Language-Based Mechanisms for Privacy-by-Design

Shukun Tokas, Olaf Owe, and Toktam Ramezanifarkhani

Department of Informatics, University of Oslo
{shukunt,olaf,toktamr}@ifi.uio.no

Abstract. The privacy by design principle has been applied in system engineering. In this paper, we follow this principle, by integrating necessary safeguards into the program system design. These safeguards are then used in the processing of personal information. In particular, we use a formal language-based approach with static analysis to enforce privacy requirements. To make a general solution, we consider a high-level modeling language for distributed service-oriented systems, building on the paradigm of active objects. The language is then extended to support specification of policies on program constructs and policy enforcement. For this we develop i) language constructs to formally specify privacy restrictions, thereby obtaining a policy definition language, ii) a formal notion of policy compliance, and iii) a type and effect system for enforcing and analyzing a program's compliance with the stated polices.

Keywords: Privacy by Design · Language-based Privacy · Privacy Compliance · Static Analysis

1 Introduction

Advances in information technologies have often led to concerns about privacy. With the adoption of information and communication technology in our daily lives, the gathering and processing of personal information fundamentally increases the potential for privacy threats. In particular, privacy and data protection features are often ignored by conventional engineering approaches [1] or accommodated as an afterthought. Aligning the software ecosystem with the privacy-related requirements is an essential step towards better data protection. In order to endorse privacy as a first-class requirement and promote privacy compliance from the outset of product development, the *privacy by design* (PbD) requirement has been formally embedded in the GDPR regulations (Article 25 [2]). Article 25 [2] obliges the controllers to design and develop products with a built-in ability to demonstrate compliance towards the data protection obligations.

The main idea of privacy by design is to make privacy a key consideration in development of systems. Privacy by design is a framework consisting of seven foundational principles: *i)* proactive not reactive; preventive not remedial, *ii)* privacy as default setting, *iii)* privacy embedded into design, *iv)* full functionality - positive-sum, not zero-sum, *v)* end-to-end security - full lifecycle protection, *vi)* visibility and transparency, and *vii)* respect for user privacy - keep it user-centric. We focus on the *privacy embedded-into-design* principle, due to its potential connection with language mechanisms. We explore the idea of adding privacy requirements into programming/specification languages and use static analysis for enforcing such privacy requirements.

In this paper, we follow the privacy by design principle, by integrating necessary safeguards into the processing of personal information, using a language-based approach. In particular, we explore how to formalize fundamental privacy principles and to provide built-in abilities to fulfill data protection obligations. As a step towards this goal we develop a policy specification language that provide constructs for specifying privacy requirements on sensitive (personal) data In particular, a policy is given by a set of triples that put restrictions on the *principals* that may access the information for certain *purposes* and the permitted *access rights*. Such policy statements are then linked with language constructs of a high-level modeling language oriented towards distributed and service-oriented systems. Policies are annotated with the *data types* and *methods*.

Certain aspects of privacy restrictions can be expressed by means of static concepts, while others can only be expressed at runtime, such as *data subject*, *consent*, and other user-defined changes. In this paper, we focus on statically declared policies and implicit consent (at compile time presence of policy implies consent). Changes in consent and policies are handled at runtime through predefined functionalities, which is beyond the scope of this article. In addition to *read* and *write* access, we consider *incremental* access (*incr*), allowing addition of sensitive information without read access and without modifying existing information. For instance, in a healthcare setting, a lab assistant may have incr access to treatment data, while a nurse may have both read and incremental access ($read \sqcup incr$), and a doctor may have full access ($read \sqcup write$). We formalize a notion of policy compliance, to develop a scheme of policy inheritance. Finally, to enforce policy compliance, we define a set of rules, i.e., the type and effect system that checks that the policies are respected when the sensitive information is accessed. The theory of the current work is presented in more details in [3].

In summary, the main idea is to provide language constructs that express privacy policy specifications capturing static aspects of privacy and use these to statically analyze a program's compliance with the policy specifications. We make the following contributions: i) propose a policy language for specifying purpose, access and policy requirements (see Figure 1), ii) formalize a notion of policy compliance, iii) show how the policy language can be used with an underlying object-oriented language, and iv) develop a mechanic type and effect system for analyzing a program's compliance with the annotated privacy policies.

