DOI 10.12694/scpe.v17i3.1179
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 3, pp. 171–187. http://www.scpe.org © 2016 SCPE

PRAVAH: PARAMETERISED INFORMATION FLOW CONTROL IN E-HEALTH[*]

CHANDRIKA BHARDWAJ AND SANJIVA PRASAD[†]

Abstract. We study the problem of enforcing information flow control (IFC) in eHealth systems. IFC mechanisms allow users to control the release and propagation of sensitive information so that confidential information is not observable to unintended principals while collaborating with other legitimate principals. We describe the methodology for modelling the information flow control requirements in a hospital domain using *Pravah*, a parameterised lattice-based IFC framework. The key advantage of using the parameterised security class lattice is greater precision in stating policies, enhanced usability and a reduced overhead in creating security tags. We can then use type-checking to statically verify that user programs do not violate stated security policies when accessing or manipulating data records. We discuss the main issues in designing the parameterised security class lattice.

Key words: Information flow control, Security, Lattice, Program verification, Parameterised security class.

AMS subject classifications. 03G10, 68N17, 68N18, 68P20, 68Q60

1. Introduction.

Consider a patient Puja who visits a doctor Divya in a hospital for a consultation. Nurse Neetu performs the preliminary medical examination, and uploads Puja's vital parameters (taken using various medical devices) to the hospital's Electronic Medical Record (EMR) system. Following this, Dr Divya attends to Puja and enters her diagnosis and notes in the EMR, along with instructions for the nurse. During such *healthcare encounters*[1], data are captured and stored in database tables of the hospital's EMR system, as shown in Table 1.1. Tables 1.3, 1.4 & 1.5 list the base types of a small set of the fields that are filled when a patient visits a doctor in the hospital. The data record related to an encounter is an evolving data structure, which is accessed and modified by different principals and spans the duration of the encounter and beyond.

Security of medical records is a major concern, not merely due to legislative requirements [1, 2, 3] but also since security breaches lead to inefficiencies in medical information systems [4]. Ensuring compliance with the safety, security and reliability requirements of healthcare information systems is best achieved by applying state-of-the-art formal techniques [5]. For instance, Pervez et al. use probabilistic model checking techniques to validate device interoperability in HL7-compliant standards [6, 7]. This paper proposes ideas from programming languages and systems, in particular security and type-checking frameworks, to ensure secure information management in healthcare information systems.

Access control mechanisms (ACMs) are the typical approach for ensuring security of electronic medical records. ACMs protect data by ensuring that they may be read or written only by authorised entities. The focus of ACMs is securing identification and managing identities, entitlements and privileges [8]. While sophisticated ACMs address most security requirements, it may still be possible for a trusted principal to either deliberately (or inadvertently) release or propagate sensitive data. For example, a doctor who has accessed his patients' information (using the function `viewAssigndEncounters` in Example 2) may thereafter accidentally upload these records including the patients' personally identifiable information onto a public server, thus violating patients' data confidentiality.

This problem may be addressed by following the seminal *information flow control* framework (IFC) [9, 10]. IFC extends access control by not only regulating who is allowed to access what data but also the subsequent use of the data accessed. Every data item is tagged with a *security class*. The security classes are organized in a *lattice* with respect to which *permitted information flows* are defined. The security policies define which flows of data from one security class to another are permitted and which are prohibited. By program/system analysis techniques such as type-checking [10, 11], it is possible to track and restrict the release of data as they flow through the system [10, 12]. This end-to-end approach is complementary to that of ACMs, which manages

[*]This work was supported by a research grant (2012-15) from DeitY, Ministry of Communication and Information Technology, Government of India: *Foundations of Scalable and Trustworthy Last-mile Healthcare*.

[†]Department of Computer Science & Engineering, Indian Institute of Technology Delhi, India 110016, (`chandrika@cse.iitd.ac.in, sanjiva@cse.iitd.ac.in`).

[1]We use the term "encounter" to denote the meeting of a patient with a medical care professional, which may have a wider connotation than a medical examination.

Table 1.1
EncounterRecords: Data stored in a typical encounter

Eid	Pid	Did	Date	Age	BP val	Diagnosis	Doc-Notes	Instructions For Nurse
91	1	5	02/01/15	33	140/90	Hyper-tension	Patient is hypochon-driac	Ensure AHT/ diuretic is taken in prescribed dose
92	2	6	04/01/15	28	80/65	Hypoten-sion	Pain in hands	-

Table 1.2
Classification of data using different security types

Field Name	Eid	Pid	Age	BP val	Diagnosis	Doc-Notes	Instruc-tions Fr Nurse
Base Type	int	int	int	int∗int	str	str	str
Simple Security Class	$\perp$	$\perp$	P	P	P	D	P
Dependent Security Class	$\perp$	$\perp$	$P(pid, \perp)$	$P(pid, eid)$	$P(\perp, eid)$	$D(did, pid, eid)$	$P(\top, eid)$

identities/roles and privileges at the point of accessing data.

Not all data entered in a single record belong to the same *security clearance* level. Therefore, we require appropriate IFC mechanisms to be adopted and integrated into the EMR system to prevent data belonging to a high security class (e.g. doctors' confidential notes) from flowing into any data objects belonging to a low security class (e.g., public files and servers). However, a simple security class lattice may be too coarse to adequately enforce the desired security policies in the eHealth domain.

The main contribution of this paper is describing a methodology for designing more precise information flow control mechanisms in a hospital domain. We present a parameterised security class lattice model, ordered by *permissible* flow relations [9], which helps in enforcing security policies typically required in a hospital. Using a simple example of such *lattices*, we illustrate how the parameterised model can capture the security requirements and permitted flows at a *fine-grained level.* The design of the security classes and the choice of parameters is not trivial, and the lattice has to be carefully designed to permit desired flows while precluding prohibited ones. Our parameterised framework, *Pravah*, is able to account for different relationships, roles and administrative structures obtained in a hospital. This paper extends our earlier work [13] where we used a prototypical hospital scenario to illustrate some of the principles involved in designing the lattice; in particular, we have refined the security class for doctors and have considered more complex lattice structures. We discuss how the lattice may be modified when introducing new classes of principals, to whom restricted or aggregated versions of the data records are to be released.

Subsequent to this introduction, we present in Sect. 2 a simple information flow control framework and motivate the need for parameterisation which supports more precise information flow control policies. In Sect. 3, using a simple example, we illustrate the parametric framework along with detailed explanations of the security lattice, and a discussion of some of the factors governing its design. In Sect. 4, we use the dependent-type IFC techniques of [11] to present some simple type-checked programs that guarantee compliance with the example policies. Sect. 5 presents the extensions to the system to include principals such as nurses and researchers. Finally, we conclude in Sect. 6, discussing related and future work.

2. Information Flow Control in Hospitals. Information flow analysis requires classifying information into different security classes, which are ordered in a pre-order. We start with a simple linear order $\perp < U < P < D < \top$, which expresses the following notion (Fig. 5.1). $\perp$ represents the most permissive class, i.e., public information; U represents information that is accessible to all registered users in a hospital EMR, and P represents all data which are patient-related. D represents the security class in which the information observable to only doctors belongs, while $\top$ is the most restrictive security class (which might even be data that no one can observe). Information is *allowed to flow* from a lower security class to a higher one, while any flow from a

TABLE 1.3
Users: Database table in hospital EMR

FieldName	**Base Type**	**Simple Sec.Class**	**Dependent Sec.Class**
uid	int	$\perp$	$\perp$
name	str	U	$U(uid)$
dob	str	U	$U(uid)$
gender	str	U	$U(uid)$
phone	int	U	$U(uid)$

TABLE 1.4
EncounterRecords: Database table in hospital EMR

Field Name	**Base Type**	**S.Sec. Class**	**Depndt Class**
eid	int	$\perp$	$\perp$
pid	int	$\perp$	$\perp$
nid	int	$\perp$	$\perp$
date	str	P	$P(pid, eid)$
gps	str	P	$P(pid, eid)$
prvsMI	int	P	$P(pid, \top)$
bp_val	int*int	P	$P(pid, eid)$
medkitid	int	P	$P(\perp, eid)$

TABLE 1.5
Diagnosis: Database table in hospital EMR

Field Name	**Base Type**	**S.Sec. Class**	**Depndt Class**
did	int	$\perp$	$\perp$
pid	int	$\perp$	$\perp$
eid	int	$\perp$	$\perp$
doc_conf	str	D	$D(did, pid, eid)$
prescriptn	str	P	$P(pid, eid)$
instruct-nurse	str	P	$P(\top, eid)$

higher to a lower class is a violation. These security classes allow categorisation of information captured in an encounter according to their confidentiality requirements (see "*simple security classes*" in Tables 1.2, 1.3, 1.4 & 1.5) and enable hospital administrators to specify and enforce security policies such as "*Doctors' confidential notes should not be observable to anyone except doctors*" or "*All patient-specific data captured in an encounter should not be leaked to a public server*".

