheme for input are somewhat similar to the one for assignment, essentially treating output $c\,!\,e$ as an assignment $c := e$, and input $c\,?\,x$ as an assignment $x := c$. One might consider to adopt a more permissive type system by using L_{null} for output (rather than $[\ell', \ell']$) but this would open for some mild information flow due to communication.

The rule for 'external' non-deterministic choice takes care of correlation flows [12, 13]. It makes use of $\texttt{uniq}(L_i)$ to ensure that all modified variables (if any) have the same security level. The rules for conditional and iteration are essentially identical and make use of guards of the form $e_1 \to C_1 \,\square\, \cdots \,\square\, e_n \to C_n$. They take care of implicit flows by checking that $\ell_i \sqsubseteq L_i$ whenever $\bigwedge_i \rho \vdash e_i :$

$$\frac{\rho \vdash C : L}{\vdash \boldsymbol{L}\, D\, C : \checkmark} \text{ where } \begin{cases} \rho(x) = (t, \ell) \text{ whenever } x : t\,\ell \text{ in } D \\ \rho(c!) = (t, \ell) \text{ whenever } c! : t\,\ell \text{ in } D \\ \rho(c?) = (t, \ell) \text{ whenever } c? : t\,\ell \text{ in } D \\ D \text{ only mentions security levels in } \boldsymbol{L} \\ \text{all } c? : t\,\ell \text{ in } D \text{ have } \ell \neq \bot \end{cases}$$

$$\frac{\bigwedge_i \ \vdash \boldsymbol{L}_i\, D_i\, C_i : \checkmark}{\vdash \texttt{par } \boldsymbol{L}_1\, D_1\, C_1 \ \Box\ \cdots \ \Box\ \boldsymbol{L}_n\, D_n\, C_n \texttt{ rap} : \checkmark} \text{ where } \begin{cases} t_i = t_j \\ \text{whenever } c? : t_i\, \ell_i \text{ in } D_i \\ \text{whenever } c! : t_j\, \ell_j \text{ in } D_j \end{cases}$$

Fig. 7. Well-formedness of agents and systems.

$\texttt{bool}\,\ell_i$ and $\bigwedge_i \rho \vdash C_i : L_i$. They take care of bypassing flows [12, 13] whenever some $e_i \wedge e_j$ is satisfiable for $i \neq j$. This is expressed using the set `cosat` that contains those *distinct* pairs (i, j) of indices such that $e_i \wedge e_j$ is satisfiable; it may be computed using a Satisfaction Modulo Theories (SMT) solver such as Z3 [5] or it may be approximated using the DAG-based heuristics described in [12]. Whenever this is the case, the condition $\ell_j \sqsubseteq L_i$ checks that the bypassing flows are admissible, and the condition $\texttt{uniq}(L_i)$ checks the correlation flows are admissible.

In the rule for relocation one might consider to adopt a more permissive type system by using L_{null} for relocation (rather than $[\bot, \bot]$) but this would open for some mild information flow due to relocation.

Well-typed Agents To finish the considerations of security at the agent-level we may consider a judgement that takes the form

$$\vdash \boldsymbol{L}\, D\, C : \checkmark$$

and that is defined by the topmost rule in Figure 7. In addition to ensuring that the command is well-typed, it ensures that the variables, channels and security level occurring in a command are only the local ones, in line with the semantics not admitting any variables shared between agents, and it ensures that no information is received at the lowest security level as discussed previously.

5 System-Level Security

For systems we might extend the judgement $\vdash \cdots : \checkmark$ from agents to systems as suggested in the bottommost rule in Figure 7. In addition to ensuring that each agent is well-formed, it ensures that all information about a channel agree with respect to the type given to it. Clearly, there is no similar condition on the security levels because they are likely to come from different security domains. We shall only allow to use the semantics on well-typed systems S (i.e. satisfying $\vdash S : \checkmark$) which means that a few of the conditions in Figure 4 about the security levels of channels become superfluous.