Paper Outline. The rest of the paper is structured as follows. Section 2 presents the formalization of privacy policies, including a policy definition language and a formalization of policy compliance. Section 3 introduces the core language, with support for the specification of privacy principles. Section 4 presents the type and effect system. Section 5 demonstrates the analysis on a small case study. Section 6 discusses related work, and Section 7 concludes the paper.

A	$::=$	$read \mid incr \mid write \mid self$	basic access rights
		$\mid no \mid full \mid rincr \mid wincr$	abbreviated access rights
		$\mid A \sqcap A \mid A \sqcup A$	combined access rights
$\mathcal{P}$	$::=$	(I, R, A)	policy
$\mathcal{P}s$	$::=$	$\{\mathcal{P}^*\} \mid \mathcal{P}s \sqcap \mathcal{P}s \mid \mathcal{P}s \sqcup \mathcal{P}s$	policy set
$\mathcal{RD}$	$::=$	$\texttt{purpose}\ R^+$	
		$[\texttt{where}\ Rel\ [\texttt{and}\, Rel]^*]$	purpose declaration
Rel	$::=$	$R^+ < R^+$	sub-purpose declaration

Fig. 1. BNF syntax definition of the policy language. I ranges over interface names and R over purpose names. The operators $\sqcup$ and $\sqcap$ denote join and meet, respectively.

2 Language Constructs for Policy Specification

Privacy policies are often described in natural language statements. To verify formally that the program satisfies the privacy specification, the desired notions of privacy need to be expressed explicitly. To formalize such policies, we define a policy specification language. Furthermore, to establish a link between policies and programming language constructs, we extend the syntax and semantics of a small core language (see Section 3). In our setting, a privacy policy is a statement that expresses permitted use of the sensitive information by the declared program entities. To support privacy-by-design, we define policies at the design level, and associate policies to data types and methods of interfaces and classes, such that the policies of a method in a class must comply with the corresponding policy in an interface of the class. In particular, a policy is given by a set of triples that put restrictions on: What *principals* may access the sensitive data, which *purposes* are allowed, and which *access-rights* are permitted. That being the case, a policy $\mathcal{P}$ is given by a triple (I, R, A), where *i)* I ranges over interfaces, which are organized in an open-ended inheritance hierarchy, *ii)* R ranges over purposes, which are organized in a hierarchy (reflecting specialization), and *iii)* A ranges over access rights, which are organized in a lattice. Thus principals are expressed by the Interfaces, while new language constructs are added to represent purposes, access rights, and policies.

The language syntax for policies is summarized in Figure 1, where [ ] is used as meta-parenthesis, and superscripts * and $^+$ denote general and non-empty repetition, respectively. Here we briefly discuss the specification constructs.

Principal describes the roles that can access sensitive information and is given by an interface. For instance for a call $x := o.m(\overline{e})$, where o is typed by an interface with policy (I, R, A), the caller object must support interface I. Interfaces are organized in an open-ended inheritance hierarchy, letting $I < J$ denote that I is a subinterface of J. For example,

$$Specialist < Doctor < HealthWorker$$

Any is predefined as the least specialized interface, i.e., the superinterface of all interfaces. We let $\leq$ denote the transitive and reflexive extension of $<$.

Purpose names are used to restrict usage of sensitive data to specific purposes. Such purpose names can be organized in a hierarchical structure, reflecting a *purpose hierarchy* [4]. We let purposes be organized in a directed acyclic graph reflecting specialization. Purpose names are defined by the keyword **`purpose`**. For instance, the declaration

$$\texttt{\textbf{purpose}}\ \mathit{spl\_treatm}, \mathit{treatm}\ \texttt{\textbf{where}}\ \mathit{spl\_treatm} < \mathit{treatm}$$

makes *spl_treatm* more specialized purpose than *treatm*. If data is collected for the purpose of *spl_treatm* then it cannot be used for *treatm*. However, if it is collected for the purpose of *treatm* then it can be used for *spl_treatm*. We let $\leq$ denote the transitive and reflexive extension of $<$.

Access-right describes permitted operations on sensitive data. Access rights are given by a lattice, with meet and join operations (see Figure 2): *read* gives read access, *write* gives write access (without including read access), *incr* allows addition of new information but neither read nor write is included. The combination of *read* and *incr*, i.e., $\mathit{read} \sqcup \mathit{incr}$ is abbreviated *rincr* gives read and incremental access. Similarly, $\mathit{write} \sqcup \mathit{incr}$ is abbreviated as *wincr*, which gives write and incremental access. Full access is given by a combination of *read* and *write* (which includes incremental access), i.e., *full* is the same as $\mathit{read} \sqcup \mathit{write}$. These general access rights can be combined with access rights on self, i.e., access rights when the principal is the subject herself. (details are omitted). For instance, a nurse should be able to see treatment data of a patient and add new data, and needs *rincr* access, while a lab assistant may add lab data and needs only *incr* access. A patient should see data about herself, which requires $\mathit{self} \sqcap \mathit{read}$.