However, this coarse security lattice does not allow one to specify policies that are precise enough to protect the confidentiality of information of one patient vis-a-vis another. For example, any patient in the hospital who has at least the security level P is allowed to call the function `viewPatientEncounters` (Example 1)[2] for any patient id, thus violating other patients' data confidentiality. Similarly, any doctor in the hospital with security level D can call a function `viewAssigndEncounters` (Example 2) for any doctor id and can not only see clinical and personally identifiable information (PII) of any patient captured in an encounter but can also read any other doctor's confidential notes. The code fragments presented in this paper are written in an ML-like functional language with side effects.

Example 1 Function `viewPatientEncounters` retrieves the list of encounters related to a patient id, from collection `EncounterRecords`.

```
let viewPatientEncounters = λ (pid_a).
  foreach(x in !EncounterRecords) with y = {}
    do let enc = !x in
      if(enc.pid == pid_a) then enc::y else y
```

Example 2 Function `viewAssignedEncounters` retrieves the list of encounters (of various patients) assigned to a doctor id, by simulating a join between collections `EncounterRecords` and `Diagnosis`.

```
let viewAssigndEncounters = λ (uidd).
  (foreach(x in !Diagnosis) with res_x = {} do
    let docnote = !x
    in if(docnote.did == uidd) then
```

[2]Examples are written in a functional language with imperative features as specified in [11] since we type-check our examples using their software.

```
    (foreach (y in !EncounterRecords) with res_y = {} do
    let tuple_enc = !y
    in if(tuple_enc.pid == docnote.pid and tuple_enc.eid == docnote.eid)
  then tuple_enc::res_y else res_y )
  else res_x )
```

To prevent such confidentiality breaches, hospital administrators should specify and follow policies such as:

P1 a *registered user's* information is observable only to herself but not to other users;

P2 all *patient-specific* information (such as any diagnosis, her name, address, blood-pressure value etc.) is observable to only the patient concerned and all the doctors who are assigned to her; and

P3 a doctor's *confidential notes* are observable to only the doctor herself.

To express such precise policies and enforce them in programs which compute on the sensitive encounter data retrieved from hospital EMRs, we refine the security lattice using the idea of dependent types from [14, 11] and *statically type check the code* for policy violations. The dependent type framework permits security classes to be parameterised over data values encountered at run time. We can therefore formulate permissible flows that are indexed by the data values present in the data records.

Using dependent security types, the security class U is *refined* to $U(uid)$, by splitting the previous class U into n compartments (assuming n users). $U(1)$ now represents the security class of the user with $uid = 1$ and is incomparable with $U(i)$, $\forall i \neq 1$. Each parameter value serves as a selector for information related to that value. By incomparable, we mean that neither $U(j) \leq U(i)$ nor $U(i) \leq U(j)$ holds, $\forall i \neq j$. Apart from the *bona fide* parameter values, two additional fictional parameter values are introduced: $\perp$ and $\top$, where $U(\perp) < U(i)$ and $U(i) < U(\top)$ for any i, refining the original class U into a small lattice. When we don't care about the specific value of parameter, we use index $\perp$: $U(\perp)$ captures the idea of information accessible to any user. When we want to over-approximate on a parameter, we use $\top$: $U(\top)$ is used for e.g., aggregation of information from many (or all) users. Classes indexed with $\perp$ and $\top$ act as connectors in the flow lattice and usually facilitate checking of permissible flows. Note that if we use $\top$ to over-approximate any parameter in a parameterised security class then the flows from that class have to be carefully regulated when designing the lattice to prevent policy violations. While it is possible to develop a more elaborate and fine-grained parameterisation scheme (e.g., indexed with each subset of principals), the flatter and more succinct lattices presented here suffice for most scenarios and policies encountered in the eHealth domain.

Similarly, we index the security classes P and D with patient and doctor ids. Instead of a unary predicate, we choose to parameterise the security class P on both *pid* (patient's *id*) and *eid* (encounter-id), to reflect the functional relationship between a patient and an encounter [13]. In similar fashion, we parameterise the security class D on *pid*, *did* (doctor's *id*) and *eid* to reflect the functional relationship between a doctor, a patient and an encounter. As a result, we get parameterised security classes $P(pid, eid)$ and $D(did, pid, eid)$, where *pid* is patient's *id* and *did* is doctor's *id*. For example, $P(1, 91)$ represents the security class of information related to the patient with $pid = 1$ and the encounter with $eid = 91$ and $D(5, 1, 91)$ represents the security class of information which is related to both the doctor with $id = 5$ and the patient with $id = 1$ and has been captured in the encounter with $eid = 91$. Considering the *eid* while indexing the higher security classes has a twofold advantage: (1) it captures the functional dependency between a patient and an encounter (similarly, between a doctor, a patient and an encounter); (2) it also allows us to capture the joint (though not equal, especially in case of a hospital) readership on the data captured in an encounter, i.e., data $\in P(pid, eid)$ can flow into security class $D(did, pid, eid)$ because doctor (*did*) has been assigned the encounter *eid* of patient (*pid*). These security classes have been used to annotate the types of fields in the database tables of the EMR (see "*dependent security classes*" in Tables 1.2, 1.3, 1.4, & 1.5).

3. The Parameterised Security Class Lattice. To capture the hospital's confidentiality policies, we define a partial order on these new security classes, a minimal example of which is shown in Fig. 3.1. Information is permitted to flow from a lower security class to a higher security class only if a path exists between them in the lattice. Note that it is the policies which essentially determine the structure of the lattice. Observe that the refined classes are not connected in a simply stacked manner with the topmost element of a hitherto lower class connected to the bottom-most element of a hitherto higher class. Several flows are prohibited, in keeping with the policies. We explain the meaning of the security classes in the lattice below.

3.1. Parameterised Security Class $U(uid)$. This security class allows policy **P1** to be enforced in a hospital EMR.

- $U(uid)$ corresponds to the class of information which is observable to a *registered user* with $id = uid$. For example, if Puja's *uid* is 1 and Priti's *uid* is 2, then Puja's personal data such as her date of birth, medical history, etc. will be tagged with $U(1)$ and these data cannot flow into data objects that are bound to the class $U(2)$. Neither can the data belonging in class $U(1)$ flow into a data object bound to class $U(\perp)$ as they would then be observable by all the *registered users*, which is a violation of policy **P1**.
- $U(\perp)$ represents the security class of data that are observable by all registered users in the hospital, such as public phone numbers of the hospital, the names of specialist doctors, their working hours and consultation fees, etc. The data tagged with $U(\perp)$ class can flow into the data objects bound to higher security classes in the lattice, e.g., $U(1)$ or $U(2)$ (see Fig. 3.1).
- $U(\top)$ represents the security class of data belonging to more than one *registered user* of a hospital, e.g., a list of the names of all the users along with their dates of birth. So, such data should *not be allowed* to flow into lower classes such as $U(1)$, which is observable by patient Puja. Additionally, such data should not be allowed to flow into data objects bound to the $P(\perp, \perp)$ class, which would be observable to all the patients. Such data may only flow into the most restrictive class, i.e. $\top$, from where data cannot flow out.

3.2. Parameterised Security Class $P(pid, eid)$. This security class allows policy **P2** to be enforceable in a hospital EMR.