$$\frac{\begin{array}{c}\ell' \sqsubseteq_i \ell'' \\ i \in \Sigma\end{array}}{\Sigma \vdash (i,\ell') \mapsto (i,\ell'')} \qquad \frac{\begin{array}{c}\Sigma \vdash (i',\ell') \mapsto (i,\ell) \\ \Sigma \vdash (i,\ell) \mapsto (i'',\ell'')\end{array}}{\Sigma \vdash (i',\ell') \mapsto (i'',\ell'')} \qquad \frac{\begin{array}{c}c? : t\,\ell'' \text{ in } D_{i''} \\ c! : t\,\ell' \text{ in } D_{i'} \\ i', i'' \in \Sigma\end{array}}{\Sigma \vdash (i',\ell') \mapsto (i'',\ell'')}$$

Fig. 8. System-Level Information Flow.

$$\frac{(C_i,\sigma_i) \to^{\mathtt{go}\ l} (C_i',\sigma_i')}{\begin{array}{l}(\mathtt{par}\ \cdots\ \square\ \boldsymbol{L}_i\, D_i\, C_i\ \square\ \cdots\ \mathtt{rap}, \cdots \sigma_i \cdots) \\ \quad \to (\mathtt{par}\ \cdots\ \square\ \boldsymbol{L}_i\, D_i\, C_i'\ \square\ \cdots\ \mathtt{rap}, \cdots \sigma_i'' \cdots)\end{array}} \quad \text{if} \begin{cases} \sigma_i \text{ covers } D_i \\ \forall j \in \Sigma : \\ \quad (\Sigma \vdash_j \ell' \mapsto \ell'') \Rightarrow \ell' \sqsubseteq_j \ell'' \\ \text{where} \\ \quad \Sigma = \{i\} \cup \{j \mid \sigma_j(\mathtt{loc}) = l\} \end{cases}$$

$$\text{where } \sigma_i''(\mathtt{loc}) = l \text{ and } \sigma_i''(x) = \begin{cases} \sigma_i'(x) & \text{if } x : t\,\ell \text{ in } D_i \text{ and } \ell = \bot \\ \mathtt{0}_t & \text{if } x : t\,\ell \text{ in } D_i \text{ and } \ell \neq \bot \end{cases}$$

Fig. 9. Semantics of relocation (dynamic version).

Our current setup creates the risk that the communications between an agent and its environment (i.e. the other agents) give rise to information flow that would not be admitted within the agent itself. Hence there is the risk that communication leads to local information flow not captured by the type system of the previous section.

We shall be interested in recording when the declaration of channels in the system may give rise to an information flow from some $\ell' \in \boldsymbol{L}_{i'}$ to some $\ell'' \in \boldsymbol{L}_{i''}$. We shall write this as

$$\Sigma \vdash (i',\ell') \mapsto (i'',\ell'')$$

where $\Sigma \subseteq \{1, \cdots, n\}$ records the agents at the location of interest including $i', i'' \in \Sigma$. The definition is given in Figure 8 where we write $\sqsubseteq_i$ for the partial order of $\boldsymbol{L}_i$.

The definition specialises to the case where $\ell' \in \boldsymbol{L}_i$ and $\ell'' \in \boldsymbol{L}_i$ for $i \in \Sigma$ and motivates defining

$$\Sigma \vdash_i \ell' \mapsto \ell'' \quad \text{iff} \quad \Sigma \vdash (i,\ell') \mapsto (i,\ell'') \wedge \ell' \in \boldsymbol{L}_i \wedge \ell'' \in \boldsymbol{L}_i$$

and we say that there is a *global information flow* from $\ell' \in \boldsymbol{L}_i$ to $\ell'' \in \boldsymbol{L}_i$ via Σ. We are now ready to define our notion of when a system S is secure but shall do so in two steps.