A single policy ($\mathcal{P}$) is given by a triple (I, R, A), and a policy set ($\mathcal{P}_s$) is given by a set of policy triples (with meet and join operation defined). For our purposes, we annotate methods with single policies while data types are annotated with policy sets reflecting the permitted usage by different principals.

Example. The example in Figure 3 gives an illustration for declaring policies, and annotating methods and types with policies. The policy $(\mathit{Doctor}, \mathit{treatm}, \mathit{rincr})$ restricts access to objects typed by the *Doctor* interface, for only *treatm* (treatment) purposes, and with *rincr* data access. This is checked by a type and effect system in section 4. The policy set

$$\{(\mathit{Doctor}, \mathit{treatm}, full), (\mathit{Doctor}, \mathit{treatm}, \mathit{rincr}), (Nurse, \mathit{treatm}, \mathit{read})\}$$

restricts access by these three policies. Here, the policy $(\mathit{Doctor}, \mathit{treatm}, \mathit{rincr})$ is redundant since $(\mathit{Doctor}, \mathit{treatm}, \mathit{rincr}) \sqsubseteq (\mathit{Doctor}, \mathit{treatm}, full)$, and is colored grey to indicate that. Method *makePresc* has policy $(\mathit{Doctor}, \mathit{treatm}, \mathit{rincr})$, meaning that this method must be called by a Doctor object (or a more specialized object), for purposes of treatment and with read and incremental access (but not write access). Thus a doctor can add new prescription, but not change

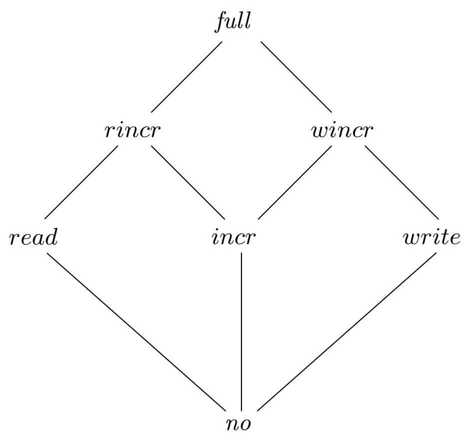

Fig. 2. The lattice for general access rights (without *self*). Note that *rincr* is the same as $read \sqcup incr$, *wincr* is the same as $write \sqcup incr$, and *full* is the same as $read \sqcup write$.

or remove old ones. Method *getPresc* has policy $(Nurse, treatm, read)$, meaning that this method must be called by a Nurse object (or a more specialized object such as a Doctor object), for purposes of treatment, and with read-only access. These two methods, with associated policies, are inherited in interface *PatientData*.

2.1 Policy Compliance Definition

Here, we briefly present a few definitions needed to express policy compliance.

Definition 1 (Policy Compliance). *The sub-policy relation* $\sqsubseteq$*, expressing* policy compliance*, is defined by*

$$(I', R', A') \sqsubseteq (I, R, A) \triangleq I \leq I' \wedge R' \leq R \wedge A' \sqsubseteq A$$

(where the last $\sqsubseteq$ *operation is on access rights) with* $\bullet$ *as* bottom element*, representing non-sensitive information. It follows that* $\sqsubseteq$ *is a partial order.*

A policy $\mathcal{P}'$ complies with $\mathcal{P}$ if it has the same or larger interface, the same or more specialized purpose, and if the access rights of $\mathcal{P}'$ are the same or weaker than that of $\mathcal{P}$. In particular, the policy of the implementation of a method should comply with that of the interface. Note that $\bullet \sqsubseteq \mathcal{P}$ expresses that an implementation without access to sensitive information complies with any policy.

Moreover, the use of *self* in the access part allows us to distinguish between different kinds of self access for different purposes, such as $(Patient, all, read \sqcap self)$ and $(Patient, private\_settings, self)$. The latter gives full access to data about *self* for purposes of *private settings*, while the first gives read access to data about *self* for all purposes.