- $P(pid, eid)$ corresponds to the class of information which is captured in an encounter with $id = eid$ and is specific to a patient with $id = pid$. For instance, information such as Puja's blood-pressure value or the date when the encounter between Puja and Dr Divya took place will be tagged with security class $P(1, 91)$, where 91 is the id of the encounter. These data cannot flow into a data object bound to security class $P(2, 92)$ because data objects tagged with security class $P(2, 92)$ are observable to Priti, and such a flow will violate the policy **P2**. Thus, $P(1, 91) \not\leq P(2, 92)$ in the security lattice (see Fig. 3.1). Similarly, we disallow the flow of information from security class $P(1, 91)$ to the data objects bound to the security class $P(\perp, \perp)$, as those will be observable to all the patients. Security class $P(\perp, \perp)$ is lower than $P(1, 91)$ in the security lattice (see Fig. 3.1).
  The flow of the information from the security class $U(uid)$ to $P(pid, eid)$ is allowed for all *eid* if $uid = pid$ in the security lattice, i.e., $\forall uid, eid. U(uid) \leq P(uid, eid)$ ensures the policy **P2**.
- $P(\perp, \perp)$ represents the security class of information which is observable to all the patients in a hospital, such as the number for emergencies, the names of specialist doctors, etc. Such information can flow into data objects belonging to higher security classes, e.g., $P(1, \perp)$ or $P(\perp, 91)$ in the lattice (Fig. 3.1).
- $P(pid, \perp)$ represents information which is specific to a patient with $id = pid$ but is not specific to any encounter, e.g., Puja's personally identifiable information (PII). Information from this class can flow into data objects bound to $P(pid, eid)$ but *not* in $P(\perp, \perp)$ as that would then be observable to all patients, violating policies **P1** and **P2**.
- $P(\perp, eid)$ is the security class for information captured in an encounter with $id = eid$ but not specific to any patient, e.g., the id of the medical devices used to measure Puja's vital parameters, *uid* of the nurse who performs the medical examination of Puja, etc. Such information is captured in an encounter record for situations when the doctor wants to confirm the integrity or authenticity of the data that she will use for making decisions and is typically present in metadata received from the machines. Such data can flow into data objects belonging to security class $P(pid, eid)$ but not in $P(\perp, \perp)$, for reasons mentioned above.
- $P(\perp, \top)$ is the security class for information which does *not* belong to any specific patient but is derived from more than one encounter, e.g., the list of identities of medical devices used in a laboratory, etc. In the lattice, $\forall eid, P(\perp, eid) \leq P(\perp, \top)$.
- $P(\top, \perp)$ is the security class for information which has been derived from information belonging to more than one patient but has *not* been captured in any specific encounter. An example of such information

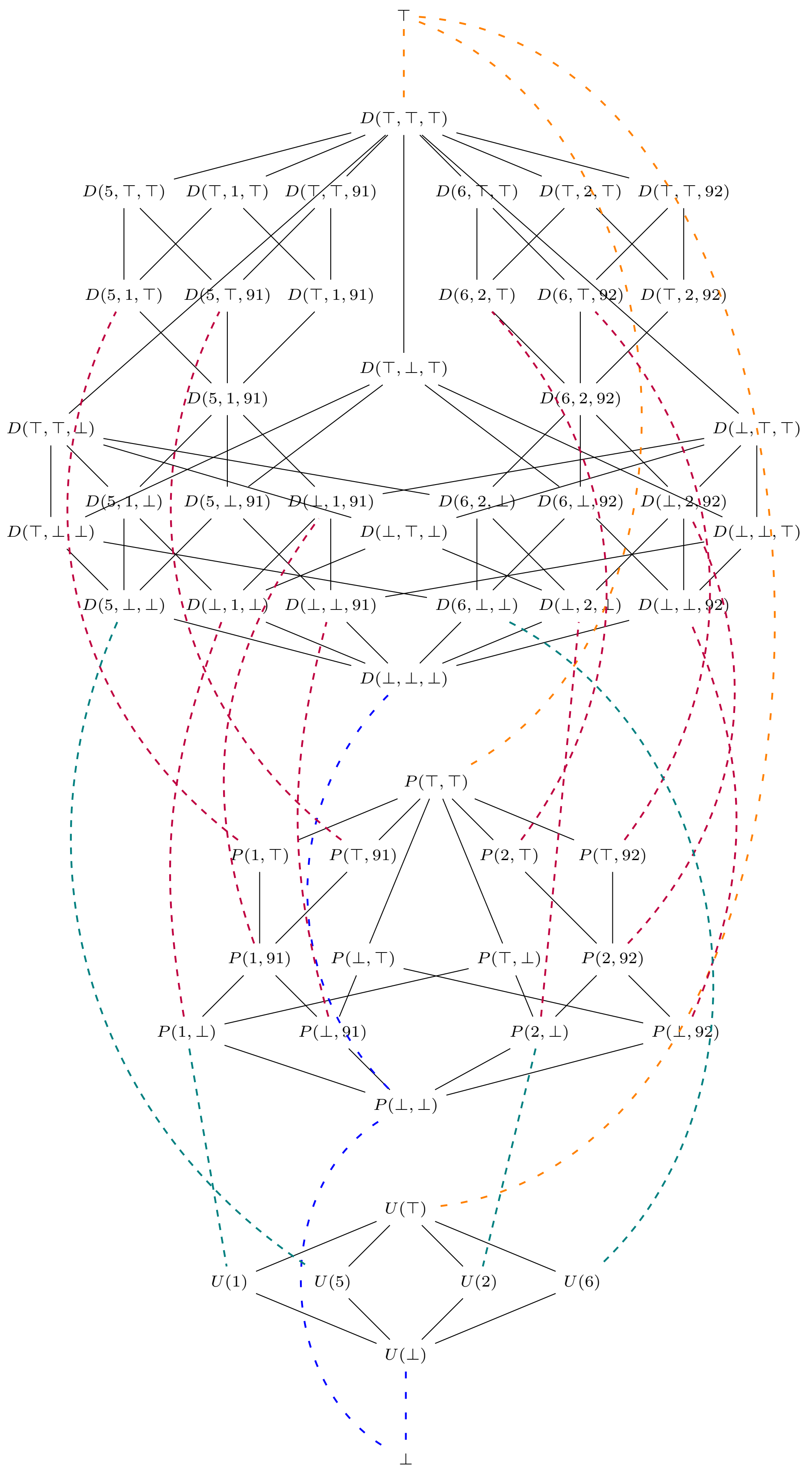

FIG. 3.1. *Minimum example of parameterised security class lattice for hospital domain.*

can be the list of diabetes patients treated in the hospital. In the lattice, $\forall pid, P(pid, \bot) \leq P(\top, \bot)$.

- $P(\top, eid)$ is the security class for the information which has been captured in an encounter with $id = eid$ but may belong to more than one patient. This is a special class, as it is not likely for more than one patient to be involved in the same encounter. As the information from security class $P(pid, eid)$ can flow into the data objects bound to this class, for any *pid*, this class is more restrictive than $P(pid, eid)$. It is interesting to note that the flow of information from security class $P(\top, \bot)$ to $P(\top, 91)$ is not allowed in the security lattice (Fig. 3.1), which may seem non-intuitive. But if one chooses to allow such a flow, it would enable the flow of information from the security class $P(2, \bot)$ to the security class $D(5, \top, 91)$ and violate the policy **P2**, as the data objects bound to the security class $D(5, \top, 91)$ are observable to the doctors who are not assigned to Priti ($pid = 2$). Therefore, in the security class lattice $\forall eid, P(\top, \bot) \nleq P(\top, eid)$. Similarly, $\forall pid, P(\bot, \top) \nleq P(pid, \top)$.
- $P(pid, \top)$ is the security class for information which is derived from data of more than one encounter and belongs to a patient with $id = pid$. This security class is more restrictive than $P(pid, eid)$. For instance, the complete medical profile of Puja can be tagged with security class $P(1, \top)$, which requires higher security than data from a single encounter.
- $P(\top, \top)$ is the security class for the information which consists of data belonging to more than one patient and has been derived from more than one encounter. An example of such information is a list of all the patients treated for a particular disease. Since data objects in this class have information obtained from multiple patient records, therefore, it can only flow to the most restrictive class, i.e., $\top$, from where data is not permitted to flow out.

3.3. Parameterised Security Class $D(did, pid, eid)$. This security class allows policy **P3** to be enforceable in a hospital EMR.