Definition 1. *A system* $\mathtt{par}\ \boldsymbol{L}_1\, D_1\, C_1\ \square\ \cdots\ \square\ \boldsymbol{L}_n\, D_n\, C_n\ \mathtt{rap}$ *is* secure *with respect to Σ whenever every global information flow from $\ell' \in \boldsymbol{L}_i$ to $\ell'' \in \boldsymbol{L}_i$ via Σ is consistent with $\boldsymbol{L}_i$: $\forall i \in \Sigma : \forall \ell', \ell'' \in \boldsymbol{L}_i : (\Sigma \vdash_i \ell' \mapsto \ell'') \Rightarrow \ell' \sqsubseteq_i \ell''$.*

To incorporate our notion of a system being secure into the well-formedness rules for systems would require us to fix the set Σ and hence limit the possibility of the system to adapt as agents are allowed to roam throughout the system.

Instead, we shall incorporate our notion of a system being secure into the semantics by allowing an agent to relocate only if security is not jeopardised. This gives rise to the completion of the semantics (of Figure 4) that is shown in Figure 9. Here the placement information Σ is computed from the local memories and we require security for all agents at the location to which the agent relocates (including the agent itself).

This motivates the following definition and proposition stating that the semantics of Figure 9 preserves security.

Definition 2. *A configuration* $(\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1\ [\,]\ \cdots\ [\,]\ \boldsymbol{L}_n\, D_n\, C_n\ \texttt{rap}, \sigma_1 \cdots \sigma_n)$ *is* secure *whenever* $\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1\ [\,]\ \cdots\ [\,]\ \boldsymbol{L}_n\, D_n\, C_n\ \texttt{rap}$ *is secure with respect to* $\Sigma_l = \{j \mid \sigma_j(\texttt{loc}) = l\}$ *for all locations* l*.*

Example 2. The system in Example 1 in an initial state located as stated in Example 1, yields a configuration that is secure. However, the dynamic semantics guards against any information flow from S to C due to relocation. □

Proposition 1. *Security is preserved under evaluation by the semantics of Figures 4 and 9.*

Proof. Suppose that $(\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1\ [\,]\ \cdots\ [\,]\ \boldsymbol{L}_n\, D_n\, C_n\ \texttt{rap}, \sigma_1 \cdots \sigma_n)$ is secure and that

$$\begin{array}{rl} & (\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1\ [\,]\ \cdots\ [\,]\ \boldsymbol{L}_n\, D_n\, C_n\ \texttt{rap}, \sigma_1 \cdots \sigma_n) \\ \to & (\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1'\ [\,]\ \cdots\ [\,]\ \boldsymbol{L}_n\, D_n\, C_n'\ \texttt{rap}, \sigma_1' \cdots \sigma_n') \end{array}$$

The resulting configuration $(\texttt{par}\ \boldsymbol{L}_1\, D_1\, C_1'\ [\,]\ \cdots\ [\,]\ \boldsymbol{L}_n\, D_n\, C_n'\ \texttt{rap}, \sigma_1' \cdots \sigma_n')$ is trivially secure if one of the first two rules in Figure 4 was used for the transition. In the case where the rule of Figure 9 is used for the transition we note that removing the i'th agent from the location $\sigma_i(\texttt{loc})$ does not jeopardise security, and adding the i'th agent to the location l does not jeopardise security either, because of the tests present in Figure 9.

Although our approach is motivated by the development of [4] (as mentioned in the Introduction) there are a substantial number of differences. In [4] programs are only allowed to be straight-line programs, so that there are no implicit flows into constructs that exchange data between different security domains, and the permissible flows are constrained to be determined by partial functions; we allow implicit, bypassing and correlation flows, we support the dynamic relocation of processes, and we do not require the permissible flows to be constrained by partial functions. With respect to security policies the approach of [4] applies the framework of Lagois connections [9] which forces them to impose additional technical[3] constraints on top of those needed in our development.

[3] In the terminology of [4] we impose constraints similar to their SC1(=LC1) and SC2(=LC2) but do not require any of their PC1, PC2, LC3, LC4, CC1, CC2 which are purely needed to stay within the framework of [9].

6 Precomputing Security Checks

It is possible to check for security in cubic time with respect to the size of the system S. To see this, first note that the number of security levels and channels considered is linear in the size of the system S. Next note that using Figure 8 to compute $\Sigma \vdash (i, \ell) \mapsto (i', \ell')$ amounts to computing the transitive closure of binary relations and that this can be done in cubic time.

