Future-dependent Flow Policies with Prophetic Variables

Ximeng Li[*]
TU-Darmstadt
Darmstadt, Germany
li@mais.informatik.tu-darmstadt.de

Flemming Nielson
Technical University of Denmark
Lyngby, Denmark
fnie@dtu.dk

Hanne Riis Nielson
Technical University of Denmark
Lyngby, Denmark
hrni@dtu.dk

ABSTRACT

Content-dependency often plays an important role in the information flow security of real world IT systems. Content-dependency gives rise to informative policies and permissive static enforcement, and sometimes avoids the need for downgrading. We develop a static type system to soundly enforce future-dependent flow policies — policies that can depend on not only the current values of variables, but also their final values. The final values are referred to using what we call *prophetic variables*, just as the initial values can be referenced using logical variables in Hoare logic. We develop and enforce a notion of future-dependent security for open systems, in the spirit of "non-deducibility on strategies". We also illustrate our approach in scenarios where future-dependency has advantages over present-dependency and avoids mixtures of upgradings and downgradings.

Keywords

Information Flow Control; Future-Dependent Policies; Prophetic Variables; Security Type Systems

1. INTRODUCTION

The protection of confidential information from inadvertent leakage to unintended sinks is one of the central concerns in the area of information flow security (e.g., [18]). Such leakage can happen not only explicitly through data-flow chains, but also implicitly via control dependencies. To guard against subtle forms of leakage such as the implicit ones, it is usually desirable to guarantee the satisfaction of noninterference properties [7]. In this respect an important technique is security type systems [21] — they efficiently ascertain that a system is secure before ever launching it.

Oftentimes whether a variable in a program is "confidential" or "public" is contingent on the run-time values or statuses of certain parameters, which has led to extensive research in the last decade into content-dependent (or conditional, value-sensitive, etc.) information-flow control [3, 4, 14, 2, 1, 9, 15, 8, 12]. For example, in a database system, each row in a table on salary information is likely to concern a different employee, and it is sensible to allow disclosure of the salary field of each different row to the employee described by that row. Supposing we need to traverse the table and temporarily hold the current row in a structure, the classification of the salary field will then depend on the current value of the user ID field.

Interesting research results have been produced on the relationship between content-dependent policies and downgrading (or declassification) policies. It is shown in [3] that simple declassification operations, and "noninterference until declassification" [5], can be encoded in Flow Locks, where policies can vary with the current status of *lock variables*.

A somewhat different, but important concern is on the scenarios where content-dependent policies save the need for downgrading, giving rise to *noninterference* results. With *multiplexing*, messages from a number of sources need to go through a shared channel, before being directed in a demultiplexer to different destinations according to their address fields. In case the different sources have different security levels, the level of the shared channel needs to be lifted to the most restrictive one of those, for the flows into the shared channel to be deemed secure. However, this necessitates downgrading in demultiplexing, when a message coming from a low source needs to be directed to a low sink, via the high channel in the middle. It turns out to be natural [15, 8] to formulate a content-dependent policy for the shared channel: the security levels of the messages are dependent on their address fields. The need for downgrading is thereby avoided and noninterference guaranteed [8].

In a class of application scenarios, including one that we will discuss shortly, using policies that depend on current values can lead to a succession of upgradings and downgradings, due to a sequence of updates made to the variables that the policies depend on. However, all such updates are finally completed and the policies stabilized, and it becomes desirable to state the policies directly in terms of the final values of variables. For *open systems*, the flow of information can depend on future inputs from the computing environment, and the uncertainties of the environment also makes future-dependent flow policies preferable.

In this paper, we devise a security type system to enforce content-dependent flow policies that are allowed to depend on the *final values* of variables. We also develop a novel future-aware security property in the spirit of non-

[*]The research was completed at the Technical University of Denmark.

PLAS'16, October 24 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4574-3/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2993600.2993603

deducibility on strategies [22], for asynchronously communicating programs open to the environment, and show that our type system is sound with respect to the property.

As is the case for other developments on conditional flow policies (e.g., [4, 1, 8]), information on the statuses or values of variables is needed in the security enforcement. To this end, our type system incorporates a Hoare logic component. In Hoare logic, *logical variables* (or ghost variables) provide the ability to refer to the *initial values* of variables. In contrast, we need the capacity for relating to the *final values*, and introduce what we will call *prophetic variables*, for the predictive nature they possess. Unlike for logical variables, which are completely captured by the standard rules of Hoare logic, dedicated mechanisms are needed for the *backward preservation* of facts about prophetic variables, to bring what may happen in the future to the present.

Prophetic variables can be regarded as constants to the extent that logical variables can. This has a beneficial effect on the security enforcement in *concurrent programs*: the values of prophetic variables are not affected by concurrent updates; hence no additional interference analysis is needed to cater for potential policy changes triggered by such updates.

This paper is structured as follows. In Section 2, we introduce an example scenario that motivates future-dependent flow policies. In Section 3, we introduce the sequential fragment of our programming language — a While language extended with asynchronous communication. We then present future-dependent flow policies in Section 4, their type-based enforcement in Section 5, and the security guarantees in Section 6. In Section 7, we consider a further example where the uncertainties of the input environment add to the meaningfulness of prophetic variables. In Section 8, we demonstrate that the use of prophetic variables allows direct extension of the development to a concurrent language where processes communicate asynchronously. We discuss related work and conclude our development in Section 9.

2. A QUESTIONNAIRE EXAMPLE

Consider a voluntary online questionnaire to be answered among 100 people, such that the statistics generated from the answers is eventually revealed *only* to the participants. For this purpose it is natural to record whether each user has participated using some flag variables, in order not to repeatedly gather answers from the same user, and to be able to output the statistics only to the participants after the questionnaire is closed.

To express the dependency of the "right" to know the statistics on participation, a first thought may be to state the following policy about the answer submitted by user i:

> For each $j \in \{0, ..., 99\}$, if user j has participated *so far*, then it is given access to information about the answer.

However, such a policy leads to a succession of downgradings each happening with the participation of a new user, on the actual policy for the answer. Each such downgrading basically says that information about the answer needs to be revealed to one more participant, ahead of when this information is actually revealed (which happens in the end).

An alternative approach, however, is to relate directly to the final participation statuses:

> For each $j \in \{0, ..., 99\}$, if user j *eventually* participates, then it is given access to information about the answer.

For each potential eventual situation of participation, the policy above is concretized into a fixed one that is enforced throughout.

```
1   while $cont < N_{\max}$ do
2     $i := 0$;
3     while $i < 100$ do
4       if $par_i = 0$ then
5         if-recv($c_i$, $ans_i$) :
6           $par_i := 1$;
7           $res := aggr(res, i, ans_i)$
8         else skip
9         fi
10      else skip
11      fi;
12      $i := i + 1$
13    od;
14    $cont := cont + 1$
15  od;
16  $i := 0$;
17  while $i < 100$ do
18    if $par_i = 1$ then
19      send($c'_i$, $res$)
20    else skip
21    fi;
22    $i := i + 1$
23  od
```

Figure 1: The Code for the Questionnaire Server

A simple-minded questionnaire program is given in Figure 1. We assume that all variables are always implicitly initialized to 0. For a number $N_{\max}$ of times the program circles through the users 0-99, simulating a time period in which the questionnaire is open (lines 1-15). The flag variable recording whether user i has participated is par_i. When user i is visited, with no participation/answer currently recorded ($par_i = 0$), we input from the channel c_i to see if an answer is currently available. If so, the if-recv construct stores the answer in the variable ans_i, and proceed into its first branch, where the participation of user i is recorded by setting par_i to 1, and the answer ans_i is aggregated into the current statistics held in the variable res. After the questionnaire is closed, the final statistics is communicated to each user i such that par_i has the value 1, via the channel c'_i (lines 16-23). The use of asynchronous communication prevents the program from being blocked due to user inactivity.

Suppose user i is represented by the security principal USR_i. It is natural for the channel c'_i that returns the statistics to user i to have the simple policy $\{USR_i\}$, indicating that information about the content of this channel can only be revealed to USR_i.

In our policy language the channel c_i can be described by

$$(\underline{par_i} = 1 \;\Rightarrow \bigcurlywedge_{0 \le j < 100} (\underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\})) \curlywedge$$
$$(\underline{par_i} = 0 \;\Rightarrow\; \{USR_i\}) \tag{1}$$

In the above, $\mathbf{Pr}$ represents the set $\{USR_0, ..., USR_{99}\}$ of all security principals in this scenario. Each $\underline{par_i}$ is the *prophetic variable* for the program variable par_i, representing the value of par_i when the program finishes execution. The connectives $\Rightarrow$ and $\curlywedge$ can be understood as logical implication and conjunction, respectively. Hence the policy says

1. if the final value of par_i is 1, then the information about the answer submitted by user i cannot flow to

any principal USR_j ($0 \leq j < 100$) such that the final value of $\underline{par_j}$ is still 0, and

2. if the final value of par_i is 0, then the information about the answer submitted by user i can only flow to the principal USR_i.

Concerning point 2 above, note that even if $\underline{par_i}$ ends up with 0, user i may still have submitted an answer, which is not recorded by the program, due to the use of asynchronous communication.

Consider the situation where only the users 3, 20, 55 and 58 eventually participate in the questionnaire. We have $\underline{par_3} = \underline{par_{20}} = \underline{par_{55}} = \underline{par_{58}} = 1$, and $\underline{par_i} = 0$ for all $\underline{i \notin \{3, 20, 55, 58\}}$. The policy on the channel c_3 for the answer of user 3 will be $\{USR_3, USR_{20}, USR_{55}, USR_{58}\}$. Our type system will enforce this constant policy on all runs resulting in the same eventual situation concerning the participation of users. For example, variations in the content of the channel c_3, as introduced by the environment, are allowed to cause variations in the content of the channel c'_{20}, but not of c'_{99}.

It is not difficult to see that the policy (1) always directly gives the ultimate end-to-end leakage of the submitted answers, and no downgrading of actual policies happens.

3. LANGUAGE

We consider a simple While language extended with asynchronous communication. A program communicates with its environment as modeled by a set of strategies [22], each of which specifying a possible communication attempt of the environment after each execution trace. Later in Section 8, we deal with the scenario where the environment contains other programs, leading to open, concurrent systems. The assumption of an open system is suitable for demonstrating the benefits of using future-dependent flow policies in the face of uncertainties concerning the input data in the future. The use of asynchronous communication helps the programmer provide better guarantees of termination, in the face of uncertainties concerning whether the environment provides the data needed or not.

Syntax.

The syntax of our language is given in Figure 2.

$$\begin{array}{rl} a ::= & n \mid x \mid g(a_1, ..., a_m) \\ b ::= & \mathsf{true} \mid a_1 \ cop \ a_2 \mid b_1 \wedge b_2 \mid \neg\, b_1 \\ S ::= & \mathsf{skip} \mid x := a \mid \\ & \mathsf{send}(c, a) \mid \mathsf{recv}(c, x) \mid \mathsf{if\text{-}recv}(c, x) : S_1 \ \mathsf{else}\ S_2 \ \mathsf{fi} \mid \\ & S_1; S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \mid \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od} \end{array}$$

Figure 2: The Syntax

An arithmetic expression a can be a constant n, a variable $x \in \mathbf{Var}$, or the application of a function g on arithmetic expressions a_1, ..., a_n. A boolean expression b can be the constant true for truth, the comparison $a_1 \ cop \ a_2$ of two arithmetic expressions a_1 and a_2, a logical conjunction, or a logical negation. For a statement S, apart from the standard While language constructs, we admit three kinds of communications: the statement $\mathsf{send}(c, a)$ outputs the evaluation result of a to the channel $c \in \mathbf{Ch}$, the statement $\mathsf{recv}(c, x)$ inputs the oldest element of the buffered contents for channel c into x, blocking in case the buffer is empty, and the statement $\mathsf{if\text{-}recv}(c, x) : S_1$ else S_2 fi inputs the oldest element in the buffer of c and executes S_1 in case the buffer of c is non-empty, and executes S_2 without performing any actual input in case the buffer is empty.

Semantics.