- $D(did, pid, eid)$ corresponds to the class of information captured in an encounter (with $id = eid$) of a patient with $id = pid$ and is specific to a doctor with $id = did$. For instance, information such as Dr Divya's confidential notes about Puja's diagnosis or her mental condition will be tagged with security class $D(5, 1, 91)$, where Dr Divya's *uid* is 5, Puja's *uid* is 1 and 91 is the id of the related encounter. These data cannot flow into a data object bound to security class $D(6, 2, 92)$ as it would then be observable to Dr Dipti ($uid = 6$), and such a flow will violate the policy **P2** and **P3**. Thus, $D(5, 1, 91) \nleq D(6, 2, 92)$ in the security lattice (see Fig. 3.1). These data cannot even flow into a data object bound to security class $D(7, 1, 97)$ as Puja may have visited Dr. Devi (uid=7) only once for cosmetic treatment and such a flow will violate the policy **P3**. Thus, $D(5, 1, 91) \nleq D(7, 1, 97)$ and $D(5, 1, 91) \nleq D(7, 1, 91)$ in the security lattice. Similarly, we disallow the flow of information from security class $D(5, 1, 91)$ to the data objects bound to the security class $D(\bot, \bot, \bot)$, as those will be observable to all doctors.
  Note that following hold in the security lattice:
  $\forall pid, did, eid. P(pid, eid) \leq D(did, pid, eid)$. Such a relationship is required to allow the flow of patient-specific information from the security class $P(pid, eid)$ to data objects bound to security class $D(did, pid, eid)$ to ensure the policy **P2**, as the data objects bound to the security class $D(did, pid, eid)$ are observable to the doctor with $id = did$ who is assigned to the patient and needs to see the information captured in an encounter to make clinical decisions.
- $D(\bot, \bot, \bot)$ represents the security class of information which is observable to all the doctors in a hospital, such as the checklist for a surgery etc. Such information can flow into data objects belonging to higher security classes, e.g., $D(5, \bot, \bot)$ or $D(\bot, 1, \bot)$ or $D(\bot, \bot, 91)$ in the lattice (Fig. 3.1).
- $D(did, \bot, \bot)$ represents information which is specific to a doctor with $id = did$ but is not specific to any encounter, e.g., Divya's personal notes. Information from this class can flow into data objects bound to $D(did, \bot, eid)$ and $D(did, pid, \bot)$ but *not* in $D(\bot, \bot, \bot)$ as that class is observable by all doctors and such a flow will violate policy **P3**.
- $D(\bot, pid, eid)$ is the security class for information that is specific to a patient (*pid*) and is captured in an encounter with $id = eid$ but is not specific to a particular doctor, e.g., Puja's blood pressure value, the id of the medical devices used to measure Puja's vital parameters, etc. Such data can flow into data objects belonging to security class $D(did, pid, eid)$, where *did* is uid of the doctor who has been

assigned this encounter but not into $D(\bot,\bot,\bot)$, for reasons mentioned above.

- $D(\bot,\top,\top)$ is the security class for information which does *not* belong to any specific doctor but is derived from more than one encounter, e.g., collection of encounter records of patients suffering from HIV. In the lattice, $\forall eid, D(\bot, pid, eid) \leq D(\bot,\top,\top)$.
- $D(\top,\bot,\bot)$ is the security class for information which has been derived from information belonging to more than one doctor but has *not* been captured in any encounter in the hospital, e.g., a discussion amongst doctors which may be about modifying existing checklists and protocols in a hospital. In the lattice, $\forall did, D(did,\bot,\bot) \leq D(\top,\bot,\bot)$.
- $D(\top, pid, eid)$ is the security class for the information which is specific to a patient and has been captured in an encounter with $id = eid$ but may belong to more than one doctor, e.g., the list of opinions of different doctors assigned to a patient. As the information from security class $D(did, pid, eid)$ can flow into the data objects bound to this class, for any did who has been assigned this encounter, this class is more restrictive than $D(did, pid, eid)$. Note that the flow of the information from the security class $D(\top,\bot,\bot)$ to $D(\top, 2, 92)$ is not allowed in the security lattice (Fig. 3.1), which may seem non-intuitive. But if one chooses to allow such a flow, then it would enable the flow of information from the security class $D(5,\bot,\bot)$ to the security class $D(\top, 2, 92)$ which violates the policy **P1** and **P3**, as the data objects bound to the security class $D(\top, 2, 92)$ are observable to the doctors with $uid \neq 5$, i.e., information specific to Dr Divya such as her salary, address and confidential notes will become observable to Dr Dipti. Therefore, in the security class lattice $\forall pid, eid, D(\top,\bot,\bot) \nleq D(\top, pid, eid)$. Similarly, $\forall did, eid, D(\bot,\top,\bot) \nleq D(did,\top, eid)$ and $\forall did, pid, D(\bot,\bot,\top) \nleq D(did, pid,\top)$.
- $D(did, pid,\top)$ is the security class for the information which is derived from the data of more than one encounter of patient (pid) and belongs to a doctor with $id = did$. This security class is more restrictive than $D(did, pid, eid)$. For instance, Divya's notes on the complete medical profile of Puja can be tagged with security class $D(5, 1,\top)$, which requires more security than the notes on data from a single encounter.
- $D(\top,\top,\top)$ is the security class for the information which consists of data belonging to more than one doctor and has been derived from more than one encounter of different patients. Examples of such information can be statistical reports or subjective analyses of patients' diagnosis (& treatment) or doctors' clinical performance, which lead to modifications in the hospital's private protocols. Such information requires a very high security classification in a hospital EMR. Such data cannot flow into any lower security classes as they may be observable to unintended readers. Therefore, it can flow to only the most restrictive class, i.e., $\top$.

TABLE 3.1
Inter-role information flow relations defined for e-Health.

1.	$\forall x, U(x) \rightarrow P(x, \_)$
2.	$\forall x, U(x) \rightarrow D(x, \_, \_)$
3.	$\forall pid, P(pid,\bot) \rightarrow D(\_, pid,\bot)$
4.	$\forall eid, P(\bot, eid) \rightarrow D(\_,\bot, eid)$
5.	$\forall pid, \forall eid, P(pid, eid) \rightarrow D(\_, pid, eid)$
6.	$\forall pid, \forall eid, P(pid,\top) \rightarrow D(\_, pid,\top)$
7.	$\forall pid, \forall eid, P(\top, eid) \rightarrow D(\_,\top, eid)$

4. Typechecked Programs. Our goal is to statically ensure (by typing) the confidentiality of information stored in an EMR. The lattice (Fig. 3.1) described above gives insights into how one can approach *type-checking* the information with security classes in an EMR in a fine-grained manner. In this section we show, with a series of code snippets, how to statically check and enforce the policies (**P1, P2 & P3**) in programs that operate on the data stored in a hospital's EMR, using the parameterised security classes. We use the type-checking framework of [11] to validate the programs. *The dependent types of the programs can be seen as formal proofs of the enforcement of the policies.*

Consider the *dependent security* type annotations in Tables 1.3, 1.4, & 1.5 and the following code snippets annotated with dependent security types:

Example 3 Retrieving encounters of patient with $pid = 2$

```
let viewPatientEncounters = λ(pida: int^⊥) =>
 [ret_type](foreach(x in !EncounterRecords) with y = {}:ret_type do
  let tuple = !x in
  if(tuple.pid == pida) then tuple::y else y)
in let n = 2 in (viewPatientEncounters(n))
```

In Example 3, `EncounterRecords` is a (mutable) collection of references (Table 1.4) and `y` gets the following security type since we are retrieving records with *pid* value 2: $\sum[pid\colon \bot, eid\colon \bot, nid\colon \bot, date\colon P(2, eid), gps\colon P(2, eid), prvsMI\colon P(2, \top), bp\_val\colon P(2, eid), medkitid\colon P(\bot, eid)]\hat{}\bot$. The presence of security type $P(2, eid)$ ensures that information such as the blood pressure value cannot flow into security classes accessible to principals prohibited by policy **P2**.