Hence the application of the transition in Figure 9 can also be done in cubic time. We might expect to do better because we only perform the check for one location but it seems unfeasible to state a better worst-case complexity bound. This might make the approach unfeasible in practice in case n is very large.

To circumvent these problems we shall assume that while there might be many agents they will fall in a smaller number of groups sharing security lattices and channels. This seems a very realistic assumption for large collective adaptive systems. Define two agents indexed by i and j to be equivalent, written $i \sim j$, whenever

$$\begin{array}{l} \boldsymbol{L}_i = \boldsymbol{L}_j \\ c! : t\,\ell \text{ in } D_i \Leftrightarrow c! : t\,\ell \text{ in } D_j \\ c? : t\,\ell \text{ in } D_i \Leftrightarrow c? : t\,\ell \text{ in } D_j \end{array}$$

(for all choices of c, t, and ℓ). This requires the two agents to agree on the security lattice and their channels but not necessarily on their local variables.

This gives rise to equivalence classes $E_1, \cdots, E_N$ covering $\{1, \cdots, n\}$. For each equivalence class E_j we further choose a representative member $e_j \in E_j$ (say the least element of E_j). For an agent $i \in \{1, \cdots, n\}$ we next define $[i] \in \{1, \cdots, N\}$ to be the index of the equivalence class containing i, i.e. $i \in E_{[i]}$.

Lemma 1. *A system is secure with respect to Σ if and only if the system is secure with respect to $\{e_{[i]} \mid i \in \Sigma\}$.*

Proof. This follows from observing that we have $i \sim e_{[i]}$ and $i \sim j \Leftrightarrow [i] = [j]$ for all i, j.

While this result can be used to make the semantics more feasible (in case N is considerably smaller than n) we can go even further in obtaining a practical semantics. To do so we shall make use of a mapping

$$\Delta : \mathtt{Loc} \times \{1, \cdots, N\} \to \mathbb{N}$$

that for each location and (index of an) equivalence class gives the number of agents of that equivalence class that are currently at that location. We further define

$$\Delta \bullet l = \{e_k \mid \Delta(l, k) > 0\}$$

to be the set of representative members of agents present at the location l.

Restricting our attention from $\{1, \cdots, n\}$ to the representative members $\{e_1, \cdots, e_N\}$, and letting Σ range over subsets of the latter rather than the

$$\frac{(C_i,\sigma_i) \to^{\tau} (C_i',\sigma_i')}{\begin{array}{l}(\Delta, \texttt{par} \cdots \ \square\ \boldsymbol{L}_i\, D_i\, C_i\ \square\ \cdots \texttt{rap}, \cdots \sigma_i \cdots) \\ \quad \to (\Delta, \texttt{par} \cdots \ \square\ \boldsymbol{L}_i\, D_i\, C_i'\ \square\ \cdots \texttt{rap}, \cdots \sigma_i' \cdots)\end{array}} \quad \text{if } \sigma_i \text{ covers } D_i$$

$$\frac{(C_i,\sigma_i) \to^{c\,!\,v} (C_i',\sigma_i') \qquad (C_j,\sigma_j) \to^{c\,?\,v} (C_j',\sigma_j')}{\begin{array}{l}(\Delta, \texttt{par} \cdots \boldsymbol{L}_i\, D_i\, C_i\ \square\ \boldsymbol{L}_j\, D_j\, C_j \cdots \texttt{rap}, \cdots \sigma_i \sigma_j \cdots) \\ \quad \to (\Delta, \texttt{par} \cdots \boldsymbol{L}_i\, D_i\, C_i'\ \square\ \boldsymbol{L}_j\, D_j\, C_j' \cdots \texttt{rap}, \cdots \sigma_i' \sigma_j' \cdots)\end{array}} \quad \text{if } \begin{cases} \sigma_i \text{ covers } D_i \\ \sigma_j \text{ covers } D_j \\ i \neq j \\ c? : t\,\ell \text{ in } D_i \\ \ \text{and } \ell \neq \bot_i \\ c! : t'\,\ell' \text{ in } D_j \\ \sigma_i(\texttt{loc}) = \sigma_j(\texttt{loc}) \end{cases}$$