We define the semantics of our language in a two-level manner. The first level is environment-agnostic, and is concerned with the judgment $\langle S, m\rangle \to \langle S', m'\rangle$. This represents a step of execution of the statement S in the memory m, resulting in the statement S' and the memory m'. Memories $m, m' \in \mathbf{Mem}$ capture the states for both variables and channels. In more detail, $m(x)$ is the value of variable x, and $m(c)$ is a FIFO queue holding the contents buffered for channel c. The semantic rules of the first level are given in Figure 3, where for an arithmetic expression a, $[\![a]\!] : \mathbf{Mem} \to \mathbf{Val}$ specifies the evaluation of a in each memory $m \in \mathbf{Mem}$ into a value, and for a boolean expression b, $[\![b]\!] : \mathbf{Mem} \to \{tt, \mathit{ff}\}$ specifies the evaluation of b in each memory $m \in \mathbf{Mem}$ into a truth value. We leave the details of their definitions unspecified.

$$\langle \mathsf{skip}, m\rangle \to \langle \mathsf{skip}, m\rangle$$

$$\langle x := a, m\rangle \to \langle \mathsf{skip}, m[x \mapsto [\![a]\!]m]\rangle$$

$$\frac{\langle S_1, m\rangle \to \langle S'_1, m'\rangle}{\langle S_1; S_2, m\rangle \to \langle S'_1; S_2, m'\rangle} \text{ if } S'_1 \neq \mathsf{skip}$$

$$\frac{\langle S_1, m\rangle \to \langle \mathsf{skip}, m'\rangle}{\langle S_1; S_2, m\rangle \to \langle S_2, m'\rangle}$$

$$\langle \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, m\rangle \to \langle S_1, m\rangle \quad \text{if } [\![b]\!]m = tt$$

$$\langle \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, m\rangle \to \langle S_2, m\rangle \quad \text{if } [\![b]\!]m = \mathit{ff}$$

$$\langle \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}, m\rangle \to \langle S; \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}, m\rangle \quad \text{if } [\![b]\!]m = tt$$

$$\langle \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}, m\rangle \to \langle \mathsf{skip}, m\rangle \quad \text{if } [\![b]\!]m = \mathit{ff}$$

$$\langle \mathsf{send}(c, a), m\rangle \to \langle \mathsf{skip}, m[c \mapsto v \cdot m(c)]\rangle \quad \text{if } [\![a]\!]m = v$$

$$\langle \mathsf{recv}(c, x), m\rangle \to \langle \mathsf{skip}, m[c \mapsto \gamma][x \mapsto v]\rangle \quad \text{if } m(c) = \gamma \cdot v$$

$$\langle \mathsf{recv}(c, x), m\rangle \to \langle \mathsf{recv}(c, x), m\rangle \quad \text{if } m(c) = \epsilon$$

$$\langle \mathsf{if\text{-}recv}(c, x) : S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, m\rangle \to \langle S_1, m[c \mapsto \gamma][x \mapsto v]\rangle \\ \text{if } m(c) = \gamma \cdot v$$

$$\langle \mathsf{if\text{-}recv}(c, x) : S_1\ \mathsf{else}\ S_2\ \mathsf{fi}, m\rangle \to \langle S_2, m\rangle \quad \text{if } m(c) = \epsilon$$

Figure 3: The Semantics of Statements

We elaborate on the execution of the communication statements as described in Figure 3. The statement $\mathsf{send}(c, a)$ simply performs an enque operation of the evaluation result of a into the original queuing buffer $m(c)$ of the channel c, resulting in the augmented buffer $v \cdot m(c)$ for c. The statement $\mathsf{recv}(c, x)$ updates x with v only if the queuing buffer of the channel c is non-empty, and of the form $\gamma \cdot v$, where v is the oldest element; otherwise the input is blocked and an empty step is produced. On the other hand, the statement $\mathsf{if\text{-}recv}(c, x) : S_1$ else S_2 fi branches to either S_1 or S_2 depending on whether the buffer of the channel c is empty. In case it is not, the updates to the memory are like those for $\mathsf{recv}(c, x)$; otherwise no update to the memory happens.

In Figure 3, the rule for skip, assignment, and communications produces the statement skip in the results of their one-step executions. The effect is that termination becomes unobservable; hence our security type system no longer needs to deal with the orthogonal issue of external timing channels.

The second level of our semantics is environment-aware. The environment can affect the communication channels just like the program can, depending on the observed execution *history* captured by memory traces $t \in \mathbf{Tr}$. A trace t is simply a sequence $m_1.m_2.....m_k$ of memories resulting from execution. We write $t(j)$ for the j-th memory in t, and start the indices at 1, so that the last memory is $t(|t|)$ where $|t|$ is the length of t. For simplicity we assume that the first memory in any trace is always a specific one that maps all variables to the value 0, and all channels to the empty buffer ϵ. The environment is then modeled abstractly as a set

$$ES \;\subseteq\; \mathbf{Tr} \to \{\mathsf{send}(c, v), \mathsf{recv}(c, [\,]), \diamond \mid c \in \mathbf{Ch}, v \in \mathbf{Val}\}$$

of *strategies*. The membership of a strategy ξ in ES represents the possibility of the environment communicating in the manner specified by ξ. In more detail, $\xi(t)$ is the communication potentially attempted by the environment after the trace t. The action $\mathsf{send}(c, v)$ represents an attempt to output the value v to the channel c, the action $\mathsf{recv}(c, [\,])$ represents an attempt to input a value from the channel c, and the action $\diamond$ represents that nothing is attempted.

$$\frac{\langle S, t(|t|)\rangle \to \langle S', m'\rangle}{\xi \vdash \langle S, t\rangle \to \langle S', t.m'\rangle}$$

$$\frac{\xi(t) = \mathsf{send}(c, v)}{\xi \vdash \langle S, t\rangle \to \langle S, t.(t(|t|)[c \mapsto v \cdot t(|t|)(c)])\rangle}$$

$$\frac{\xi(t) = \mathsf{recv}(c, [\,]) \quad t(|t|)(c) = \gamma \cdot v}{\xi \vdash \langle S, t\rangle \to \langle S, t.(t(|t|)[c \mapsto \gamma])\rangle}$$

$$\frac{\xi(t) = \mathsf{recv}(c, [\,]) \wedge t(|t|)(c) = \epsilon \;\vee\; \xi(t) = \diamond}{\xi \vdash \langle S, t\rangle \to \langle S, t.t(|t|)\rangle}$$

Figure 4: The Environment-aware Semantics

The environment-aware semantics establishes the judgment $\xi \vdash \langle S, t\rangle \to \langle S', t'\rangle$. This represents either an execution step of the statement S, or an action of the environment according to the strategy ξ. The former case is captured by the first rule of Figure 4, saying that given trace t, if in the environment-agnostic semantics S can execute for one step from the last memory $t(|t|)$ of t, updating the memory to m' and becoming the statement S', then in the environment-aware semantics S can execute with history t, updating it to $t.m'$ and also becoming S'. The latter case is captured by the remaining rules in the same figure.

4. FUTURE-DEPENDENT FLOW POLICIES

Our policy language allows the actual confidentiality policy of channels and variables to depend on the current, as well as the final values of variables. The dependency on final values is achieved using prophetic variables $\mathbf{PphVar} \subseteq \{\underline{x} \mid x \in \mathbf{Var}\}$. In this section we introduce the syntax and semantics of policies and their static comparison. We then elaborate on channel policies, and formalize policy environments to be used for the type system.

The syntax of the policy language is given below.

$$\begin{aligned}
P &::= \{p_1, ..., p_m\} \mid \phi \Rightarrow P \mid P_1 \curlywedge P_2\\
\phi &::= \mathsf{true} \mid \hat{a}_1 \; cop \; \hat{a}_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \forall x.\phi\\
\hat{a} &::= n \mid x \mid \underline{x} \mid g(\hat{a}_1, ..., \hat{a}_n)
\end{aligned}$$

Given a set $\mathbf{Pr}$ of security principals, a policy P can be a set $\{p_1, ..., p_m\} \subseteq \mathbf{Pr}$ of principals to whom the disclosure of information is restricted, an implication $\phi \Rightarrow P$ that imposes the restrictions of P only when ϕ is satisfied, or a conjunction $P_1 \curlywedge P_2$ that jointly imposes the restrictions of P_1 and P_2. This latter case can be understood in a way similar to how the reader policies of different owners are combined in the Decentralized Label Model [13].

In policies of the form $\phi \Rightarrow P$, the formula ϕ is allowed to be a comparison of two *extended arithmetic expressions* $\hat{a}_1$ and $\hat{a}_2$, that can contain prophetic variables. An example is the term $\underline{par_j} = 0$ in the policy (1) of Section 2. The syntax of ϕ also covers the case $\forall x.\phi$, and the reason is related to our type system: for a formula ϕ to hold after an input into x, $\forall x.\phi$ needs to hold before it, and the universal quantifier will be incorporated into the policies by our typing rules.

To precisely understand the security implications of these syntactical policies, we need to be concerned with the concrete reader sets permitted by them, given each particular valuation of program variables and prophetic variables. Hereafter, we will use $\varsigma \in \mathbf{St}_{\mathrm{pph}}$ to represent a mapping capturing the valuation of prophetic variables, and use $\uplus$ to represent the combination of two mappings with disjoint domains. Given the intuitive meanings of the syntactical policies, it is not difficult to understand the following way of interpreting them.

$$\begin{aligned}
[\![\{p_1, ..., p_m\}]\!](m \uplus \varsigma) &= \{p_1, ..., p_m\}\\
[\![\phi \Rightarrow P]\!](m \uplus \varsigma) &= \begin{cases} [\![P]\!](m \uplus \varsigma) & \text{if } m \uplus \varsigma \models \phi\\ \mathbf{Pr} & \text{otherwise}\end{cases}\\
[\![P_1 \curlywedge P_2]\!](m \uplus \varsigma) &= [\![P_1]\!](m \uplus \varsigma) \cap [\![P_2]\!](m \uplus \varsigma)
\end{aligned}$$

In the definition above, we have tacitly assumed that the domain of $m \uplus \varsigma$ covers all the free variables/prophetic variables of the policies interpreted, and left $m \uplus \varsigma \models \phi$ unspecified. This latter definition is straightforward. Note that for prophetic variables $\underline{x}$ we have $[\![\underline{x}]\!](m \uplus \varsigma) = \varsigma(\underline{x})$.

EXAMPLE 1. *Denoting by P_{20} the policy (1) of Section 2 for the channel c_{20}, we have*

$$[\![P_{20}]\!](m \uplus \varsigma_{3,20,55,58}) = \{USR_3, USR_{20}, USR_{55}, USR_{58}\},$$

where m is an arbitrary memory, $\varsigma_{3,20,55,58} = \varsigma_0[\underline{par_3} \mapsto 1][\underline{par_{20}} \mapsto 1][\underline{par_{55}} \mapsto 1][\underline{par_{58}} \mapsto 1]$, and $\varsigma_0 = \biguplus_{0 \leq j < 100} [\underline{par_j} \mapsto 0]$ maps all the prophetic variables of the participation flags to 0.

We can now easily decide the relative strength of policies based on the readers they allow: P_1 is more confidential than P_2 if P_1 allows less readers than P_2 does. However, to statically decide whether a policy is more confidential than another in all situations, a *syntactical condition* is needed.

To arrive at a concise syntactical ordering between policies, we consider a subset of syntactical policies in the *Implication Normal Form* (INF). These are policies of the form

$$(\phi_1 \Rightarrow L_1) \curlywedge (\phi_2 \Rightarrow L_2) \curlywedge ... \curlywedge (\phi_n \Rightarrow L_n),$$

where L_1, ..., L_n are sets of principals. For INF policies, we then define a pre-order $\preceq_{\mathrm{INF}}$:

$$\bigcurlywedge_i (\phi_i \Rightarrow L_i) \preceq_{\mathrm{INF}} \bigcurlywedge_j (\phi'_j \Rightarrow L'_j) \;\triangleq\; \forall i : \forall p \notin L_i : \exists j : \phi_i \Rightarrow \phi'_j \wedge p \notin L'_j.$$

Informally, a first INF policy is less than or equal to a second one if for each conjunct of the first policy, and each principal p not in the reader set of that conjunct, there exists a conjunct in the second policy, such that the condition of the former conjunct implies the condition of the latter one, and p is not in the reader set of the latter conjunct either. The ordering $\preceq_{\text{INF}}$ is sound in the following sense, where (and hereafter) $\mathbf{D}_{mp}$ represents the domain of the mapping mp, and $fv(P)$ is the set of free variables (*including prophetic variables*) of the policy P.