We define a lattice over sets of policies with meet and join operations, and generalize the definition of compliance to sets of policies:

```
purpose $basic\_treatm, treatm$ where $basic\_treatm < treatm$

policy $\mathcal{P}_{Doc} = (Any, treatm, full)$
policy $\mathcal{P}_{AddPresc} = (Doctor, treatm, rincr)$
policy $\mathcal{P}_{GetPresc} = (Nurse, treatm, read)$
policy $\mathcal{P}_{Presc} = \{\mathcal{P}_{GetPresc}, \mathcal{P}_{AddPresc}, \mathcal{P}_{Doc}\}$

type Presc == Patient * String :: $\underline{\mathcal{P}_{Presc}}$

interface Patient extends Subject {Void getSelfData() :: $\underline{\mathcal{P}_{SelfPresc}}$}
interface AddPresc {Void makePresc(Presc newp):: $\underline{\mathcal{P}_{AddPresc}}$}
interface GetPresc {Presc getPresc(Patient p) :: $\underline{\mathcal{P}_{GetPresc}}$}
interface PatientData extends AddPresc, GetPresc {}
interface Nurse extends Principal { Presc nurseTask() :: $\underline{\mathcal{P}_{GetPresc}}$}
interface Doctor extends Nurse{ Void doctorTask(Patient p) :: $\underline{\mathcal{P}_{Doc}}$}

class PATIENTDATA() implements PatientData {
  type PData = List[Presc] :: $\mathcal{P}_{Presc}$
  PData pd = empty();
  Presc getPresc(Patient p){return last(pd/p)} :: $\mathcal{P}_{GetPresc}$
  Void makePresc(Presc newp) {
     if newp $\neq$emptyString() then pd:+ newp fi } :: $\mathcal{P}_{AddPresc}$ }

class DOCTOR() extends NURSE implements Doctor{//inherits pd
  Void doctorTask(Patient p){
    Presc oldp = pdb.getPresc(p);
    String text = ...; //new presc using symptoms info and oldp
    Presc newp = (p, text); // here, new sensitive data is created!
    pdb!makePresc(newp)}:: $\mathcal{P}_{Doc}$ }
```

Fig. 3. Interface, class, type, and policy definitions for the Prescription Example. Grey policy specifications are implicit while underlined ones need to be explicitly stated. A class implementation of Nurse is omitted. The projection pd/p is the list of strings associated to patient p, and the function $last$ gives the last element.

Definition 2 (Compliance of Policy Sets).

$$\{\mathcal{P}'_i\} \sqsubseteq \{\mathcal{P}_j\} \triangleq \forall i \,.\, \exists j \,.\, \mathcal{P}'_i \sqsubseteq \mathcal{P}_j$$

This expresses that a policy set S' complies with a policy set S if each policy in S' complies with some policy in S. We define meet and join operations over policy sets by set union and a kind of intersection, respectively, adding implicitly derivable policies:

Definition 3 (Join and Meet over Policy Sets).

$$S \sqcup S' \triangleq closure(S \cup S')$$

$$S \sqcap S' \triangleq \mathit{closure}(\{P \mid P \sqsubseteq S \wedge P \sqsubseteq S'\})$$

where the closure operation *is defined by*

$$\mathit{closure}(S) \triangleq S \cup \{(I, R, A \sqcup A') \mid (I, R, A) \sqsubseteq S \wedge (I, R, A') \sqsubseteq S\}$$

We have a lattice with $\emptyset$ as the bottom element. The closure operation adds implicitly derivable policies, and ensures that $\{(I, R, A \sqcup A')\} \sqsubseteq \{(I, R, A)\} \sqcup \{(I, R, A')\}$. For instance, $\{(\mathit{Doctor}, \mathit{treatm}, \mathit{read})\} \sqcup \{(\mathit{Doctor}, \mathit{treatm}, \mathit{write})\}$ is the same as $\{(\mathit{Doctor}, \mathit{treatm}, \mathit{full})\}$. These constructs are useful in specification of constraints and in capturing access to sensitive information with declared privacy policies. The meet operation typically reflects worst-case analysis.