Example 4 Retrieving encounters assigned to a doctor.

```
let viewAssignedEncounters = λ(uidd:int^⊥).
 foreach(x in !Diagnosis) with res_x = {} do
  let tuple_doc = !x in if(tuple_doc.did == uidd)
   then foreach(y in !EncounterRecords) with res_y = {} do
    let tuple_enc = !y in
    if(tuple_enc.eid == tuple_doc.eid and tuple_enc.pid == tuple_doc.pid)
    then tuple_enc::res_y else res_y
  else res_x
in let f=first(viewAssignedEncounters(5)) in f
```

In Example 4, function `viewAssigndEncounters()` is used to retrieve the encounters which have been assigned to doctor with $did = 5$ by creating a join between `EncounterRecords` & `Diagnosis` and `f` gets security type $\sum[pid\colon \bot, eid\colon \bot, nid\colon \bot, date\colon P(pid, eid), gps\colon P(pid, eid), prvsMI\colon P(pid, \top), bp\_val\colon P(pid, eid), medkitid\colon P(\bot, eid)]\hat{}\bot$. This list can be accessed only by the assigned doctor because of the permissible flows in the lattice between these patients' parameterised security classes and the doctor's security class.

Example 5 Updating an existing encounter record.

```
let t=first((foreach(x in !EncounterRecords)
 with y={} do let t_enc = !x in
 if(t_enc.pid == 2 and t_enc.eid == 92) then t_enc.bp_val::y else y))
in foreach(x in !EncounterRecords) with _ do
 let t_enc = !x in
  if(t_enc.pid == 2 and t_enc.eid == 92) then let new_rec = [pid=t_enc.pid, eid=t_enc.eid,
    bp_val=t+".00",...]
 in x := new_rec
```

In Example 5, `y` gets security type $P(2, 92)$ since we are retrieving a record with $uid = 2$ and $eid = 92$. To type the record initializing the reference `new_rec`, we need to obtain the type $[pid\colon \bot \times eid\colon \bot \times bp\_val\colon P(2, 92) \times \ldots]$. But since we know `t` has security level $P(2, 92)$ and $t\_enc.pid = 2$ & $t\_enc.eid = 92$, the assignment $x := new\_rec$ can be deemed secure.

If we change the last conditional to be `if` $t\_enc.pid = 1$, then we would be trying to associate data of security type $P(2, 92)$, value `t`, with the security type $P(1, 92)$ meant for patient with *pid* 1. Such a flow of information is illegal and hence will not pass type-checking.

Example 6 Function `addFeedbackEncounter` allows a doctor to add her notes/feedback in the `Diagnosis` table.

```
let addFeedbackEncounter = λ(uid_d:int^⊥, pid_d:int^⊥, eid_d:int^⊥).
 foreach(p in viewAssignedEncounters(uid_d)) with _ do
```

```
  if(p.eid == eid_d and p.pid == pid_d) then
  foreach(y in !Diagnosis) with _ do
  let tfeed = !y in
   if(tfeed.eid == p.eid and tfeed.pid == p.pid) then
   let up_rec = [did=tfeed.did,pid=tfeed.pid,eid=tfeed.eid, doc_conf= feedback(p.pid, p.eid, p),...]
   in y := up_rec
in addFeedbackEncounter
```

In Example 6, the types ensure that only the doctor who has been assigned a particular encounter can see the doctor's notes. Function `viewAssignedEncounters` has type $(\prod(uid_d)\colon \bot).F$, where type F is $\sum[pid\colon \bot, eid\colon \bot, nid\colon \bot, date\colon P(pid, eid), gps\colon P(pid, eid), prvsMI\colon P(pid, \top), bp\_val\colon P(pid, eid), medkitid\colon P(\bot, eid)]^*$ (cf. Example 4). As there is no dependency we can refine the function type to be $Int^{\bot} \to F$ and `p` will have type F. Function `feedback` returns the doctor's personal notes on a particular encounter and has type $\prod u : \bot.\prod e : \bot.\prod r : F.P(u, e)$. So its return type in the call `feedback(p.pid, p.eid, p)` has type $P(p.pid, p.eid)$. Now, to declare the assignment `y:=up_rec` safe, we need to check if `up_rec` has the same type as the type specified for elements of collection `Diagnosis`. So we will need to check if `feedback(p.pid, p.eid, p)` has type $D(tfeed.did, p.pid, p.eid)$.

We know that $p.pid = tfeed.pid$ and $p.eid = tfeed.eid$ and the following holds in the security class lattice (Fig. 3.1): $\forall eid, P(pid, eid) \leq D(\bot, pid, eid)$ and $\forall eid, D(\bot, pid, eid) \leq D(did, pid, eid)$. We can therefore up-classify `feedback(p.pid, p.eid, p)` to security class $D(tfeed.did, p.pid, p.eid)$ where $tfeed.did$ represents the doctor to whom the encounter has been assigned.

Without dependent security types, we would not have fine-grained control over typing and have type $P(\top, \top)$ for `feedback(p.pid, p.eid, p)` due to which
`addFeedbackEncounter` will not type-check in spite of being secure.

5. Extensions to the Lattice. The total order (Fig. 5.1) described in Sect. 2 was minimal in that it considered a very simple ordering of security classes, and only a small class of roles (i.e., doctor and patient) in a hospital encounter. In practice, there are several other kinds of principals involved in a super-speciality hospital. We illustrate the issues encountered when extending the security lattice in different ways.

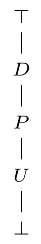

FIG. 5.1. *Total Order for Hospital Domain.*

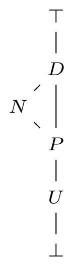

FIG. 5.2. *PreOrder involving Nurses.*

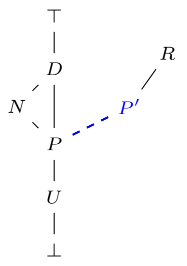

FIG. 5.3. *PreOrder with Nurses and Researchers.*

5.1. Nurses. Nurses play an important healthcare role, especially for in-patients. While most medical data being monitored are accessible to the nurses, a patient may not wish her insurance and payment history [1] to be accessible to them. Likewise, nurses may not want to share their personal information with other nurses, patients, or even doctors. Thus, another security class, i.e., N is added (Fig. 5.2) in the ordering considered in Sect. 2. Security class N represents all the data which are nurse-related.

This security class enables hospital administrators to specify and enforce security policies such as "*Nurses' confidential notes are not observable to patients*" or "*Nurses' personally identifiable information (PII) is not accessible to patients*", as information is not allowed to flow from a higher security class to a lower security class.
Both $P \leq N$ and $P \leq D$ hold in the preorder (Fig. 5.2), due to which all patient related data can flow into security class N. Any nurse in the hospital who has security level N would be able to call a function

TABLE 5.1
Nurse: Database table in hospital EMR

Field Name	Base Type	Sec. Class	Depndt Class
nid	int	$\perp$	$\perp$
qualificatn	str	N	$N(nid, \perp)$
specialisatn	str	N	$N(nid, \perp)$
salary	int	N	$N(nid, \top)$
address	str	N	$N(nid, \top)$
p_review	str	N	$N(nid, \top)$

TABLE 5.2
EncounterRecords: Database table in hospital EMR

Field Name	Base Type	Sec. Class	Depndt Class
eid	int	$\perp$	$\perp$
pid	int	$\perp$	$\perp$
nid	int	$\perp$	$\perp$
date	str	P	$P(pid, eid)$
gps	str	P	$P(pid, eid)$
prvsMI	int	P	$P(pid, \top)$
prvsMTP	int	P	$P(pid, \top)$
bp_val	int*int	P	$P(pid, eid)$
medkitid	int	P	$P(\perp, eid)$
observatn	str	N	$N(nid, eid)$

`viewAssignedPatients` (Example 7) for any nurse id and can see sensitive information of all the patients, irrespective of the date and time when the patient visited the hospital. As argued earlier, unparameterised security classes do not allow one to specify policies that are precise enough to protect the confidentiality of information. Without parameterisation, the following policies cannot easily be enforced: "*Sensitive information specific to a nurse such as her home address, salary, etc. is not observable by any other nurse or doctor*" or "*A patient's encounter specific information is observable by* only *those nurses who have been assigned the encounter with that patient.*".