LEMMA 1. *Suppose P_1 and P_2 are two policies in the implication normal form. If $P_1 \preceq_{\text{INF}} P_2$, then we have*

$$\forall m, \varsigma \text{ s.t. } fv(P_1) \cup fv(P_2) \subseteq \mathbf{D}_{m \uplus \varsigma} : [\![P_1]\!](m \uplus \varsigma) \supseteq [\![P_2]\!](m \uplus \varsigma).$$

This order is, however, incomplete. Consider a scenario where p_1 and p_2 are the only security principals ($\mathbf{Pr} = \{p_1, p_2\}$), and the two policies to be compared are $\mathsf{false} \Rightarrow \{p_1\}$ and $\mathsf{false} \Rightarrow \{p_1, p_2\}$, where false is a syntactical sugar for $\neg\mathsf{true}$. Given arbitrary m and ς, the interpretation of both policies is always $\{p_1, p_2\}$, and $\{p_1, p_2\} \subseteq \{p_1, p_2\}$. However, we have $\mathsf{false} \Rightarrow \{p_1\} \not\preceq_{\text{INF}} \mathsf{false} \Rightarrow \{p_1, p_2\}$. The incompleteness of $\preceq_{\text{INF}}$ is non-troublesome since it is used in our security type system, and information flow type systems are usually far from being complete themselves.

To statically compare two arbitrary policies we introduce a *congruence relation* $\equiv$, which structurally identifies policies that are semantically the same, and can be used to turn a given policy into the implication normal form. The relation $\equiv$ is defined as the smallest congruence relation satisfying the rules in Figure 5, where $\star$ is the policy $\mathsf{true} \Rightarrow \mathbf{Pr}$.

$$P \curlywedge P \equiv P \qquad P_1 \curlywedge P_2 \equiv P_2 \curlywedge P_1$$
$$P_1 \curlywedge (P_2 \curlywedge P_3) \equiv (P_1 \curlywedge P_2) \curlywedge P_3$$
$$\star \curlywedge P \equiv P \qquad (\mathsf{true} \Rightarrow \emptyset) \curlywedge P \equiv (\mathsf{true} \Rightarrow \emptyset)$$
$$\mathsf{true} \Rightarrow P \;\equiv\; P \qquad \mathsf{false} \Rightarrow P \;\equiv\; \star \qquad \phi \Rightarrow \star \;\equiv\; \star$$
$$\phi_1 \Rightarrow (\phi_2 \Rightarrow P) \;\equiv\; (\phi_1 \wedge \phi_2) \Rightarrow P$$
$$\phi \Rightarrow (P_1 \curlywedge P_2) \;\equiv\; (\phi \Rightarrow P_1) \curlywedge (\phi \Rightarrow P_2)$$

Figure 5: The Structural Equivalence of Policies

EXAMPLE 2. *Still denote by P_{20} the policy (1) of Section 2. This policy is not in the implication normal form. However, we have*

$$P_{20} \;\equiv\; \bigcurlywedge_{0 \le j < 100} ((\underline{par_{20}} = 1 \wedge \underline{par_j} = 0) \Rightarrow \mathbf{Pr} \setminus \{USR_j\}) \;\curlywedge$$
$$(\underline{par_{20}} = 0 \Rightarrow \{USR_{20}\})$$

In the above, the policy to the right of $\equiv$ is already in the implication normal form. More importantly, it is semantically the same as P_{20}, which is guaranteed by the following general result.

LEMMA 2. *Suppose P_1 and P_2 are two policies. If $P_1 \equiv P_2$, then we have*

$$\forall m, \varsigma \text{ s.t. } fv(P_1) \cup fv(P_2) \subseteq \mathbf{D}_{m \uplus \varsigma} : [\![P_1]\!](m \uplus \varsigma) = [\![P_2]\!](m \uplus \varsigma).$$

We are in a position to introduce a syntactical ordering $\preceq$ applicable on all policies, by building on $\preceq_{\text{INF}}$ and $\equiv$:

$$P_1 \preceq P_2 \;\triangleq\; \exists P_1', P_2' \text{ in the implication normal form :}$$
$$P_1 \equiv P_1' \wedge P_2 \equiv P_2' \wedge P_1' \preceq_{\text{INF}} P_2'.$$

From Lemma 1 and Lemma 2, the soundness of $\preceq$ follows.

LEMMA 3. *Suppose P_1 and P_2 are two policies. If $P_1 \preceq P_2$, then we have*

$$\forall m, \varsigma \text{ s.t. } fv(P_1) \cup fv(P_2) \subseteq \mathbf{D}_{m \uplus \varsigma} : [\![P_1]\!](m \uplus \varsigma) \supseteq [\![P_2]\!](m \uplus \varsigma).$$

Presence Policies and Content Policies.

The bulk of existing research on conditional information flow policies does not address communication (known exceptions are [15] and [8]), which is a ubiquitous means of information exchange in real-world IT systems. We deal with communication, and, in doing so, distinguish between *presence policies* and *content policies*. For a channel c, the presence policy of c governs the confidentiality of whether any message is present in the buffer of c (or of the emptiness of the buffer of c), and the content policy of c governs the confidentiality of the content of individual messages in this buffer. Initiated in [17], this is a fine-grained treatment allowing one to separate the leakage incurred by the blockage of inputs from the leakage of communication content. We assume as in [17] that for each channel, its presence policy is no more confidential than its content policy — observing the content of a message implies knowing its presence.

Policy Environments.

To formulate the type system and the security property, we introduce policy environments $\mathcal{P} = (\mathcal{P}^\circ, \mathcal{P}^\bullet)$. Here $\mathcal{P}^\circ$ assigns a presence policy $\mathcal{P}^\circ(c)$ to each channel c, and $\mathcal{P}^\bullet$ assigns a content policy $\mathcal{P}^\bullet(c)$ to each channel c and a content policy $\mathcal{P}^\bullet(x)$ to each program variable $x \in \mathbf{Var}$. Prophetic variables do not have policies of their own, although they can be used in policies of channels and program variables.

When the policy (1) of Section 2 was introduced, it was concerned with the protection of the answers submitted by the users for the questionnaire, rather than whether the users do give answers. Hence with the policy environments just introduced, this policy can be referred to as $\mathcal{P}^\bullet(c_i)$ — it is the content policy of c_i.

Channel Policies Depend only on Prophetic Variables.

For a channel c, the dependency of its content policy $\mathcal{P}^\bullet(c)$ on program variables can be difficult to use. This is because $\mathcal{P}^\bullet(c)$ homogeneously governs all the messages buffered for c. The values of the program variables depended upon are likely to change in undesirable ways while a message is moving from one end of the queuing buffer to the other. In addition, the correspondence between the values of these variables and the position of each individual message in the buffer is also likely to be unclear. On the other hand, prophetic variables can be regarded as constants as logical variables can, resolving the problem caused by variability. Therefore a reasonable condition to impose is $fv(\mathcal{P}^\bullet(c)) \subseteq \mathbf{PphVar}$, i.e., channel policies can only depend on prophetic variables (and constants). We also require $fv(\mathcal{P}^\circ(c)) \subseteq \mathbf{PphVar}$, which is not a serious limitation because of the constraint already imposed between presence policies and content policies.

5. TYPE-BASED POLICY ENFORCEMENT

Ever since it is pioneered in [21], the type-based approach to the static enforcement of information flow policies has been developed extensively. Conditional flow policies increase the *precision* of enforcement, which fixed security types lack. In this section, we develop an information flow type system for future-dependent, conditional flow policies with prophetic variables. In Subsection 5.1, we intuitively describe how the prophetic variables in our policies are dealt with in the type system. In Subsection 5.2, we present the typing rules and illustrate their uses.

5.1 Dealing with Prophetic Variables

As policies can be dependent on prophetic variables, in the type system we need information not only on the program variables in the policies (as can be provided by a Hoare logic as in [15] and [8]), but also on the prophetic variables. We provide some intuition on how the information about the prophetic variables can be provided and utilized before moving on to the formal development.

In the questionnaire scenario of Section 2, it is sensible for the program variable *res* to have the policy

$$\curlywedge_{0\leq j<100} (\underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\}), \tag{2}$$

saying that information about the statistics generated from all the answers can only be revealed to the participants. In the output $\mathsf{send}(c'_i, res)$ of the statistics to the user i, since $\mathcal{P}^\bullet(c'_i) = \{USR_i\}$, intuition tells us that we need to know $\underline{par_i} = 0$ is *not* the case; otherwise the subtraction $... \setminus \{\overline{USR_i}\}$ in the policy of *res* will prevent USR_i from obtaining any information about *res*. Fortunately, we know (and can formally verify) that for each $\underline{par_j}$, once it is set to 1, its value remains to be 1:

$$\bigwedge_{0\leq j<100} (par_j = 1 \Rightarrow \underline{par_j} = 1). \tag{3}$$

On this basis, it is not difficult to infer $\underline{par_i} = 1$, because the output $\mathsf{send}(c'_i, res)$ resides in the true branch of the conditional $par_i = 1$ in the program.

In the above, the formula (3) has been the key ingredient that is informative about the prophetic variables. Since prophetic variables predict the final values of variables, their values are supposed to be the same as their corresponding program variables, at the end of the program in Figure 1:

$$\bigwedge_{0\leq j<100} (par_j = \underline{par_j}). \tag{4}$$

The formula (3) has two properties:

1. it is a logical consequence of (4), which itself must hold at the end of the questionnaire program;
2. it is preserved backwards, from the end to the beginning.

In our type system, a formula like (3) will be an important component that makes information about prophetic variables accessible in all program points. Such a formula will correspondingly be called a *prophetic formula*.

Remark 1. *If the policy of ans_i was dependent on the* current *values of all the flag variables par_j, its policy*

$$\curlywedge_{0\leq j<100} (par_j = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\})$$

would directly allow the aforementioned output to leak information to USR_i. However, before finally reaching the output of the statistics, the policy of ans_i would go through a series of weakenings (downgradings). Consider the case $i = 0$. After the assignment $par_1 := 1$ during the period when the questionnaire is open, the policy of ans_0 would be downgraded to allow the readership of USR_1. Our type system and security property will maintain the difference between conditional flow policies and downgrading, and will correspondingly forbid such downgradings.

5.2 The Type System

The typing of a statement S is performed in two stages, the first one dealing with the main constraints on policies created by the dependencies between the channels and variables of S, and the second one imposing some additional healthiness conditions.

The First Stage of the Type System..

The first stage establishes the judgment

$$\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ S\ (\{\phi'\}, l', X').$$

The $\mathcal{P}$ is a policy environment. The precondition ϕ and postcondition ϕ' obey Hoare logic rules, and provide information on the current values of variables. The formula ϕ_{pph} is a prophetic formula that provides information about the final values of variables. The role of "PC labels" that keep track of implicit flows is played by X and X' that are two sets of variables whose information may be implicitly leaked. On the other hand, l and l' can be understood as "PC labels" that capture the information that can be implicitly leaked via the presence of messages in communication. The way l and l' are updated and used will be explained in detail when discussing the typing rules.

The typing rules of the first stage is given in Figure 6. A few pieces of notation need to be clarified first. The notation $\widetilde{\cdot}$ expands channels and variables into sets of instances by giving concrete values to any variable indices contained in them. For a variable-indexed channel or variable u_i, where i is itself a program variable, $\widetilde{u_i} = \{u_0, u_1, ...\} \cap \mathbf{Var}$. For a channel or variable u not indexed by a variable, $\widetilde{u} = \{u\}$. We lift this notation to sets U of channels and variables in the natural way: $\widetilde{U} = \bigcup_{u\in U} \widetilde{u}$.