Definition 4 (Implication on Policy Set). *We define the notation $\mathcal{P}s' \Rightarrow \mathcal{P}s$ ($\mathcal{P}s'$ implies $\mathcal{P}s$) by $\{\bullet\} \Rightarrow \mathcal{P}s$ and $\mathcal{P}s \sqsubseteq \mathcal{P}s'$ for $\mathcal{P}s'$ other than $\{\bullet\}$.*

Implication is used to check policy compliance of an actual parameter with respect to a formal parameter. If $\{\bullet\}$ is the policy on the actual parameter and $\mathcal{P}_{doc}$ the policy on the formal parameter, we will check $\{\bullet\} \Rightarrow \mathcal{P}_{doc}$.

Policies on Methods. Let $\mathcal{P}_{I,m}$ denote the policy of a method m given in an interface I, and $\mathcal{P}_{C,m}$ denote the policy of a method m given in a class C. We will require that the implementation of a method in a class (C) respects the policy stated in the interface (I), i.e., $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m}$. And we also require that a method redefined in an interface (I) respects the policy of that method in a superinterface (J), i.e., $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$. By transitivity of $\sqsubseteq$, a method implementation in a class that respects the policy given in an interface also respects the policy of the method given in a superinterface, i.e., $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m}$ and $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$ implies $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{J,m}$. For instance, consider an interface *GetPresc* with a method *getPresc*() with policy $(Nurse, \mathit{treatm}, \mathit{read})$. An implementation of this method in a class must have a policy that complies with it, such as $(Any, \mathit{treatm}, \mathit{read})$, $(Nurse, \mathit{treatm}, \mathit{self} \sqcap \mathit{read})$, or $(Nurse, \mathit{basic\_treatm}, \mathit{read})$. In contrast, the implementation cannot have policy $(\mathit{Doctor}, \mathit{treatm}, \mathit{read})$, as this would not allow a $Nurse$ as the caller object, and also not $(Nurse, \mathit{all}, \mathit{rincr})$, because this violates purpose and access restrictions.

Policies on Types. We let the policy of a type T, denoted $\mathcal{P}_T$, be a policy set. Let the policy set $\{(\mathit{Doctor}, \mathit{treatm}, \mathit{rincr}), (Nurse, \mathit{treatm}, \mathit{read})\}$ be the policy set on type *Presc*. This allows the data of type *Presc* to be accessed based on these two policies, depending on the calling context. For instance, if the caller is a *Doctor* object and the purpose is *treatm* then *read* as well as *incr* access is allowed on data of type *Presc*. The policy set of an actual variable must imply the policy set of the type of the corresponding formal variable. Together, the policies on methods and types provide sufficient abstractions to control access to sensitive data.

In the next section we consider a high-level imperative language for service-oriented systems where policy specifications are integrated.

3 Embedding Policy with Program Constructs

We target object-oriented, distributed systems (OODS) and consider the active object programming paradigm [5], which is based on the actor model [6] and gives a high-level view of communication aspects in OODS. In the active object model, objects are autonomous and execute in parallel, communicating by so-called asynchronous method invocations. We assume interface abstraction, i.e., an object can only be accessed through an interface and remote field access is illegal. This allows us to focus on major challenges of modern architectures, without the complications of low-level language constructs related to the shared-variable concurrency model.

Pr	$::= [\mathcal{T} \mid \mathcal{RD} \mid In \mid Cl]^*$	program
$\mathcal{T}$	$::=$ **type** $N\,[\overline{T}] = <type\_expression>\ [::\mathcal{P}s]$	type definition
T	$::= I \mid \mathsf{Int} \mid \mathsf{Any} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Void} \mid \mathsf{List}[T] \mid N$	types
In	$::=$ **interface** I [**extends** I^+] $\{D^*\}$	interface declaration
Cl	$::=$ **class** $C\ ([T\ z]^*)$	class definition
	[**implements** I^+] [**extends** C]	support, inheritance
	$\{[T\ w\ [:= ini]]^*$	fields
	$[B\ [::\mathcal{P}]]$	class constructor
	[[**with** I] $M]^*\}$	methods
D	$::= T\ m([T\ y]^*)\ [::\mathcal{P}]$	method signature
M	$::= T\ m([T\ y]^*)\ [\{s\}]\ [::\mathcal{P}]$	method definition
B	$::= \{[T\ x\ [:= rhs];]^*\ [s;]$ **return** $rhs\}$	method blocks
v	$::= w \mid x$	assignable variable
e	$::= v \mid y \mid z \mid \mathsf{this} \mid \mathsf{caller} \mid \mathsf{void} \mid f(\overline{e})$	pure expressions
ini	$::= e \mid$ **new** $C(\overline{e})$	initial value of field
rhs	$::= ini \mid e.m(\overline{e})$	right-hand sides
s	$::= \mathsf{skip} \mid s; s$	sequence
	$\mid v := rhs \mid v :+ rhs \mid e!m(\overline{e}) \mid I!m(\overline{e})$	assignment and call
	$\mid$ **if** e **then** s [**else** s] **fi**	if statement
	$\mid$ **while** e **do** s **od**	while statement

Fig. 4. BNF syntax of the core language. A field variable is denoted w, a local variable x, a method parameter y, a class parameter z, and list append is denoted $+$. The brackets in $[T]$ and $[\overline{T}]$ are ground symbols.