To prevent such confidentiality breaches, hospital administrators should specify and follow policies such as:

P2' all *encounter-specific* information of a patient (such as temperature, blood-pressure value, pulse-rate, SpO_2 value, etc.) is observable to only the patient concerned and all the nurses and doctors who have been assigned the encounter with the patient; and

P4 a patient's medical history beyond an encounter is observable to *only* the patient concerned and all the doctors (*not nurses*) who are assigned to her;

P5 a nurse's sensitive information and her confidential notes are observable to her alone.

To express these precise policies and enforce them in programs which compute on the sensitive data retrieved from the hospital EMR, we refine the security class N using both nid and eid, to reflect the functional relationship between a nurse and an encounter. As a result, we get parameterised security class $N(nid, eid)$, where nid is the nurse's id and eid the encounter id. For example, $N(3, 91)$ represents the security class of information related to the nurse with $nid = 3$ and the encounter with $eid = 91$. The parameterised security class $N(nid, eid)$ allows policy **P2'**, **P4** and **P5** to be enforceable in the hospital EMR (Fig. 5.4).

5.1.1. Typechecked Programs.

We illustrate how parameterised security class $N(nid, eid)$ can enable hospital administrators to enforce security policies **P2'**, **P4**, **P5**.

Example 7 Look up patients' information assigned to a Nurse

```
let viewAssignedPatients = λ(nid_d:int^⊥).
 foreach(x in !EncounterRecords) with res_x={} do
  let t_enc = !x
  in if(t_enc.nid == nid_d) then
     foreach (y in !Users) with res_y={} do
     let t_usr = !y
     in if(t_usr.uid == t_enc.pid) then t_usr::res_y else res_y
    else res_x
in let f = first(viewAssignedPatients(3)) in f
```

In Example 7, function `viewAssignedPatients()` is used to retrieve user-specific details (name, age, gender, etc.) about the patients whose encounter has been assigned to nurse Neetu ($nid = 3$). Here, hospital administrators can enforce policy **P2'** using parameterised security class $N(nid, eid)$, by specifying the following permissible information flow in the lattice: $P(\top, eid) \rightarrow N(\perp, eid)$ (Fig. 5.4). As a result, a nurse can retrieve

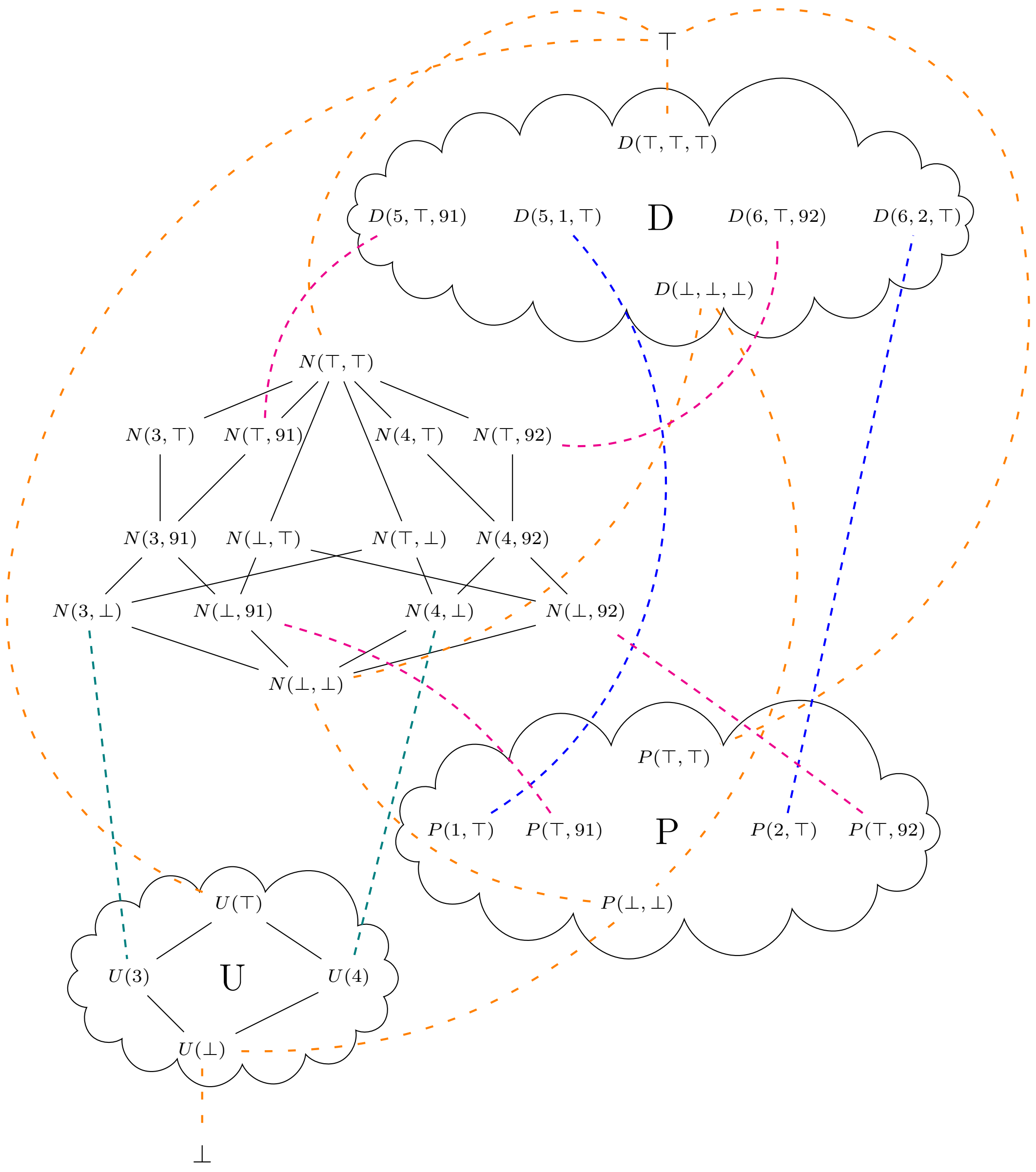

Fig. 5.4. *Example parameterised security class for Nurses.*

the user-specific information of *only* those patients who have been assigned to her by the hospital.

Example 8 Look up performance review specific to a Nurse.

```
let viewReview = λ(uida:int^⊥).
  foreach(x in !Employee) with y = {} do
    let tuple = !x in
      if(tuple.nid == uida) then
        [nid = tuple.nid, p_review = tuple.p_review] :: y
      else y
```

```
in let n = 3 in (viewReview(n))
```

In Example 8, function `viewReview()` is used to view the confidential performance review of nurse Neetu ($nid = 3$). Policy **P5** is enforceable in Examples 8 & 9, if sensitive information of nurses, such as their salary and performance review, is classified in security class $N(nid, \top)$. The return type of function `viewReview()` in Example 8 is $\Sigma[nid : \bot, p\_review : N(3, \top)]\hat{}\bot$ which ensures that the information from performance review of nurse Neetu ($nid = 3$) cannot flow to security classes whose data is observable to other nurses with $nid \neq 3$ or even to any doctor.

Example 9 Look up a salary statement specific to a Nurse.

```
let viewSalary = λ(uida:int^⊥).
  foreach(x in !Employee) with y = {} do
     let tuple = !x in
        if(tuple.nid == uida) then
          [nid = tuple.nid, sal = tuple.sal] :: y
        else y
in let n = 3 in (viewSalary(n))
```

In Example 9, function `viewSalary()` is used to view the salary of nurse Neetu ($nid = 3$). If hospital administrators classify salary in security class $N(nid, \top)$, then the return type of function `viewSalary()` is $\Sigma[nid : \bot, sal : N(3, \top)]\hat{}\bot$ which ensures that no nurse apart from Neetu ($nid = 3$) can access the output of function call `viewSalary(3)`.