We write $\mathcal{P}_{\text{var}}$ for the restriction of $\mathcal{P}^\bullet$ to the smaller domain containing only variables (note that only content policies are involved), $\mathcal{P}^\bullet[U]$ for the policy $\curlywedge_{u\in U} \mathcal{P}^\bullet(u)$ where U can contain variables and channels, $\mathcal{P}^\circ[U]$ for the policy $\curlywedge_{c\in U} \mathcal{P}^\circ(c)$, $\phi \Rightarrow \mathcal{P}$ for the *content* policy environment $\mathcal{P}_1^\bullet$ such that $\mathbf{D}_{\mathcal{P}_1^\bullet} = \mathbf{D}_{\mathcal{P}}$ and $\forall u \in (\mathbf{Var} \cup \mathbf{Ch}) \cap \mathbf{D}_{\mathcal{P}_1^\bullet} : \mathcal{P}_1^\bullet(u) = (\phi \Rightarrow \mathcal{P}^\bullet(u))$, $\mathcal{P}[u \mapsto P]$ for an update or an extension of $\mathcal{P}^\bullet$, mapping $u \in \mathbf{Var} \cup \mathbf{Ch}$ to P, depending on whether u is already in the domain of $\mathcal{P}^\bullet$ or not, $\mathcal{P}[a/x]$ (resp. $\mathcal{P}[c/x]$) for a substitution of a for x (resp. c for x) in all the conditions of the policies recorded in $\mathcal{P}^\bullet$. Finally, we lift the policy comparison operation $\preceq$ introduced in Section 4 to policy environments:

$$\mathcal{P}_1^\bullet \preceq \mathcal{P}_2^\bullet \triangleq \forall u \in \mathbf{D}_{\mathcal{P}_1} \cap \mathbf{D}_{\mathcal{P}_2} \cap (\mathbf{Var} \cup \mathbf{Ch}) : \mathcal{P}_1^\bullet(u) \preceq \mathcal{P}_2^\bullet(u).$$

Essentially, $\mathcal{P}_1^\bullet \preceq \mathcal{P}_2^\bullet$ calls for syntactical comparison of the policies of all variables and channels in the intersection of the domains of $\mathcal{P}_1^\bullet$ and $\mathcal{P}_2^\bullet$.

In Figure 6, the occurrences of the prophetic formula ϕ_{pph} in red correspond to the validation of its backward preservation, while the occurrences in blue correspond to its usage in the comparison of policies.

We direct our attention to the individual typing rules. The rule for skip is trivial — neither the pre-condition ϕ nor the security context consisting of l and X is altered.

$$\frac{}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ \mathsf{skip}\ (\{\phi\}, l, X)}$$

$$\frac{\begin{array}{c}\phi \Rightarrow \phi'[a/x] \qquad (\phi' \wedge \phi_{\text{pph}})[a/x] \Rightarrow \phi_{\text{pph}} \\ \phi_{\text{pph}} \wedge \phi \Rightarrow \mathcal{P}_{\text{var}}[x \mapsto \mathcal{P}^{\bullet}[X \cup fv(a)] \curlywedge l] \preceq \mathcal{P}_{\text{var}}[a/x]\end{array}}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ x := a\ (\{\phi'\}, l, X)}$$

$$\frac{\begin{array}{c}\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ S_1\ (\{\phi''\}, l_1, X_1) \\ \mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi''\}, l_1, X_1)\ S_2\ (\{\phi'\}, l_2, X_2)\end{array}}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ S_1; S_2\ (\{\phi'\}, l_2, X_2)}$$

$$\frac{\begin{array}{c}\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi \wedge b\}, l, X \cup \widetilde{fv(b)})\ S_1\ (\{\phi'\}, l', X') \\ \mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi \wedge \neg b\}, l, X \cup \widetilde{fv(b)})\ S_2\ (\{\phi'\}, l', X')\end{array}}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\ (\{\phi'\}, l', X')}$$

$$\frac{X \cup \widetilde{fv(b)} \subseteq X' \quad \mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi \wedge b\}, l, X')\ S\ (\{\phi\}, l, X')}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}\ (\{\phi \wedge \neg b\}, l, X')}$$

$$\frac{\begin{array}{c}l \curlywedge (\phi_{\text{pph}} \wedge \phi \Rightarrow \mathcal{P}^{\bullet}[X]) \preceq \mathcal{P}^{\circ}(c) \\ \phi_{\text{pph}} \wedge \phi \Rightarrow \mathcal{P}_{\text{var}}[c \mapsto \mathcal{P}^{\bullet}[fv(a) \cup X] \curlywedge l] \preceq \mathcal{P}_{\text{var}}\end{array}}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, l, X)\ \mathsf{send}(c, a)\ (\{\phi\}, l, X)}$$

$$\frac{\begin{array}{c}(\phi \wedge \phi_{\text{pph}})[c/x] \Rightarrow \phi_{\text{pph}} \\ l \curlywedge (\phi_{\text{pph}} \wedge \forall x.\phi \Rightarrow \mathcal{P}^{\bullet}[X]) \preceq \mathcal{P}^{\circ}(c) \\ \phi_{\text{pph}} \wedge \forall x.\phi \Rightarrow \mathcal{P}_{\text{var}}[x \mapsto \mathcal{P}^{\bullet}[\{c\} \cup X]] \preceq \mathcal{P}_{\text{var}}[c/x]\end{array}}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\forall x.\phi\}, l, X)\ \mathsf{recv}(c, x)\ (\{\phi\}, \mathcal{P}^{\circ}[\widetilde{c}], X)}$$

$$\frac{\begin{array}{c}(\phi \wedge \phi_{\text{pph}})[c/x] \Rightarrow \phi_{\text{pph}} \\ l \curlywedge (\phi_{\text{pph}} \wedge \forall x.\phi \Rightarrow \mathcal{P}^{\bullet}[X]) \preceq \mathcal{P}^{\circ}(c) \\ \phi_{\text{pph}} \wedge \forall x.\phi \Rightarrow \mathcal{P}_{\text{var}}[x \mapsto \mathcal{P}^{\bullet}[\{c\} \cup X]] \preceq \mathcal{P}_{\text{var}}[c/x] \\ \mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, \mathcal{P}^{\circ}[\widetilde{c}], X)\ S_1\ (\{\phi'\}, l', X') \\ \mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi\}, \mathcal{P}^{\circ}[\widetilde{c}], X)\ S_2\ (\{\phi'\}, l', X')\end{array}}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\forall x.\phi\}, l, X)\ \mathsf{if\text{-}recv}(c, x) : S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\ (\{\phi'\}, l', X')}$$

$$\frac{\begin{array}{c}\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi_1'\}, l_1', X_1')\ S\ (\{\phi_2'\}, l_2', X_2') \\ l_1 \preceq l_1' \quad l_2' \preceq l_2 \quad X_1 \subseteq X_1' \quad X_2' \subseteq X_2 \\ \phi_1 \Rightarrow \phi_1' \quad \phi_2' \Rightarrow \phi_2\end{array}}{\mathcal{P}, \phi_{\text{pph}} \vdash (\{\phi_1\}, l_1, X_1)\ S\ (\{\phi_2\}, l_2, X_2)}$$

Figure 6: The Information Flow Type System

The rule for assignments $x := a$ follows Hoare logic to require that the precondition ϕ should imply $\phi'[a/x]$ where ϕ' is the postcondition (first premise). The rule soundly guarantees that if the prophetic formula ϕ_{pph} holds after the assignment is performed, it also holds beforehand (second premise). The third premise constructs two policy environments, $\phi_{\text{pph}} \wedge \phi \Rightarrow \mathcal{P}_{\text{var}}[x \mapsto \mathcal{P}[X \cup fv(a)] \curlywedge l]$ and $\mathcal{P}_{\text{var}}[a/x]$, and aims to ensure that they obey the syntactical order $\preceq$, which has been lifted to policy environments. Essentially, there is a policy constraint imposed on each variable. For x, we have the following comparison of conditional policies:

$$\phi_{\text{pph}} \wedge \phi \Rightarrow \mathcal{P}[X \cup fv(a)] \curlywedge l \ \preceq\ \mathcal{P}^{\bullet}(x)[a/x].$$

Here the substitution $[a/x]$ bridges the post-state and the pre-state, the coverage of the variables in $fv(a)$ in constructing the policy on the left-hand side takes care of the explicit flows from the free variables of a to x, the consideration of X and l takes implicit flows from conditionals, as well as previous blockage of inputs, into account, and the implication from $\phi_{\text{pph}} \wedge \phi$ incorporates what is known about the current values (ϕ) and the final values (ϕ_{pph}) of variables. For variables $x' \neq x$, we have the following comparison of conditional policies:

$$\phi_{\text{pph}} \wedge \phi \Rightarrow \mathcal{P}^{\bullet}(x') \preceq \mathcal{P}^{\bullet}(x')[a/x].$$

Since ϕ_{pph} and ϕ are satisfied before the assignment (which is guaranteed by the type system itself), the constraint above essentially requires the policy of x' after the assignment to be no less restrictive than its policy before, thereby eliminating the possibilities of downgrading the actual policy of x'.

The typing rules for output and blocking input have certain commonalities with the assignment rule, since certain analogies can be identified with assignments, when thinking of a channel as a special variable, although output and blocking input also affect the presence of values.

The rule for outputs $\mathsf{send}(c, a)$ uses the postcondition ϕ directly as the precondition, since no variable update is involved. For the same reason, the validity of ϕ_{pph} must be preserved backwards, and no explicit constraint in this regard is needed. The PC component X governing implicit flows via conditionals is also naturally kept the same. The first premise ensures that the presence of messages buffered for c cannot implicitly leak the presence of messages buffered for channels involved in previous communications (l), or the content of passed conditionals (X). Since an output to c is non-blocking, and the presence of messages bufferred for c when the output is reached cannot be leaked via low effects in the continuation, the "presence context" l remains the same after the output. The second premise is devised by exploiting the analogy of $\mathsf{send}(c, a)$ and the imaginary assignment construct $c := a$. At first sight, the substitution $[a/c]$ is needed on the second occurrence of $\mathcal{P}_{\text{var}}$ by this analogy; however, no policy is allowed to depend on channels, which means $\mathcal{P}_{\text{var}}[a/c] = \mathcal{P}_{\text{var}}$. Note that no downgrading of the actual policy of any variable or channel is allowed.

The rule for blocking inputs $\mathsf{recv}(c, x)$ derives the precondition $\forall x.\phi$ given the postcondition ϕ. This is because of the uncertainties about the environment, and correspondingly about what can be received from c into x. As in the rule for assignments, the first premise ensures the backward preservation of the prophetic formula ϕ_{pph}. In the substitution $[c/x]$, c is a symbolic variable having the same name as the input channel, capturing an arbitrary value of x after the input. The second premise is concerned with implicit flows involving the presence of messages. As in the output case, it ensures that the presence of messages buffered for c cannot implicitly leak the presence of messages buffered for channels involved in previous communications (l), or the content of passed conditionals (X). Since the presence of messages bufferred for c when the input is reached can be leaked via low effects in the continuation, the presence context is raised to $\mathcal{P}^{\circ}[\widetilde{c}]$ after the input. The third premise is devised by exploiting the analogy of $\mathsf{recv}(c, x)$ and the imaginary assignment construct $x := c$.

The rule for non-blocking inputs $\mathsf{if\text{-}recv}(c, x) : S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$ shares certain commonalities with the rule for $\mathsf{recv}(c, x)$ since an input from c to x is involved as long as there are messages present for c. A major difference is that the presence of messages buffered for the input channel affects the continuation due not to the blockage of execution, but to the execution of different branches (S_1 or S_2). Hence we type S_1 and S_2 in the presence context $\mathcal{P}^{\circ}[\widetilde{c}]$ (which is above l since $\mathcal{P}^{\circ}(c)$ is required to be), capturing such implicit flows of presence information. In the typing of S_1 and S_2, the security contexts at the end are required to be the same (l',

X'), which is taken to be the security context at the end of the non-blocking input. This is feasible because our subtyping rule will allow the adjustment of finishing contexts into more confidential ones.

We briefly comment on the typing rules for the composite statements including sequential composition, if and while. The Hoare logic components are in accordance with standard Hoare logic rules. There is no need to impose explicit constraints for the backward preservation of the prophetic formula, since the only possible updates to variables (that necessitate such explicit constraints) must happen inside the sub-statements. In addition, the spirit embodied in the treatment of the security contexts l and X is analogous to that of basic type systems enforcing unconditional policies.