We propose a small core language, based on Creol [7], centered around a few basic statements. It has a compositional semantics which is beneficial to analysis [7,8]. The language is imperative and strongly typed, with data types for data structure locally inside a class. The data type sublanguage is side-effect-free. The motivation is that the language gives high-level descriptions of distributed systems and synchronous and asynchronous interaction based on methods, thereby avoiding shared variable access, and avoiding explicit signaling and notification. The BNF syntax of the language is summarized in Figure 4. As before, optional parts are written in brackets (except for type parameters, as in $\mathsf{List}[T]$, where the brackets are ground symbols). Class parameters (z), method parameters (y) the implicit class parameter this and the implicit method parameter caller are read-only. A class may implement a number of interfaces, and for each method of an interface (of the class) it is required that the class defines the method such that policy of each method parameter and return value are respected. Additional methods may be defined in a class, but these may not be called from outside the class. The language supports single class inheritance and multiple interface inheritance (using the keyword **`extends`**). Below, we give BNF syntax for method and type declarations.

Definition 5 (Method Declaration Syntax).

$$T\ m([Y\ y]^*)\ [:: \mathcal{P}]$$

where T is the result type and Y is the type of parameter y.

An inherited method m inherits the policy of m from the superinterface, unless the interface declares its own policy for m. However, the redefined policy of m (of interface I) cannot be more restrictive than that of the superinterface (J), i.e., $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$, ensuring that a class implementation of m satisfying $\mathcal{P}_{I,m}$ also satisfies any declarations of m in a superinterface.

Definition 6 (Data Type Declaration Syntax and Sensitivity).

$$\texttt{type}\ N\ [\mathit{TypeParameters}] =<\mathit{type\_definition}>\ \ [:: \mathcal{P}s]$$

where the type parameters are optional. The predefined basic types (Nat, Int, String, Bool, Void) are non-sensitive*. A user-defined type is* sensitive *if a policy set is specified in the type definition.*

For example, a sensitive *String* type restricted by a policy $\mathcal{P}_s$ can be defined by

$$\texttt{type}\ \mathit{Info}\ = \mathit{String} :: \mathcal{P}_s$$

and encryption could go from Info to String, and decryption the other way.

We consider next *sensitive* functions, which create new sensitive data, for instance a product of individually non-sensitive data may be sensitive. Generator functions (here called constructors) are considered *sensitive* if they i) combine information about a subject with non-sensitive or sensitive information or ii) use sensitive information. We assume that sensitive generators produce sensitive types (with some exceptions, such as constructors of encrypted data). Defined functions are *sensitive* if their type is sensitive and the definition directly or indirectly contains a sensitive application of a constructor. For instance we may (recursively) define a parameterized list type by $\mathit{List}[T] = \mathit{empty}()|\mathit{append}(\mathit{List}[T]*T)$ meaning that lists have the form $\mathit{empty}()$ or $\mathit{append}(l, x)$, where l is a list and x a value of type T. (We let the notation $l+x$ abbreviate $\mathit{append}(l, x)$.) The list is sensitive if T is sensitive, but the append constructor function is not sensitive. A pair product type can be defined by $\mathit{PatientData} = (\mathit{Patient} * \mathit{String})$ where *Patient* is a interface representing a data *subject*. This type is sensitive (even though *String* is not), and the pair ($\mathit{current\_patient}$, "*no health problems*") is a sensitive application of the product constructor. These examples suffice for

$$\text{(P-VAR)}\ \frac{\mathit{read} \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C,m))}{C,m \vdash [\Gamma]\ v ::\ \Gamma[v] \sqcap \Gamma[pc]}$$