Example 10 Look up partial medical history of a Patient.

```
let viewMTPHis = λ(pida:int^⊥).
  foreach (x in !EncounterRecords) with y = {} do
    let t_enc = !x in
      if (t_enc.pid == pida) then
        [pid = t_enc.pid, prvsMTP = t_enc.prvsMTP] :: y
      else y
in let n = 1 in (viewMTPHis(n))
```

In Example 10, function `viewMTPHis()` is used to retrieve the history of MTPs (abortions) of a patient. Patients usually do not want to share such incidents with anyone other than their doctors. Hospital administrators can enforce such security requirements (policy **P4**) by putting such sensitive medical history data in security class $P(pid, \top)$, as these are inaccessible to nurses (Fig. 5.4). The flow $\forall pid, P(pid, \top) \longrightarrow N(\_, eid)$ is *not* allowed in the preorder specifying permissible flows (Fig. 5.4). As a result, the output of function call `viewMTPHis(1)` has type $\Sigma[pid : \bot, prvsMTP : P(1, \top)]\hat{}\bot$, making it inaccessible to any nurse.

Example 11 Look up complete medical history specific to a Patient.

```
let viewCompleteMedHis = λ(pida:int^⊥).
  foreach(x in !EncounterRecords) with y = {} do
     let tuple = !x in
    if(tuple.pid == pida) then
      tuple::y
    else y
in let n = 1 in (viewCompleteMedHis(n))
```

In Example 11, function `viewCompleteMedHis()` is used to retrieve the complete medical history of a patient present in the system. Typically, patients prefer not to share their *complete* medical history with all doctors they meet for consultation and only trust family doctors with their complete history. Hospital administrators need to be able to enforce security policies **P2 & P4** to prevent such information leakage. With appropriate classification of data (see Table 5.2), the return type of function call `viewCompleteMedHis(1)` becomes: $\Sigma[pid : \bot, eid :$

$\perp, nid : \perp, date : P(1, eid), prvsMI : P(1, \top), prvsMTP : P(1, \top), bp\_val : P(1, eid), medkitid : P(\perp, eid)]\hat{}\perp$ which has the patient's medical history including previous Medical Termination of Pregnancy details, classified in security class $P(1, \top)$, making it unobservable to nurses (as explained in the previous example) or any doctor who has not been allowed to observe all encounters of patient Puja ($pid = 1$). Examples 10 & 11 illustrate how parameterised security classes enable hospital administrators to enforce security policy **P4**.

5.2. Researchers. Medicine goes beyond merely treating patients. Medical professionals including doctors, health administrators and researchers require access to large corpora of *bona fide* medical cases for analyses that may help provide better treatments, avoid epidemics, and otherwise improve the delivery of healthcare. For research, authentic details about diagnosis, treatment and prognosis of the patients are also required. But such data sharing between hospitals and researchers would violate most of the security policies discussed till now, i.e., **P1**, **P2**, **P2'**, **P3**, **P4** and **P5**.

Typically, hospitals sign a confidentiality agreement with patients stating that their data will not be given out to third parties without their consent, which is usually taken prior to starting treatment. To protect patient confidentiality, hospitals release medical data for research purposes only after "de-identifying" (anonymising) the patient specific information in a new copy of the data [15]. However, it becomes a non-trivial task for hospital administrators to ensure that the researchers are given access to only certain kinds of information requested, e.g., "*only the medical information which has been captured during a particular time period is shared with a specific researcher*" or "*only the medical information which is related to patients diagnosed with a specific disease is shared with a researcher*". Currently, widely-used practices for this purpose primarily require manual intervention, which is typically error-prone and not sound.

To address this requirement, a security class R is added (Fig. 5.3), in which information observable to only researchers belongs. It is assumed that the hospital administrators will create a copy of patients' data and anonymise these to share with researchers. A trusted function is used to obfuscate all personally identifiable information from this new copy of the medical database. The dashed edge between security class P and security class P' represents that the data flowing from security class P to P' has been anonymised by the trusted function. This security class along with trusted anonymising functions enables hospital administrators to prevent violation of the security policies such as **P1**, **P3**, **P5**, etc.

Any researcher should be able to access all the anonymised information from the new copy of the hospital database. Additionally, a researcher would not like to necessarily share their observations and notes with other users (including other researchers) in the hospital domain. To prevent such confidentiality breaches, hospital administrators need to specify and enforce the following policies:

P6 a researcher can observe *only the non-personally identifiable* information from specific encounters which have been assigned to her,

P7 a researcher's confidential notes and observations are accessible to that researcher *alone.*

To express these precise policies and enforce them in programs which compute on the anonymised sensitive data retrieved from hospital EMRs, we refine the security class P' — which is the same as P except that it is on a different domain, i.e. $P(pid', eid')$, where we have trusted one-way function owf such that:

$$owf(pid) = pid' \text{ and } owf(eid) = eid'.$$

We parameterise the security class R on both rid (researcher's id) and eid' ($owf(eid)$). This gives us security class $R(rid, eid')$ which contains the data specific to some encounter and a researcher. For instance, $R(8, 8475)$ represents the security class of information concerning researcher with $rid = 8$ and the encounter with $eid' = 8475$.

The parameterised security classes $P(pid', eid')$ and $R(rid, eid')$ allow policies **P6** and **P7** to be enforceable in the hospital EMR. These classes allow researchers to access anonymised useful medical data without breaching confidentiality of patient data and disallow researchers from accessing anything below or above the parameterised patient classes.

5.2.1. Typechecked Programs. We illustrate how parameterised security classes $R(rid, eid')$ & $P(pid', eid')$ can enable hospital administrators to enforce security policies **P6** and **P7**.

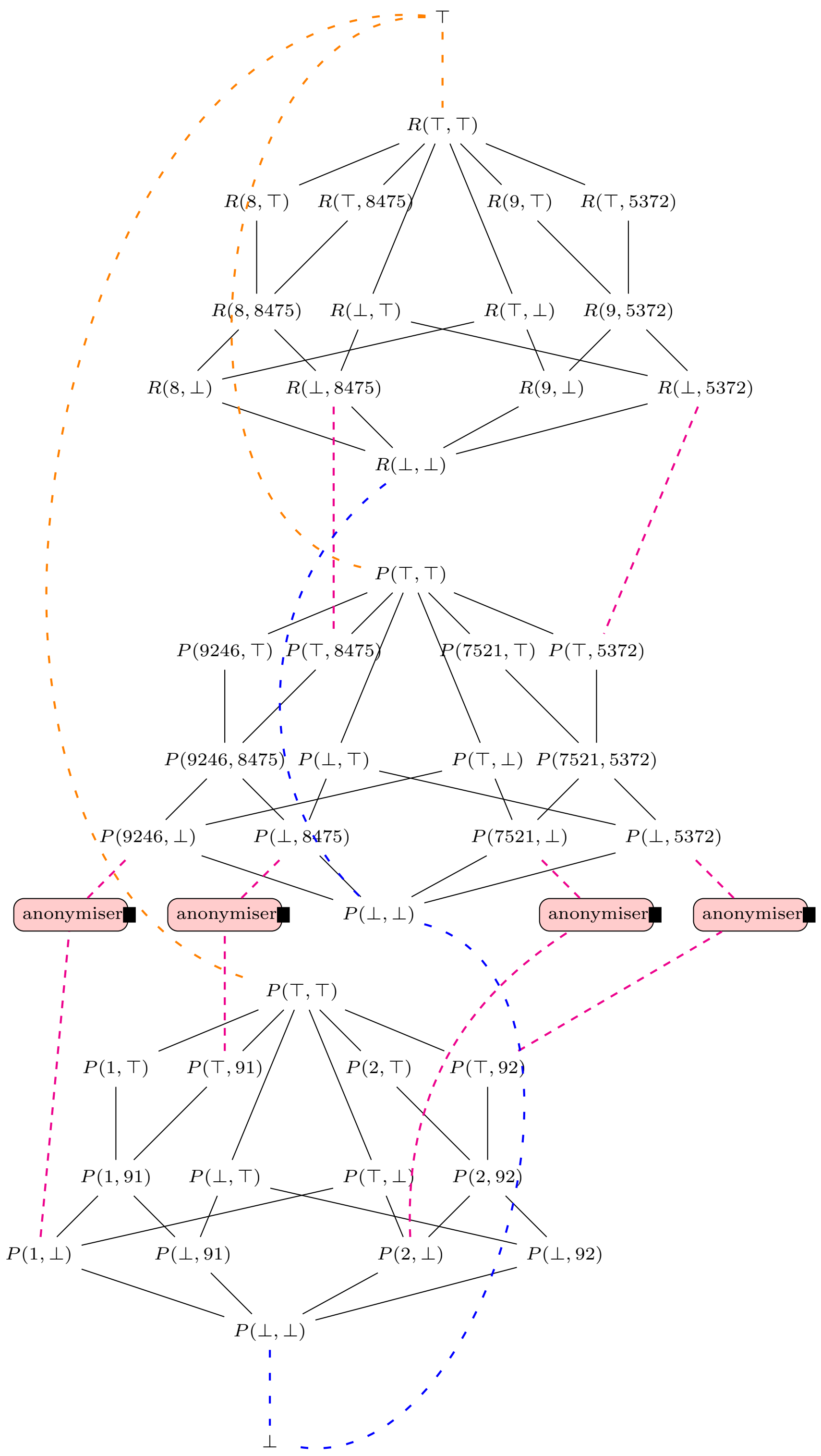

FIG. 5.5. *Example parameterised security classes for Researchers.*

TABLE 5.3
Researcher: Database table in hospital EMR

Field Name	Base Type	Sec. Class	Depndt Class
rid	int	$\bot$	$\bot$
qualificatn	str	R	$R(rid, \bot)$
specialisatn	str	R	$R(rid, \bot)$
salary	int	R	$R(rid, \top)$
address	str	R	$R(rid, \top)$