Finally, the subtyping rule in Figure 6 allows us to strengthen the precondition, weaken the postcondition, lower the security context in the beginning of a statement, and raise it in the end.

EXAMPLE 3. *The questionnaire program of Figure 1, denoted $S_{\rm qs}$, is well-typed as*

$$\mathcal{P}, \phi_{\rm pph} \vdash (\{\mathsf{true}\}, \star, \emptyset)\ S_{\rm qs}\ (\{\mathsf{true}\}, \star, \{cont, i\} \cup \bigcup_i \{par_i\})$$

with the policy (1) *for $\mathcal{P}^\bullet(c_i)$ and $\mathcal{P}^\bullet(ans_i)$ for each i, the policy* (2) *for $\mathcal{P}^\bullet(res)$, the policy $\{USR_i\}$ for each c'_i, and the policy $\star$ as all the other presence/content policies. We use $par_i = 1$ as the precondition of the assignment of line 7, and the output of line 19, and* true *as the other preconditions and postconditions.*

We type the assignment of line 7 as:

$$\begin{array}{ll}\mathcal{P}, \phi_{\rm pph} \vdash & (\{par_i = 1\}, \star, \{cont, i\} \cup \bigcup_i \{par_i\}) \\ & \quad res := aggr(res, i, ans_i) \\ & (\{\mathsf{true}\}, \star, \{cont, i\} \cup \bigcup_i \{par_i\})\end{array}$$

where $\phi_{\rm pph} = \bigwedge_{0 \le j < 100}(par_j = 1 \Rightarrow \underline{par_j} = 1)$. It is not difficult to verify that the first premise and the second premise of the assignment rule hold. The third requires

$$\begin{array}{l}(\phi_{\rm pph} \wedge (par_i = 1) \Rightarrow \\ \quad \mathcal{P}_{\rm var}[res \mapsto \mathcal{P}[\{cont, i, res, ans_i\} \cup \bigcup_i \{par_i\}] \curlywedge \star]) \\ \preceq \mathcal{P}_{\rm var}[aggr(res, i, ans_i)/res]\end{array} \qquad (5)$$

For the variable res, the requirement boils down to

$$\begin{array}{l}(\phi_{\rm pph} \wedge (par_i = 1) \Rightarrow \mathcal{P}[\{cont, i, res, ans_i\} \cup \bigcup_i \{par_i\}] \curlywedge \star) \\ \preceq\ \curlywedge_{0 \le j < 100}(\underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\})\end{array}$$

Unfolding the notation $\mathcal{P}[U]$, the above further amounts to

$$\begin{array}{l}(\phi_{\rm pph} \wedge (par_i = 1) \Rightarrow \\ \quad ((\underline{par_i} = 1\ \Rightarrow \curlywedge_{0 \le j < 100}(\underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\}))\ \curlywedge \\ \quad\ (\underline{par_i} = 0\ \Rightarrow\ \{USR_i\}))\ \curlywedge \\ \quad\ \curlywedge_{0 \le j < 100}(\underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\}) \curlywedge \star \curlywedge \star \curlywedge \star \curlywedge \star) \\ \preceq\ \ \curlywedge_{0 \le j < 100}(\underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\})\end{array}$$

Unfolding the definition of $\preceq$ and using the congruence relation $\equiv$, it is sufficient to establish the following:

$$\begin{array}{ll} & \curlywedge_j((\phi_{\rm pph} \wedge par_i = 1 \wedge \underline{par_i} = 1 \wedge \underline{par_j} = 0) \Rightarrow \mathbf{Pr} \setminus \{USR_j\}) \\ & \curlywedge (\phi_{\rm pph} \wedge par_i = 1 \wedge \underline{par_i} = 0 \Rightarrow \{USR_i\}) \\ & \curlywedge\ \curlywedge_j(\phi_{\rm pph} \wedge (par_i = 1) \wedge \underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\}) \\ \preceq_{\rm INF} & \curlywedge_j(\underline{par_j} = 0 \Rightarrow \mathbf{Pr} \setminus \{USR_j\})\ \curlywedge\ (\mathsf{false} \Rightarrow \emptyset)\end{array}$$

Using the definition of $\preceq_{\rm INF}$ given in Section 4, we can indeed verify the above condition. In more detail, for each conjunct with index j_0 in the first line and the third line, and each principal not in $\mathbf{Pr} \setminus \{USR_{j_0}\}$, it is sufficient to pick the conjunct in the last line with index j_0. For the conjunct of the second line, it is sufficient to pick the conjunct $\mathsf{false} \Rightarrow \emptyset$ *in the last line. This is because we have*

$$(\phi_{\rm pph} \wedge par_i = 1 \wedge \underline{par_i} = 0)\ \Rightarrow\ \mathsf{false}.$$

Note that information contained in the prophetic formula $\phi_{\rm pph}$ is exploited in deriving the implication. We can also verify the other constraints imposed by (5) *using the same definitions.*

Another spot in the typing of the program in Figure 1 that involves actual use of $\phi_{\rm pph}$ is the output in line 19. We have informally discussed the key insight in the typing of this output in the beginning of this section. Formally, we can establish

$$\begin{array}{ll}\mathcal{P}, \phi_{\rm pph} \vdash & (\{par_i = 1\}, \star, \{cont, i\} \cup \bigcup_i \{par_i\}) \\ & \quad \mathsf{send}(c'_i, res) \\ & (\{par_i = 1\}, \star, \{cont, i\} \cup \bigcup_i \{par_i\})\end{array}$$

This can be verified by applying the same definitions as those for the typing of the assignment of line 7; hence we dispense with going into further details.

Finally, the prophetic formula $\phi_{\rm pph}$ is indeed backward preserved. The only non-trivial case is the assignment $par_i :=$ 1 of line 6, changing the value of par_i. We have

$$\begin{array}{l}(par_i = 1 \wedge \bigwedge_j(par_j = 1 \Rightarrow \underline{par_j} = 1))[1/par_i] \Rightarrow \\ \bigwedge_j(par_j = 1 \Rightarrow \underline{par_j} = 1),\end{array}$$

where $par_i = 1$ is the postcondition of line 6.

The Second Stage of the Type System.

The second stage of our type system builds on the first and establishes the judgment

$$\mathcal{P}, \phi_{\rm pph} \vdash S,$$

representing that S is well-typed under the policy environment $\mathcal{P}$, and the prophetic formula $\phi_{\rm pph}$. The only typing rule of this stage is shown below.

$$\frac{\begin{array}{c}\mathcal{P}, \phi_{\rm pph} \vdash (\{\mathsf{true}\}, \star, \emptyset)\ S\ (\{\mathsf{true}\}, l', X') \\ (\bigwedge_{\underline{x} \in fv(\mathcal{P})}(x = \underline{x})) \Rightarrow \phi_{\rm pph} \quad nip(\mathcal{P}) \quad pph(\mathcal{P}, l', X', \phi_{\rm pph})\end{array}}{\mathcal{P}, \phi_{\rm pph} \vdash S}$$

The first premise simply says that S should be typable according to the rules of stage 1, with true as the pre- and post- condition, and with the most permissive starting context $l = \star$ and $X = \emptyset$. The second premise says that the prophetic formula $\phi_{\rm pph}$ needs to be implied by a conjunction of clauses equating all the prophetic variables appearing in any policy recorded in $\mathcal{P}$ to their corresponding program variables. This ensures the satisfaction of $\phi_{\rm pph}$ at the end of the program execution. Note that $fv(\mathcal{P}) = \bigcup_{c \in \mathbf{D}_\mathcal{P}} fv(\mathcal{P}^\circ(c))\ \cup\ \bigcup_{u \in \mathbf{D}_\mathcal{P}} fv(\mathcal{P}^\bullet(u))$.

The last two premises are two "healthiness" conditions, to be explained next.

The first condition, $nip(\mathcal{P})$, is concerned with *policy-level leakage*. Consider, for example, the policy $(x \ge 0 \Rightarrow \{A_1, A_2\}) \wedge (x < 0 \Rightarrow \{A_2, A_3\})$. If A_1 gets to know that it is not granted readership of the information protected by this policy, then A_1 knows that the value of x is less than 0. Thus there is information flow from the actual policies to the conditions. The condition $nip(\mathcal{P})$ requires for each policy P recorded in $\mathcal{P}$ that all principals not shared by all the actual policies (such as A_1 and A_3 in this example) should be always allowed as readers of the variables in P. The same kind of considerations have also been applied in previous developments on content-dependent flow policies (e.g., [8, 12]), and we postpone the detailed definition of $nip(\mathcal{P})$ to the appendix. Note that it is not always true that the security principals can observe their own readership, but eliminating such policy-level leakage is most in line with the spirit of noninterference.

The second condition, $pph(\mathcal{P}, l', X', \phi_{\text{pph}})$, pertains to the use of prophetic variables. It requires that the policy of all variables that have their prophetic counterparts should be equivalent to $\star$, and that in case prophetic variables are used in some of the policies, the finishing context must also be equivalent to $\star$. This is formally expressed as follows.

$$\begin{aligned} pph(\mathcal{P}, l', X', \phi_{\text{pph}}) \triangleq {} & (\forall \underline{x} \in fv(\phi_{\text{pph}}) : \mathcal{P}^{\bullet}(x) \equiv \star) \wedge \\ & (fv(\mathcal{P}) \cap \mathbf{PphVar} \neq \emptyset \Rightarrow l' \curlywedge \mathcal{P}[X'] \equiv \star). \end{aligned}$$

First of all, this condition ensures that the values of variables with prophetic counterparts (the variables par_j in the questionnaire scenario) converge to that of their corresponding prophetic variables, irrespective of any variations of the secrets. Second of all, it ensures that the reachability of the final program point cannot be affected by secrets either. Thus it is impossible that whether the prophetic variables can be instantiated with a combination of values depends on secrets. It is in this situation that a future-dependent security property can be most easily and convincingly formulated, which will become clearer in the next section. The condition $pph(\mathcal{P}, l', X', \phi_{\text{pph}})$ may look demanding. However, in a concurrent setting, which we will consider in Section 8, the prophetic variables may correspond to variables that are updated in only a few of the processes, and correspondingly only the finishing contexts of those processes need to be equivalent to $\star$.

Example 4. *Building on the typing judgment established for the questionnaire program in Example 3, we can establish* $\mathcal{P}, \phi_{\text{pph}} \vdash S_{\text{qs}}$.

The first premise of the typing rule for the second stage is satisfied by the result of Example 3. For the second premise, we indeed have

$$\Big(\bigwedge_{0 \leq i < 100} par_i = \underline{par_i}\Big) \;\Rightarrow\; \bigwedge_{0 \leq i < 100} (par_i = 1 \Rightarrow \underline{par_i} = 1).$$

The variables par_j *are the only ones with prophetic counterparts, and the only ones appearing in the policies. For each* j *we have* $\mathcal{P}^{\bullet}(par_j) = \star$; *hence all the user principals are allowed to observe* par_j. *In addition, the typing judgment for* S_{qs} *established in Example 3 has* $l' = \star$ *and* $X' = \{cont, i\} \cup \bigcup_i \{par_i\}$ *as the finishing context. It is not difficult to see that the requirements of* $nip(\mathcal{P})$ *and* $pph(\mathcal{P}, l', X', \phi_{\text{pph}})$ *are also fulfilled.*

6. FUTURE-DEPENDENT SECURITY

Our discussion in Section 2 already alludes to the idea of enforcing noninterference with each specific valuation of the prophetic variables. In this section, we formally develop the corresponding security property and give the soundness result of our type system with respect to it.

As mentioned in Section 2, the valuations of prophetic variables are represented by mappings $\varsigma \in \mathbf{St}_{\text{pph}}$. The domain $\mathbf{D}_\varsigma$ should be equal to the set of prophetic variables mentioned in policies, i.e., $\mathbf{D}_\varsigma = fv(\mathcal{P}) \cap \mathbf{PphVar}$. The following definition of the notation $\langle S, m\rangle \Vdash \varsigma$ expresses that the prophetic variables obtain their values, as recorded by ς, from their underlying program variables, in the resulting memory m when the execution of S is completed.