$$\frac{(C_i,\sigma_i) \to^{\texttt{go}\ l} (C_i',\sigma_i')}{\begin{array}{l}(\Delta, \texttt{par} \cdots \ \square\ \boldsymbol{L}_i\, D_i\, C_i\ \square\ \cdots \texttt{rap}, \cdots \sigma_i \cdots) \\ \quad \to (\Delta', \texttt{par} \cdots \ \square\ \boldsymbol{L}_i\, D_i\, C_i'\ \square\ \cdots \texttt{rap}, \cdots \sigma_i'' \cdots)\end{array}} \quad \text{if } \begin{cases} \sigma_i \text{ covers } D_i \\ \Delta' \bullet l \in \mathcal{S} \end{cases}$$

$$\text{where } \Delta' = \begin{cases} \Delta & \text{if } \sigma_i(\texttt{loc}) = l \\ \Delta \begin{bmatrix} (\sigma_i(\texttt{loc}), [i]) \mapsto \Delta(\sigma_i(\texttt{loc}), [i]) - 1 \\ (l, [i]) \mapsto \Delta(l, [i]) + 1 \end{bmatrix} & \text{if } \sigma_i(\texttt{loc}) \neq l \end{cases}$$

$$\sigma_i''(\texttt{loc}) = l \text{ and } \sigma_i''(x) = \begin{cases} \sigma_i'(x) & \text{if } x : t\,\ell \text{ in } D_i \text{ and } \ell = \bot \\ \mathtt{0}_t & \text{if } x : t\,\ell \text{ in } D_i \text{ and } \ell \neq \bot \end{cases}$$

Fig. 10. Semantics of systems (with precomputed security checks).

former, it makes sense to precompute the collection of Σ's, where the global information flow is consistent with the security lattices:

$$\mathcal{S} = \{\Sigma \subseteq \{e_1, \cdots, e_N\} \mid \forall j \in \Sigma : (\Sigma \vdash_j \ell' \mapsto \ell'') \Rightarrow \ell' \sqsubseteq_j \ell''\}$$

For small N it makes sense to represent this set as a list of bit-vectors of length N as there will be at most 2^N of these; if N is not small, symbolic datastructures can be used for checking $\Sigma \in \mathcal{S}$ efficiently.

This then motivates the semantics of Figure 10 that differs from our previous semantics in using the precomputed set $\mathcal{S}$ to check the permissibility of relocation more efficiently than before. The main idea is to extend a configuration with the mapping Δ and to devise a constant time operation for updating Δ in the case of relocation. This provides an essentially constant time semantics that is equivalent to the dynamic one.

Proposition 2. *The semantics of Figure 10 is equivalent to that of Figures 4 and 9:*

- *if* $(\Delta, S, \boldsymbol{\sigma}) \to (\Delta', S', \boldsymbol{\sigma}')$ *then* $(S, \boldsymbol{\sigma}) \to (S', \boldsymbol{\sigma}')$
- *if* $(S, \boldsymbol{\sigma}) \to (S', \boldsymbol{\sigma}')$ *then* $(\Delta^S_{\boldsymbol{\sigma}}, S, \boldsymbol{\sigma}) \to (\Delta^{S'}_{\boldsymbol{\sigma}'}, S', \boldsymbol{\sigma}')$

where $\Delta^S_{\boldsymbol{\sigma}}(l,k)$ *is the number of elements in* $\{i \mid \sigma_i(\texttt{loc}) = l \wedge [i] = k\}$.

7 Conclusion

We have adapted elements of non-interference to the setting of collective adaptive systems. The basic idea being that an agent must accept its environment as long as it cannot observe any violation of its own security policy. We consider this to be a realistic proposal, because in real systems one would never have any guarantees against the internal behaviour of other agents on the system, including providers of social media channels and national intelligence services. We have made an attempt at capturing the flows observable to an agent but do not fully guarantee against the exclusion from certain services; while exclusion from services is clearly observable, the reasons seldom are, and we do therefore not consider this a major drawback of our proposal.