TABLE 5.4
ResearchRecords: Database table in hospital EMR

Field Name	Base Type	Sec. Class	Depndt Class
eid	int	$\bot$	$\bot$
rid	int	$\bot$	$\bot$
conf_notes	str	R	$R(rid, eid)$
results	str	R	$R(\bot, eid)$

Example 12 Look up data from encounters assigned to Researcher.

```
let viewData = λ(uidr:int^⊥).
  foreach(x in ResearchRecords) with res_x = {}: enc do
    let tuple_r = !x
      in if(tuple_r.rid == uidr ) then
      foreach(y in EncounterRecords) with res_y = {}: enc do
      let tuple_e = !y
        in if(tuple_e.eid == tuple_r.eid) then tuple_e::res_y
           else res_y
     else res_x
in let r = first(viewData(8)) in r
```

In Example 12, function `viewData()` is defined to retrieve all patient-encounter data that are accessible to a researcher by simulating a join between the anonymised copy of collection `EncounterRecords` and the `ResearchRecords` collection. Policy **P6** is enforced in Example 12, as the return type of function call `viewData(8)` is $\Sigma[pid' : \bot, eid' : \bot, nid : \bot, date : P(pid', eid'), gps : P(pid', eid'), bp\_val : P(pid', eid'), medkitid : P(\bot, eid')]\hat{\ }\bot$ and the output is observable to researcher Richa ($rid = 8$) alone.

Example 13 Look up confidential notes of a Researcher.

```
let viewNotes = λ(uida:int^⊥).
  foreach(x in !ResearchRecords) with y = {} do
     let tuple = !x in
        if(tuple.rid == uida) then
          [rid = tuple.rid, conf_notes = tuple.conf_notes] :: y
        else y
in let n = 8 in (viewNotes(n))
```

In Example 13, function `viewNotes()` is used to view the confidential notes of researcher Richa ($rid = 8$). If hospital administrators classify the confidential notes in security class $R(rid, \top)$, then the return type of function `viewNotes()` is $\Sigma[rid : \bot, conf\_notes : R(8, \top)]\hat{\ }\bot$ which ensures that no researcher apart from Richa ($rid = 8$) can access the output of function call `viewNotes(8)`. This example illustrates how hospital administrators can enforce policy **P7** and ensure that sensitive information like these confidential notes cannot flow into data objects observable to unauthorised or unintended users in the system.

6. Conclusion. Medical data and metadata require far more sophisticated information management schemes than provided by *access control* mechanisms typically employed in hospital information systems. To ensure the *end-to-end* security/integrity of data, *information flow control* (IFC) mechanisms are required. Earlier work on IFC [9] has been extended in *decentralized* IFC (DIFC) to protect data for different users, each with their individual policy [16] and researchers have proposed an IFC-compliant database in [17]. Naive instantiations of such security lattice frameworks do not provide adequate protection or flexibility in specifying policies. Dependent-types provide a method to index information with specific users (or doctors) and thus provide far more precise and fine-grained control over information flow.

In this paper we have presented a security lattice, and discussed some of the subtleties in the design of permitted information flows. The examples are simple, but illustrate the principles involved in extending the classification to real HIS systems. In the future, we will explore how the dependent information flow types [11] can be adapted to allow for authorised declassification, along the lines of [16]. Finally, we intend to integrate the systematic tagging of data/metadata [18] into this dependent-type framework, to allow for secure information flow with high usability & minimal overhead in a federated collection of administrative domains where data from different domains are subject to different information flow policies [19].

REFERENCES

[1] U. D. of Health, H. Services, et al., *Summary of the HIPAA privacy rule*, Washington, DC: Department of Health and Human Services, (2003).

[2] M. A. Scholl, K. M. Stine, J. Hash, P. Bowen, L. A. Johnson, C. D. Smith, and D. I. Steinberg, *An introductory resource guide for implementing the health insurance portability and accountability act (HIPAA) security rule*, tech. report, United States, 2008.

[3] U. D. of Health & Human Services et al., *HITECH Act enforcement interim final rule*, US Department of, (2013).

[4] E. Marzini, P. Mori, S. D. Bona, D. Guerri, M. Lettere, and L. Ricci, *A tool for managing the X1.V1 platform on the cloud*, Scalable Computing: Practice and Experience, 16 (2015).

[5] A. Gawanmeh, H. M. N. A. Hamadi, M. Al-Qutayri, S. Chin, and K. Saleem, *Reliability analysis of healthcare information systems: State of the art and future directions*, in HealthCom, 2015, pp. 68–74.

[6] U. Pervez, A. Mahmood, O. Hasan, K. Latif, and A. Gawanmeh, *Formal reliability analysis of device interoperability middleware (DIM) based e-health system using PRISM*, in HealthCom, 2015, pp. 108–113.

[7] U. Pervez, O. Hasan, K. Latif, S. Tahar, A. Gawanmeh, and M. S. Hamdi, *Formal reliability analysis of a typical FHIR standard based e-health system using PRISM*, in Healthcom, 2014, pp. 43–48.

[8] M. Benantar, *Access Control Systems: Security, Identity Management and Trust Models*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[9] D. E. Denning, *A Lattice Model of Secure Information Flow*, Commun. ACM, 19 (1976), pp. 236–243.

[10] D. E. Denning and P. J. Denning, *Certification of Programs for Secure Information Flow*, Commun. ACM, 20 (1977), pp. 504–513.

[11] L. Lourenço and L. Caires, *Dependent Information Flow Types*, in POPL'15, ACM, 2015, pp. 317–328.

[12] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris, *Information Flow Control for Standard OS Abstractions*, in Proceedings of SOSP'07, ACM, 2007, pp. 321–334.

[13] C. Bhardwaj and S. Prasad, *Parametric Information Flow Control in eHealth*, in HealthCom, 2015, pp. 102–107.

[14] P. Martin-Löf, *Intuitionistic Type Theory: Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980*, 1984.

[15] I. C. of Medical Research, *Ethical guidelines for biomedical research on human participants*. `http://icmr.nic.in/ethical_guidelines.pdf`, 2006. Accessed: 2016-05-12.

[16] A. C. Myers and B. Liskov, *A Decentralized Model for Information Flow Control*, SIGOPS Oper. Syst. Rev., 31 (1997), pp. 129–142.

[17] D. A. Schultz and B. Liskov, *IFDB: Decentralized Information Flow Control for Databases*, in EuroSys '13, 2013, pp. 43–56.

[18] C. Bhardwaj, *Systematic Information Flow Control in mHealth Systems*, in Proceedings of Workshop on Networked Healthcare Systems (COMSNETS), 2015, NetHealth'15, IEEE, 2015, pp. 1–6.

[19] S. Prasad, *Designing for scalability and trustworthiness in mhealth systems*, in Proceedings of Distributed Computing and Internet Technology Conference, ICDCIT'15., LNCS 8956, Springer, 2015, pp. 114–133.

Edited by: Amjad Gawanmeh
Received: Feb 1, 2016
Accepted: Jul 3, 2016