Definition 1. *We write* $\langle S, m\rangle \Vdash \varsigma$ *to represent*

$$\mathbf{D}_\varsigma \neq \emptyset \;\Rightarrow\; (S = \mathsf{skip} \wedge \forall \underline{x} \in \mathbf{D}_\varsigma : \varsigma(\underline{x}) = m(x))$$

Note that $\langle S, m\rangle \Vdash \varsigma$ vacuously holds in case no prophetic variables are in use (that is, when $\mathbf{D}_\varsigma = \emptyset$).

The following definition of the notation $\xi \vdash \langle S, m\rangle \xRightarrow{t(\varsigma)}$ builds on the previous one and says that the execution of the program statement S under the environment strategy ξ results in the trace t and the valuation ς of prophetic variables.

Definition 2. *We write* $\xi \vdash \langle S, m\rangle \xRightarrow{t(\varsigma)}$ *to represent that there exist* $S_1 = S$, S_2, ..., S_n, $t_1 = m$, t_2, ..., $t_n = t$, *where* $n \geq 1$, *such that* $\langle S_n, t_n(|t_n|)\rangle \Vdash \varsigma$ *and*

$$n > 1 \;\Rightarrow\; \forall j \in \{2, ..., n\} : \xi \vdash \langle S_{j-1}, t_{j-1}\rangle \to \langle S_j, t_j\rangle.$$

Our security policies involve security principals, and correspondingly it is natural to assume that groups $\mathcal{G}$ of principals can make observation and attempt to deduce further information based on what is observed. At runtime, the conditional policies are actually interpreted as concrete sets of readers. Given such an actual policy, a set L of readers, we think of a group $\mathcal{G}$ of principals as being allowed to observe what is protected by L if and only if $L \cap \mathcal{G} \neq \emptyset$. Our programs are open to the environment and we are mostly concerned with their communication with it, thus with the observation of the queuing buffers of communication channels.

We define in Figure 7 an (overloaded) observation function $ob_\varsigma^{\mathcal{P},\mathcal{G}}(...)$ to capture what can be observed by the group $\mathcal{G}$ of principals, given the policy environment $\mathcal{P}$, and the valuation ς of prophetic variables.

$$\begin{aligned} ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, v) &= \begin{cases} v & \text{if } [\![\mathcal{P}^{\bullet}(c)]\!]\varsigma \cap \mathcal{G} \neq \emptyset \\ \odot & \text{otherwise} \end{cases} \\ ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, \gamma) &= \begin{cases} ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, \gamma(1))...ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, \gamma(|\gamma|)) & \\ \qquad\qquad \text{if } [\![\mathcal{P}^{\circ}(c)]\!]\varsigma \cap \mathcal{G} \neq \emptyset \\ \circledcirc \qquad\qquad \text{otherwise} \end{cases} \\ ob_\varsigma^{\mathcal{P},\mathcal{G}}(m) &= [(c \mapsto ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, m(c)))_{c \in \mathbf{D}_m}] \\ ob_\varsigma^{\mathcal{P},\mathcal{G}}((m_1...m_k)) &= ob_\varsigma^{\mathcal{P},\mathcal{G}}(m_1)...ob_\varsigma^{\mathcal{P},\mathcal{G}}(m_k) \end{aligned}$$

Figure 7: The Observation of Traces

We explain the defining clauses of Figure 7 in a bottom-up order. The observation of a trace $m_1.....m_k$ is produced in a point-wise fashion, by concatenating the observations of the individual memories in it. The observation of a memory, $ob_\varsigma^{\mathcal{P},\mathcal{G}}(m)$, is a mapping of each channel c in the domain of the memory to the observation $ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, m(c))$ of the buffer $m(c)$ of elements queued up for c. The observation $ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, \gamma)$ of a queue γ of elements buffered for the channel c is completely masked by a $\circledcirc$ in case the actual presence policy of c is confidential with respect to $\mathcal{G}$. Otherwise the observation of the queue is composed of the observation of the individual elements $\gamma(j)$ $(1 \leq j \leq |\gamma|)$, given by $ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, \gamma(j))$. Each element is a value v, and its observation $ob_\varsigma^{\mathcal{P},\mathcal{G}}(c, v)$ is exactly v if the actual content policy of c, obtained by interpreting $\mathcal{P}^\bullet(c)$ in ς, is non-confidential with respect to $\mathcal{G}$. Otherwise the value is masked by a $\odot$. We now explain the distinction of the two kinds of masks with two small examples. If $ob_\varsigma^{\mathcal{P},\mathcal{G}}(c,\gamma)$ results in $\odot$, then it is observed that the queuing buffer of c has exactly *one element*, but not observed what the element is. On the other hand, if $ob_\varsigma^{\mathcal{P},\mathcal{G}}(c,\gamma)$ results in $\circledcirc$, then even the size of the queue (in particular, whether it is empty or not) is not known to the observer, not to mention the elements (if any) contained.

Under each particular valuation ς of the prophetic variables, our security property will aim to ensure that the variations on the presence and content of communications introduced by the strategies of the environment, cannot trigger variations in the observation of the execution traces. To this end, we need to ensure that strategies do not introduce variations directly in the *observable* facets of their communication attempts.

We first define the notation $\alpha_1 \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{=}} \alpha_2$ where α_1 and α_2 range over the set $\{\mathsf{send}(c,v), \mathsf{recv}(c,[\,]), \diamond \mid c \in \mathbf{Ch}, v \in \mathbf{Val}\}$, to capture the observational equivalence of two communication attempts of the environment. The relation $\overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{=}}$ is the least one satisfying the following rules, where for $\alpha \neq \diamond$, $ch(\alpha)$ represents the channel of α, i.e., for any v, $ch(\mathsf{send}(c,v)) = c$, and $ch(\mathsf{recv}(c,[\,])) = c$.

$$\frac{ob_\varsigma^{\mathcal{P},\mathcal{G}}(c,v) = ob_\varsigma^{\mathcal{P},\mathcal{G}}(c,v')}{\mathsf{send}(c,v) \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{=}} \mathsf{send}(c,v')} \qquad \frac{}{\mathsf{recv}(c,[\,]) \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{=}} \mathsf{recv}(c,[\,])}$$

$$\frac{\forall j \in \{1,2\} : (\alpha_j \neq \diamond \Rightarrow [\![\mathcal{P}^\circ(ch(\alpha_j))]\!]\varsigma \cap \mathcal{G} = \emptyset)}{\alpha_1 \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{=}} \alpha_2}$$

For communication channels c whose actual presence policy is not confidential with respect to $\mathcal{G}$, a communication attempt over c can only be equivalent to another one of the same kind. In the output case, whether the communication content needs to be the same will depend on the content policy of c. On the other hand, for channels c with confidential presence policies, a communication attempt over c can be equivalent to one of a different kind (including $\diamond$) and/or over a different channel. A $\diamond$ is always equivalent to a $\diamond$.

We introduce now the last piece of auxiliary notation: $\xi_1 \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{\simeq}} \xi_2$ says that given traces with the same observation, the strategies ξ_1 and ξ_2 produce observationally equivalent communication attempts. Thus no variation is introduce directly in the observable facets of communication attempts.

$$\xi_1 \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{\simeq}} \xi_2 \triangleq \forall t_1, t_2 : ob_\varsigma^{\mathcal{P},\mathcal{G}}(t_1) = ob_\varsigma^{\mathcal{P},\mathcal{G}}(t_2) \Rightarrow \xi_1(t_1) \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{=}} \xi_2(t_2)$$

We are in a position to formally express our security property — future-dependent security. A program S is secure under the policy environment $\mathcal{P}$, denoted $Sec(S,\mathcal{P})$, if for all sets ES of environment strategies, strategies ξ_1 and ξ_2 in ES, valuations ς for prophetic variables, and groups $\mathcal{G}$ of principals, such that ξ_1 and ξ_2 are observationally equivalent to $\mathcal{G}$ under ς, for any trace t_1 produced under ξ_1, leading to the given valuation ς of prophetic variables, there exists a trace t_2 produced under ξ_2, also leading to the given valuation ς of prophetic variables, such that t_1 and t_2 are indistinguishable to the group $\mathcal{G}$ of observers. This is covered in Definition 3, where m_0 is the memory where all variables are 0 and all channels are empty.

Definition 3 $\quad$ ($Sec(S,\mathcal{P})$).

$$\begin{array}{l} Sec(S,\mathcal{P}) \triangleq \\ \quad \forall ES : \forall \xi_1, \xi_2 \in ES, \varsigma \in \mathbf{St}_{\mathrm{pph}} : \forall \mathcal{G} \;\; s.t. \;\; \xi_1 \overset{\mathcal{P},\mathcal{G}}{\underset{\varsigma}{\simeq}} \xi_2 : \\ \qquad \forall t_1 \;\; s.t. \;\; \xi_1 \vdash \langle S, m_0 \rangle \xrightarrow{t_1(\varsigma)} \; : \\ \qquad\quad \exists t_2 : \xi_2 \vdash \langle S, m_0 \rangle \xrightarrow{t_2(\varsigma)} \; \wedge \; ob_\varsigma^{\mathcal{P},\mathcal{G}}(t_1) = ob_\varsigma^{\mathcal{P},\mathcal{G}}(t_2) \end{array}$$

The use of existential quantification over the trace t_2 is based on two considerations. First of all, this guarantees that the same valuation of prophetic variables can also be reached under ξ_2, which avoids the situation where the valuations of prophetic variables are always different when the secrets take different values, rendering the security requirement vacuous. Second of all, we will eventually extend the development to a concurrent setting in Section 8, and using a "simulation" style allows seamless generalization to the concurrent setting.

Example 5. *We revisit the questionnaire example and examine the requirements of future-dependent security on the program of Figure 1.*

Take arbitrary strategies ξ_1 and ξ_2, the valuation $\varsigma_{3,20,55,58}$ of prophetic variables, and the singleton group $\mathcal{G}_5 = \{USR_5\}$ of principals, such that $\xi_1 \overset{\mathcal{P},\mathcal{G}_5}{\underset{\varsigma_{3,20,55,58}}{\simeq}} \xi_2$ and

$$\xi_1 \vdash \langle S_{\mathrm{qs}}, m_0 \rangle \xrightarrow{t_1(\varsigma_{3,20,55,58})} .$$

Suppose the first communication attempt of ξ_1, that is not $\diamond$, is $\mathsf{send}(c_{20}, v_1)$ for some value v_1, and this happens with trace t_1' (a prefix of t_1), i.e., $\xi_1(t_1') = \mathsf{send}(c_{20}, v_1)$.

Consider the potential execution of S_{qs} starting with m_0, under ξ_2. While ξ_1 is still producing $\diamond$ before its first non-$\diamond$ communication attempt, the execution of S_{qs} must be following the same path, and computing the same values, as the one under ξ_1. This is because the presence policy of the channels is $\star$, which is non-confidential w.r.t. $\mathcal{G}_5$. Hence ξ_2 must also be producing $\diamond$, and no differences are introduced into the memories. This will lead to a trace $t_2' = t_1'$, and $ob_{\varsigma_{3,20,55,58}}^{\mathcal{P},\mathcal{G}_5}(t_1') = ob_{\varsigma_{3,20,55,58}}^{\mathcal{P},\mathcal{G}_5}(t_2')$ holds. Hence we are supposed to have $\xi_1(t_1') \overset{\mathcal{P},\mathcal{G}_5}{\underset{\varsigma_{3,20,55,58}}{=}} \xi_2(t_2')$, which boils down to $\mathsf{send}(c_{20}, v_1) \overset{\mathcal{P},\mathcal{G}_5}{\underset{\varsigma_{3,20,55,58}}{=}} \xi_2(t_2')$. Since $\mathcal{P}^\circ(c_{20}) = \star$, and $[\![\mathcal{P}^\bullet(c_{20})]\!]\varsigma_{3,20,55,58} = \{USR_3, USR_{20}, USR_{55}, USR_{58}\}$, which follows directly from the calculation made in Example 1, we know that $\mathcal{P}^\circ(c_{20})$ is public w.r.t. $\mathcal{G}_5$, while $\mathcal{P}^\bullet(c_{20})$ is confidential. Hence $\xi_2(t_2') = \mathsf{send}(c_{20}, v_2)$ for some value v_2 that is not necessarily the same as v_1. The values v_1 and v_2 will be received into ans_{20} whose policy is also confidential w.r.t. $\mathcal{G}_5$, after the traces t_1' and t_2' of the same length.