$$\text{(P-FUNC)}\ \frac{\begin{array}{ll} C,m \vdash [\Gamma]\ e_i ::\ \mathcal{P} & \text{for each argument } e_i \text{ of a sensitive type} \\ \mathit{write} \sqsubseteq \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C,m)) & \text{if } f_T \text{ is a sensitive function} \end{array}}{C,m \vdash [\Gamma]\ f_T(\overline{e}) ::\ \mathcal{P}_T \sqcap \Gamma[pc]}$$

$$\text{(P-CALL)}\ \frac{\begin{array}{c} \mathcal{P}_{I,n} \sqsubseteq_{Co,R}\ \mathcal{P}_{C,m}@(C,m) \\ C,m \vdash [\Gamma]\ e :: \mathcal{P}' \\ C,m \vdash [\Gamma]\ e_i :: \mathcal{P}_i \quad \mathcal{P}_i \Rightarrow \mathcal{P}_{par(I,n)_i} \quad \text{for each i} \end{array}}{C,m \vdash [\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}_{out(I,n)}}$$

Fig. 5. Policy Rules for Expressions and Right-Hand Sides.

our purposes here. It can be detected statically if a function is sensitive (further details are omitted). Applications of sensitive functions may create new sensitive data, something which require write access. This way the policy control is driven by the declared data types rather than variable declarations. Data types are reusable and therefore their policies are likely to more reliable and appropriate than one-time adhoc specification for program variables.

When the lawful basis of processing of personal information is performance of contract or other valid bases but not the consent, the policies must be formulated in a way that ensures that they are built into the system *by default*, i.e., no measures are required by the data subject in order to maintain his/her privacy. However, when consent is the basis of processing the data subjects, choices in privacy settings are captured at runtime (as outlined in [9]).

We next show how to define static policy checking for our core langauge.

4 An Effect System for Privacy

We propose static policy checking defined by a set of syntax-directed rules, given as a type and effect system [10], but dealing with policies rather than types. We consider two kinds of judgments. For a statement s, the judgment

$$C,m \vdash [\Gamma]\ s\ [\Gamma']$$

expresses that inside a method body m and an enclosing class C, the statement(list) s when started in a state satisfying the environment Γ results in a state satisfying the environment Γ'. Here Γ is a mapping from program variable names to policy sets, such that the policy set of a variable in a given state gives an upper bound of the permitted operations. In order to deal with branches of if- and while-statements where the context policy is influenced by that of the if- and while-tests, Γ uses an additional variable pc (the program context) reflecting the current branching policy (as in [8]). Note that the rules are right-constructive in the sense that Γ' can be constructed from Γ and s.

For an expression or right-hand side e, the judgment

$$C,m \vdash [\Gamma]\ e :: \mathcal{P}s$$

expresses that the evaluation of e in a state satisfying Γ gives a value satisfying the policy set $\mathcal{P}s$, where m is the enclosing method and C the enclosing class.

Figure 5 defines the typing rules for expressions and right-hand sides, and Figure 6 defines the typing rules for (selected) statements. We let $\mathcal{P}_{I,m}$ denote the policy of method m of interface I, $\mathcal{P}_{C,m}$ denote the policy of method m of class C, and $\mathcal{P}_T$ denote the policy associated with a type T. If no policy is specified for any declaration, we understand that there is no sensitive information, i.e., the policy is $\{\bullet\}$. Data types with sensitive constructors will be considered sensitive. A non-sensitive method would not be able to access or create sensitive data, and a non-sensitive type declaration would not allow assignment of sensitive information to variables of that type.

Rule P-VAR says that the policy of a variable v (a field, parameter, or local variable) is the one recorded in Γ for v, i.e., $\Gamma[v]$, combined with that of the program context *pc*. The premise states that there must be read access to v, both according to the policy set of the variable and according to the policy set of the enclosing method body. If the policy of the enclosing method m is (I,R,A), the *policy set of the method body* is defined by

$$(I,R,A)@(C,m) \triangleq (I,R,A) \,\cup\, (\cup_i \{(I_i,R,A)\})$$

where I_i ranges over all the interfaces of C that export m. Thus the policy set of the method body is that of the method and those where this object is the principle (as seen through one of the interfaces exporting m).

Rule P-FUNC says that the policy of a function application $f_T(\overline{e})$ is that of the resulting type T (detected by ordinary typing) combined with that of the program context *pc*. Sensitive arguments must be checked (which ensure read access to the variables occurring in these arguments), and in case f is a sensitive function application, there must be write access according to the policy of T and the policy of the method body. Constants (function without arguments), as well as object creation, have policy set $\{\bullet\}$.