Our notion of *relocation* requires agents to be sanitised before they relocate, i.e. our insistence that they can only retain information at the lowest security level. It would be interesting to consider a more flexible notion of *migration* where such sanitisation is not imposed. We believe this to be feasible by creating equivalence classes of the locations amongst which migration (as opposed to relocation) might take place; however, it is not clear that the precomputation semantics can be adapted to be semantically equivalent rather than just an approximation (where only the first condition in Proposition 2 would be ensured).

Enforcing security in a distributed system with data sharing and mobile code is a notoriously hard problem. In [7] the *myKlaim* calculus is proposed as a way to model and reason about *open* systems in which external, third-party code may be allowed inside a system to then be executed in a 'sandbox' environment to maintain security. If the mobile code can be proven to comply with the local security policy, through static analysis or certification, the code is also allowed to execute outside the sandbox. The security policies considered are access control policies rather than policies for secure information flow.

The *Fabric* framework, described in [1, 8], is an ambitious effort to develop a language and underlying system for designing and implementing distributed systems in a safe and secure manner. The system supports computational models based on both mobile code and data replication with strong security guarantees. Here the security policies are based on an extended version of the *decentralised label model* [10, 11]. This allows *principals*, essentially programs, to specify degrees of trust in other (remote) programs and thereby bound the potential security impact if that node should be compromised.

The main problems, insights, and solutions concerning the relationship between secure information flow and trust are distilled and further explored in the *Flow-Limited Authorization Model* [2] and the *Flow-Limited Authorization Calculus* [3] for reasoning about dynamic authorisation decisions.

Acknowledgement The first author was supported in part by the EU H2020-SU-ICT-03-2018 Project No. 830929 CyberSec4Europe (`cybersec4europe.eu`). The third author is currently on leave from the Department of Mathematics and Computer Science, Technical University of Denmark, Kgs. Lyngby, Denmark.

References

1. Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. Sharing mobile code securely with information flow control. In *Proceedings of the Symposium on Security and Privacy (SP 2012)*, pages 191–205, 2012.
2. Owen Arden, Jed Liu, and Andrew C. Myers. Flow-limited authorization. In *Proceedings of the 28th Computer Security Foundations Symposium (CSF 2015)*, pages 569–583, 2015.
3. Owen Arden and Andrew C. Myers. A calculus for flow-limited authorization. In *Proceedings of the 29th Computer Security Foundations Symposium (CSF 2016)*, pages 135–149, 2016.
4. Chandrika Bhardwaj and Sanjiva Prasad. Only connect, securely. In *Proceedings of Formal Techniques for Distributed Objects, Components, and Systems FORTE 2019, part of DisCoTec*, volume 11535 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2019.
5. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, part of ETAPS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
6. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
7. René Rydhof Hansen, Christian W. Probst, and Flemming Nielson. Sandboxing in myKlaim. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES 2006)*, pages 174–181, 2006.
8. Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4–5):367–426, 2017.
9. Austin Melton, Bernd S. W. Schröder, and George E. Strecker. Lagois connections - a counterpart to galois connections. *Theor. Comput. Sci.*, 136(1):79–107, 1994.
10. Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997)*, 1997.
11. Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.
12. Flemming Nielson and Hanne Riis Nielson. Lightweight Information Flow. In *Models, Languages and Tools for Concurrent and Distributed Programming, Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday*, volume 11665 of *Lecture Notes in Computer Science*, pages 455–470. Springer, 2019.
13. Flemming Nielson and Hanne Riis Nielson. Secure Guarded Commands. In *From Lambda-Calculus to Cybersecurity through Program Analysis, Essays Dedicated to Chris Hankin*, volume 12065 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2020.
14. Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson, and Flemming Nielson. The logic of XACML. *Sci. Comput. Program.*, 83:80–105, 2014.
15. Dennis M. Volpano and Cynthia E. Irvine. Secure flow typing. *Computers & Security*, 16(2):137–144, 1997.
16. Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.