Since ξ_2 will make an output attempt over a channel whenever ξ_1 makes an output attempt over the same channel, although the output values may well be different, it is not difficult to see that there exists a trace t_2 such that

$$\xi_2 \vdash \langle S_{\mathrm{qs}}, m_0 \rangle \xrightarrow{t_2(\varsigma_{3,20,55,58})}$$

and $ob_{\varsigma_{3,20,55,58}}^{\mathcal{P},\mathcal{G}_5}(t_1) = ob_{\varsigma_{3,20,55,58}}^{\mathcal{P},\mathcal{G}_5}(t_2)$. This would not have been possible if the program had, e.g., output res to c_5.

Remark 2. *In case no prophetic variables are used, and the policies are dependent only on the current state, we have $\mathbf{D}_\varsigma = \emptyset$, and future-dependent security degenerates to a property that is close to the* **Strat**-*NI of [16], which also embodies the spirit of non-deducibility on strategies [22].*

The following theorem states that the type system of Section 5 soundly enforces future-dependent security.

THEOREM 1. *If there exist ϕ_{pph} such that $\phi_{\text{pph}}, \mathcal{P} \vdash S$, then we have $Sec(S, \mathcal{P})$.*

7. A LOAD-BALANCING SCENARIO

In Section 2, we have given an example where the use of future-dependent flow policies avoids having to downgrade the actual policies of inputs. In open systems, the uncertainties about future inputs from the environment can sometimes lead to changes in present-dependent policies that are more abrupt and non-sensible, further motivating the use of future-dependent flow policies.

The following program waits for some input data from the channel c_{in}, on which some local processing ($g(x_{\text{in}}, ...)$) is performed, and the result, stored in *res*, will be delivered to one of the nodes N_1 and N_2 in the network for further, remote processing. Whether N_1 or N_2 is chosen depends on which node is having a comparably lighter load when the result *res* of the local processing is ready. Hence the program sends requests for the current load to both nodes through the channels c_1 and c_2, respectively, retrieves the corresponding load data from the channels c'_1 and c'_2, and stores the figures in ld_1 and ld_2. After the load data is retrieved, the flag *ld_info* is set to 1. Finally, *res* is sent to the appropriate node based on a comparison of ld_1 and ld_2.

```
recv($c_{\text{in}}, x_{\text{in}}$);
$res := g(x_{\text{in}}, ...)$;
send($c_1$, "ld_req"); recv($c'_1, ld_1$);
send($c_2$, "ld_req"); recv($c'_2, ld_2$);
$ld\_info := 1$;
if ($ld_1 \geq ld_2$) then
   send($c^1_{\text{dat}}, res$)
else
   send($c^2_{\text{dat}}, res$)
fi
```

Consider the possible information flow policies on x_{in}. To have a more precise policy than true $\Rightarrow \{N_1, N_2\}$ (or $\star$), we need to exploit the condition $ld_1 > ld_2$, and at first sight, a viable choice may be

$$(ld_1 \geq ld_2 \Rightarrow \{N_1\}) \curlywedge (ld_1 < ld_2 \Rightarrow \{N_2\}).$$

However, right after the input into x_{in}, the current values of the variables ld_1 and ld_2 are not, and may not have to do with, the load data influencing the flow of information in x_{in}. In our particular case, both variables are 0 at that point. Supposing after receiving the actual load data into ld_1 and ld_2, it turns out that $ld_1 < ld_2$ holds, an insensible, abrupt change of the policy of x_{in} from $\{N_1\}$ to $\{N_2\}$ will thus be triggered. Using prophetic variables, we can resolve this problem with the the following policy:

$$(\underline{ld_1} \geq \underline{ld_2} \Rightarrow \{N_1\}) \curlywedge (\underline{ld_1} < \underline{ld_2} \Rightarrow \{N_2\}). \tag{6}$$

In fact, using the policy (6) for $\mathcal{P}^\bullet(c_{\text{in}})$, $\mathcal{P}^\bullet(x_{\text{in}})$ and $\mathcal{P}^\bullet(res)$, $\{N_1\}$ for $\mathcal{P}^\bullet(c^1_{\text{dat}})$, $\{N_2\}$ for $\mathcal{P}^\bullet(c^2_{\text{dat}})$, and $\star$ for all the other presence and content policies, we can type the program above with the help of the prophetic formula

$$ld\_info = 1 \ \Rightarrow\ (ld_1 = \underline{ld_1} \wedge ld_2 = \underline{ld_2}).$$

Note that the flag variable *ld_info* is needed not for the articulation of the flow policies, but for the policies to be enforceable by our current type system, which does not allow giving up on the prophetic formula at the program points where it is no longer preserved backward, even if the formula is no longer needed.

There are still two further alternatives worth considering. The first is to explicitly use the flag *ld_info* in $\mathcal{P}^\bullet(x_{\text{in}})$:

$$\begin{aligned}&(ld\_info = 0 \Rightarrow \star) \curlywedge \\ &(ld\_info = 1 \Rightarrow ((ld_1 \leq ld_2 \Rightarrow \{N_1\}) \curlywedge (ld_1 > ld_2 \Rightarrow \{N_2\}))).\end{aligned} \tag{7}$$

In this way, the actual policy always goes from allowing both readers N_1 and N_2 to allowing only one of them — only *upgrading* is involved. However, this leads to a security specification that is somewhat complex, low-level, and loose:

1. As we have mentioned, the flag variable *ld_info* and its assignment $ld\_info := 1$ are needed due to some inflexibility of our type system in the maintenance of the prophetic formula, which is not expected to be difficult to overcome. Introducing *ld_info* explicitly into the policy makes it more complicated and necessitates the retainment of the assignment $ld\_info := 1$, which is irrelevant to the computation task.

2. From the future-dependent policy (6) it is immediately clear that the end-to-end leakage of x_{in} is upper-bounded to either of N_1 and N_2, depending on the final values of the load variables. On the other hand, the present-dependent policy (7) forces the programmer to think in a low-level manner about what happens in the middle, before and after the load data is obtained.

3. The present-dependent policy (7) allows arbitrary leakage of x_{in} to N_1 and N_2 before the assignment to *ld_info*. The insertion of outputs of x_{in} to *both* c^1_{dat} *and* c^2_{dat} before the load data is even obtained, is allowed by (7), but not by (6).

The last alternative that we consider briefly, is to program the task differently. In particular, if we request and receive the load data before the input to x_{in}, the policy of x_{in} will naturally depend on the current values of ld_1 and ld_2. However, it is uncertain how long it will take for the input to retrieve data over c_{in}, and for the local processing to complete. Thus when we send out the local result in *res* consulting the load data, it may have become outdated. A natural specification of present-dependent policies is obtained now at the price of unnatural (to unworkable) programming.

8. A CONCURRENT SETTING

In a concurrent setting, non-local updates to variables that a policy depends upon can have an impact on the strength of the policy that calls for dedicated mechanisms to take under control. In fact, this is one of the key problems dealt with in the recent work [12]. Similar to the logical variables of Hoare logic, prophetic variables can largely be regarded as constants. This has the desirable effect of avoiding the aforementioned problem for policies that depend on prophetic variables, rather than ordinary program variables. In this section, we demonstrate that our development so far can be extended directly to a concurrent setting, where the program variables are local to their own processes, but a policy can nevertheless depend on a prophetic variable belonging to a different process. In the appendix, we illustrate the extension with a distributed questionnaire scenario where the process of Figure 1 communicates with users via client processes, rather than directly.

We consider systems $Sys = S_1||...||S_n$ that consist of concurrent processes S_1, ..., S_n, such that each S_i is in the syntax of Figure 2. The set $\mathbf{Var}$ of variables is now divided among the processes, and the i-th process has the set $\mathbf{Var}_i$ of local variables ($\mathbf{Var} = \mathbf{Var}_1 \uplus ... \uplus \mathbf{Var}_n$, where $\uplus$ is disjoint union). We occasionally use the more compact notation $||_i S_i$ for $S_1 \,||\, ... \,||\, S_n$.

The difference in the semantics, compared with the sequential setting, is only in the environment-aware part. That is, the rules of Figure 4 are now replaced with those of Figure 8, establishing judgments of the form $\xi \vdash \langle Sys, t\rangle \rightarrow \langle Sys', t'\rangle$. Note that the first rule of Figure 8 simply establishes an execution step of the system based on a step of one of its processes. This corresponds to the interleaving semantics of concurrency (as opposed to true concurrency), and the use of a fully nondeterministic scheduler. The remaining rules in Figure 8 are similar to their counterparts in Figure 4.

$$\frac{\langle S_i, t(|t|)\rangle \rightarrow \langle S'_i, m'\rangle}{\xi \vdash \langle ...||S_i||..., t\rangle \rightarrow \langle ...||S'_i||..., t.m'\rangle}$$

$$\frac{\xi(t) = \mathsf{send}(c, v)}{\xi \vdash \langle Sys, t\rangle \rightarrow \langle Sys, t.(t(|t|)[c \mapsto v \cdot t(|t|)(c)])\rangle}$$

$$\frac{\xi(t) = \mathsf{recv}(c, [\,]) \quad t(|t|)(c) = \gamma \cdot v}{\xi \vdash \langle Sys, t\rangle \rightarrow \langle Sys, t.(t(|t|)[c \mapsto \gamma])\rangle}$$

$$\frac{\xi(t) = \mathsf{recv}(c, [\,]) \wedge t(|t|)(c) = \epsilon \ \vee \ \xi(t) = \diamond}{\xi \vdash \langle Sys, t\rangle \rightarrow \langle Sys, t\rangle}$$

Figure 8: The Environment-Aware Semantics for Concurrent Systems

Suppose $\mathcal{P}$ is a policy environment for all variables and channels. For $x \in \mathbf{Var}_i$, we require that the program variables in $\mathcal{P}^\bullet(x)$ should be in $\mathbf{Var}_i$, but allow the prophetic variables in $\mathcal{P}^\bullet(x)$ to be based on program variables in $\mathbf{Var} \setminus \mathbf{Var}_i$. It is convenient to be able to access the part of $\mathcal{P}$ that talks only about the variables local to the i-th process, and all channels. We write $\mathcal{P}_i = (\mathcal{P}^\circ, \mathcal{P}^\bullet\!\downarrow_{\mathbf{Var}_i \cup \mathbf{Ch}})$ where $\mathcal{P}^\bullet\!\downarrow_{\mathbf{Var}_i \cup \mathbf{Ch}}$ represents the restriction of the mapping $\mathcal{P}^\bullet$ to the domain $\mathbf{Var}_i \cup \mathbf{Ch}$.

For the type system, the only changes needed are for the second stage of typing considered in Subsection 5.2. We replace the original typing rule of that stage with the following one, which establishes the judgment $\mathcal{P}, \vec{\phi}_{\text{pph}} \vdash \ ||_i\, S_i$, representing that the system $||_i\, S_i$ is well-typed under the policy environment $\mathcal{P}$ and the vector $\vec{\phi}_{\text{pph}}$ of prophetic formulas.

$$\frac{\begin{array}{c}\forall i: \mathcal{P}_i, \phi^i_{\text{pph}} \vdash (\{\mathsf{true}\}, \star, \emptyset)\ S_i\ (\{\mathsf{true}\}, l'_i, X'_i) \\ \forall i: (\bigwedge_{x \in \mathbf{Var}_i \wedge \underline{x} \in fv(\mathcal{P})} (x = \underline{x})) \Rightarrow \phi^i_{\text{pph}}\ \ nip(\mathcal{P}) \quad \forall i{:}\, pph(\mathcal{P}_i, l'_i, X'_i, \phi^i_{\text{pph}})\end{array}}{\mathcal{P}, \vec{\phi}_{\text{pph}} \vdash \ ||_i\, S_i}$$

The i-th prophetic formula, ϕ^i_{pph}, has the program variables and prophetic variables local to the i-th process as its only free variables. The premises of the typing rule are largely formulated in a per-process manner, using ingredients already explained in Subsection 5.2. We do not go into further details due to space limitation.

To build up to a notion of future-dependent security, we generalize the two pieces of notations, $\langle S, m\rangle \Vdash \varsigma$ and $\xi \vdash \langle S, m\rangle \overset{t(\varsigma)}{\Longrightarrow}$, introduced in Definition 1 and Definition 2, to the $\langle Sys, m\rangle \Vdash \varsigma$ and $\xi \vdash \langle Sys, m\rangle \overset{t(\varsigma)}{\Longrightarrow}$ of Definition 4 and Definition 5 below. We write $Var(M) = \{x \mid \underline{x} \in M\}$.

Definition 4. *We write $\langle Sys, m\rangle \Vdash \varsigma$ to represent that $Sys = S_1||...||S_n$ $(n \geq 1)$, and*

$$\forall i \in \{1, ..., n\} : \mathbf{Var}_i \cap Var(\mathbf{D}_\varsigma) \neq \emptyset \ \Rightarrow$$
$$S_i = \mathsf{skip} \wedge \forall x \in \mathbf{Var}_i \cap Var(\mathbf{D}_\varsigma) : \varsigma(\underline{x}) = m(x).$$

Definition 5. *We write $\xi \vdash \langle Sys, m\rangle \overset{t(\varsigma)}{\Longrightarrow}$ to represent that there exist $Sys_1 = Sys$, Sys_2, ..., Sys_n, $t_1 = m$, t_2, ..., $t_n = t$, where $n \geq 1$, such that $\langle Sys_n, t_n(|t_n|)\rangle \Vdash \varsigma$ and*

$$n > 1 \ \Rightarrow \ \forall j \in \{1, ..., n\} : \xi \vdash \langle Sys_{j-1}, t_{j-1}\rangle \rightarrow \langle Sys_j, t_j\rangle.$$

Our notion of future-dependent security for concurrent systems is now obtained with a direct textual replacement of S in Definition 3 with Sys. We write $Sec(Sys, \mathcal{P})$ to represent that the system Sys is secure under the policy environment $\mathcal{P}$, and we dispense with creating a new definition for it, knowing how it is obtained precisely from Definition 3.

Finally, the following soundness theorem for concurrent systems is a generalization of Theorem 1.

Theorem 2. *If there exist $\vec{\phi}_{\text{pph}}$ such that $\vec{\phi}_{\text{pph}}, \mathcal{P} \vdash Sys$, then we have $Sec(Sys, \mathcal{P})$.*

9. RELATED WORK AND CONCLUSION

As mentioned in the introduction, a number of developments in the last decade [3, 4, 14, 2, 1, 9, 15, 8, 12] have been concerned with content-dependent information flow control. Broberg and Sands [3, 4] use the statuses of locks to control the concrete policies applied on variables. Amtoft et al. [2, 1] devise a relational Hoare logic, and Nanevski et al. [14] develop a relational Hoare type theory, implicitly capturing state-dependent flow policies via conditioned equalities of the same variables in a pair of states. Lourenço and Caires propose value-dependent information flow types for a functional language. Nielson et al., and Li et al. [15, 8] enforce content-dependent flow policies for concurrent processes with local memories and synchronous communication. Murray et al. develop compositional enforcement of value-sensitive flow policies for shared-memory concurrent programs, using assumptions and guarantees (much in the manner of [10]) to constrain cross-thread interference in the values of variables. Tse and Zdancewic [20] introduce run-time principals in the Decentralized Label Model [13], and Zheng and Myers [24] consider first class information flow labels. Both developments allow run-time variability of security policies, without explicitly focusing on value-dependency. *None of the aforementioned developments is concerned with future-dependent policies.*

Two developments, by Micinski et al. [11] and by Dimitrova et al. [6], consider information flow policies that are conditional on formulas in an LTL-like logic. The policies have temporally non-local dependencies on events and values, that appear to give rise to more generality than ours; however, a precise comparison is impossible due to the use of different techniques to address problems in different settings. The technique of [11] is based on symbolic execution, which is not sound, but scales better than the model-checking-based approach of [6] in the presence of a data domain, while [6] enjoys opposite merits to those of [11]. Our technique is sound, and scales well in the presence of data domains as long as the inference of the Hoare logic pre-conditions is automated, and an appropriate prophetic formula is given for each relevant process. Neither [11] nor [6] addresses the differences between conditional flow policies and declassification policies, or touches upon the comparison between future-dependent policies and present-dependent ones, when both could lead to plausible policy specification and enforcement. On the other hand, such comparisons are an important concern of ours. Furthermore, [11] does not consider how its technique caters for concurrency, and [6] works outright with (single) transition systems. On the other hand, our prophetic variables possess the desirable properties that allow us to accommodate concurrency directly in a language-based setting.

As mentioned in Section 2, a downgrading-based viewpoint can be adopted for the questionnaire scenario. For example, Chong and Myers [5] propose policies of the form $l_1 \overset{c_1}{\rightsquigarrow} l_2 \overset{c_2}{\rightsquigarrow} \ldots \overset{c_{n-1}}{\rightsquigarrow} l_n$, where l_1, l_2, ..., l_n is a succession of security policies resulting from downgrading from each l_j to l_{j+1} on the satisfaction of the condition c_j. Although the noninterference-until property proposed and enforced in [5] only imposes a vacuous requirement starting from the first downgrading, policies in the form above are applicable to the questionnaire scenario. However, our use of future-dependent policies avoids downgrading, and other changes of the actual policy in the intermediate states (such as those in the load-balancing scenario) that can also be less sensible than mere downgrading.

In this paper, we have focused on type-based enforcement of qualitative information flow policies. In general, there are various alternatives for the same purpose, such as dynamic/hybrid monitoring, multi-execution, unwinding proofs, etc. Each class has been studied extensively. Downgrading is also not universally avoidable, and downgrading policies have been under extensive examination (e.g., [23, 19]), for performing downgrading operations in a relatively secure manner. We dispense with a more detailed discussion of the specific developments on alternative enforcement methods and on downgrading policies, due to space limitations.

Concluding Remarks.

In this paper, we have demonstrated the meaningfulness of future-dependent flow policies, and developed a static type system that enforces such policies that refer to the final values of program variables using *prophetic variables.* We formulate and guarantee a novel security property, future-dependent security, that ensures noninterference under each particular valuation of prophetic variables. Scenarios favoring future-dependent security to present-dependent security and to downgrading-related perspectives are discussed, and the benefits of prophetic variables in naturally accommodating concurrency is demonstrated.

The generalization of our technique to allow referencing the values of variables at arbitrary program points is a viable topic for future work. We expect this to be achievable by manipulating the prophetic formulas more delicately, e.g., with a static analysis revealing which prophetic formulas — originating at arbitrary program points of concern — must hold at which other program points. Since such propagation of prophetic formulas may then be interrupted halfway, special care needs to be taken in dealing with loops.

10. REFERENCES

[1] T. Amtoft, J. Dodds, Z. Zhang, A. W. Appel, L. Beringer, J. Hatcliff, X. Ou, and A. Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *Principles of Security and Trust - First International Conference, POST 2012*, pages 369–389, 2012.

[2] T. Amtoft, J. Hatcliff, E. Rodríguez, Robby, J. Hoag, and D. Greve. Specification and checking of software contracts for conditional information flow. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, pages 229–245, 2008.

[3] N. Broberg and D. Sands. Flow locks: Towards a core calculus for dynamic flow policies. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006*, pages 180–196, 2006.

[4] N. Broberg and D. Sands. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010*, pages 431–444, 2010.

[5] S. Chong and A. C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS 2004*, pages 198–209, 2004.

[6] R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. Model checking information flow in reactive systems. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012*, pages 169–185, 2012.

[7] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[8] X. Li, F. Nielson, H. R. Nielson, and X. Feng. Disjunctive information flow for communicating processes. In *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Revised Selected Papers*, pages 95–111, 2015.

[9] L. Lourenço and L. Caires. Dependent information flow types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, pages 317–328, 2015.

[10] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011*, pages 218–232, 2011.

[11] K. K. Micinski, J. Fetter-Degges, J. Jeon, J. S. Foster, and M. R. Clarkson. Checking interaction-based declassification policies for android using symbolic execution. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security*, pages 520–538, 2015.

[12] T. Murray, R. Sison, E. Pierzchalski, and C. Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *29th IEEE Computer Security Foundations Symposium, CSF 2016*, 2016.

[13] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the Sixteenth ACM Symposium on Operating System*

Principles, SOSP 1997, pages 129–142, 1997.

[14] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *32nd IEEE Symposium on Security and Privacy, S&P 2011*, pages 165–179, 2011.

[15] H. R. Nielson, F. Nielson, and X. Li. Hoare logic for disjunctive information flow. In *Programming Languages with Applications to Biology and Security - Essays Dedicated to Pierpaolo Degano on the Occasion of His 65th Birthday*, pages 47–65, 2015.

[16] W. Rafnsson, D. Hedin, and A. Sabelfeld. Securing interactive programs. In *25th IEEE Computer Security Foundations Symposium, CSF 2012*, pages 293–307, 2012.

[17] A. Sabelfeld and H. Mantel. Securing communication in a concurrent language. In *Static Analysis, 9th International Symposium, SAS 2002*, pages 376–394, 2002.

[18] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[19] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Software Security - Theories and Systems, Second Mext-NSF-JSPS International Symposium, ISSS 2003*, pages 174–191, 2003.

[20] S. Tse and S. Zdancewic. Run-time principals in information-flow type systems. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.

[21] D. M. Volpano, G. Smith, and C. E. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.

[22] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 144–161, 1990.

[23] S. Zdancewic and A. C. Myers. Robust declassification. In *14th IEEE Computer Security Foundations Workshop (CSFW-14 2001)*, page 15, 2001.

[24] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007.

APPENDIX

A. DISTRIBUTED QUESTIONNAIRE WITH CLIENT PROCESSES

We consider a distributed questionnaire scenario where the answer-gathering process of Figure 1 (which will hereafter be referred to as the server process) communicates with the participants via client processes, rather than directly. The client process for user i, S^i_{cli}, is of the following simple-minded form.

$$\mathsf{recv}(c^i_{\text{in}}, dat_i);\ \mathsf{send}(c_i, dat_i);\ \mathsf{recv}(c'_i, res_i)$$

The process attempts to input from user i an answer. If the user is determined not to participate, the process simply blocks on this input. In case an answer is obtained, it is stored in the variable dat_i, and then sent to the server process via the channel c_i. The client then attempts to input from the server the statistics of the whole questionnaire, available after it is closed.

Formally, we have the concurrent system

$$S_{\text{qs}} \mid\mid S^0_{\text{cli}} \mid\mid \ldots \mid\mid S^{99}_{\text{cli}}.$$

Since the input data over each channel c_i in the server process is now sourced from the variable dat_i, and the channel c^i_{in}, we expect $\mathcal{P}^\bullet(dat_i)$ and $\mathcal{P}^\bullet(c^i_{\text{in}})$ to be identical to $\mathcal{P}^\bullet(c_i)$, which is the policy (1) of Section 2. Indeed, the system is well-typed under the type system of Section 8, with $\mathcal{P}^\bullet(c_i)$ and $\mathcal{P}^\bullet(dat_i)$ as mentioned above, $\mathcal{P}^\circ(c^i_{\text{in}}) = \star$, $\mathcal{P}^\bullet(res_i) = \{USR_i\}$, together with the policies already used when typing the sequential process S_{qs} in Section 5. The typing can be performed using the vector

$$\bigwedge_{0 \le j < 100} (par_j = 1 \Rightarrow \underline{par_j} = 1), \mathsf{true}, \ldots, \mathsf{true}$$

of prophetic formulas. The well-typedness of the concurrent system also means that it is future-dependent secure, because of Theorem 2. Note that in the typing of S^i_{cli}, the prophetic variables $\underline{par_j}$ in the policies of c^i_{in} and dat_i are regarded as arbitrary constants; hence no potential updates to them made in the server process need to be taken into consideration.