Rule P-CALL says that the policy of a remote call $e.n_I(\overline{e})$ where I is the interface of the method (detected by ordinary typing), is the policy on the return type of the method (as given by the declaration of m in I). The first premise ensures that the policy of the called method complies with policy of the enclosing body. The second premise ensures that the callee expression has a valid policy, and the last premise ensures each actual parameter has a policy set that implies the policy set of the corresponding formal one.

The rule P-SKIP says that the environment is not changed. The rule for sequential composition says that the final environment of s_1 is used as the starting environment for the next statement s_2. The rules P-WRITE and P-LOCAL-WRITE say that the final environment is that of the right-hand side. Writing to a field requires write access, while writing to a local variable is always allowed. An incremental assignment $w :+e$ requires *incr* access, and the final environment is as for the assignment $w := w + e$. The premises for asynchronous call is as for P-CALL, and the resulting environment is unchanged (since no variable is changed).

Note that, if by mistake, no policy is specified due to forgetfulness, the static compliance checking would detect any use of sensitive information and the pro-

$$(\text{P-SKIP})\ \frac{}{C, m \vdash [\Gamma]\ skip\ [\Gamma]}$$

$$(\text{P-COMPOSITION})\ \frac{C, m \vdash [\Gamma]\ s_1\ [\Gamma_1] \quad C, m \vdash [\Gamma_1]\ s_2\ [\Gamma_2]}{C, m \vdash [\Gamma]\ s_1; s_2\ [\Gamma_2]}$$

$$(\text{P-WRITE})\ \frac{\begin{array}{c} C, m \vdash [\Gamma]\ rhs :: \mathcal{P} \\ write \sqsubseteq \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C, m)) \end{array}}{C, m \vdash [\Gamma]\ w := rhs\ [\Gamma[w \mapsto \mathcal{P}]]}$$

$$(\text{P-LOCAL-WRITE})\ \frac{C, m \vdash [\Gamma]\ rhs :: \mathcal{P}}{C, m \vdash [\Gamma]\ x := rhs\ [\Gamma[x \mapsto \mathcal{P}]]}$$

$$(\text{P-INCR})\ \frac{\begin{array}{c} C, m \vdash [\Gamma]\ rhs :: \mathcal{P} \\ incr \sqsubseteq \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C, m)) \end{array}}{C, m \vdash [\Gamma]\ w :+ rhs\ [\Gamma[w \mapsto \Gamma[w] \sqcap \mathcal{P}]]}$$

$$(\text{P-ASYNCCALL})\ \frac{C, m \vdash [\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}_{out(I,n)}}{C, m \vdash [\Gamma] \quad e!n_I(\overline{e})\ [\Gamma]}$$

Fig. 6. Policy Rules for Statements.

gram would not pass the privacy checks. In particular, data types with constructors associating data to subjects will be considered sensitive. A non-sensitive method would not be able to access or create sensitive data, and a non-sensitive type declaration would not allow assignment of sensitive information to variables of that type.

We next show how to apply the static analysis on the Prescription case study.

5 Case Study

Consider the example from Figure 3 where *Doctor*, *Nurse*, *Patient*, *PatientData*, *AddPresc*, *GetPresc* are interfaces. A *PatientData* object contains data for a number of patients, and can be accessed by doctors and nurses, based on different policies. Policies are declared by the keyword **`policy`**. Patient data *pd* of type *PData* (list of *Presc*) is labeled with polices: $\{\mathcal{P}_{GetPresc}, \mathcal{P}_{Doc}\}$, and an implicit policy $(Subject, all, self \sqcap read)$ is included in every policy set to allow read access when the principal is the data subject. This policy ($\mathcal{P}_{Presc}$) allows *(i)* a patient to access his/her own data, *(ii)* gives *full* (i.e., read, incr, write) access to the *Doctor* for *treatm* purposes, and *(iii)* gives *read-only* access to the *Nurse* for *treatm* purposes. The purpose *treatm* is declared by the keyword **`purpose`**. The policies need to be declared only once and then the effect system will keep track of the policies in a given program state. For example, the declaration of *makePresc*() includes the policy $\mathcal{P}_{AddPresc}$. Now we show an application of a few type rules, on the statements in the method $doctorTask()$ from Figure 3.

1. $x := rhs$
   $String\ text = rhs$ //Apply P-LocalWrite
   The premise $rhs :: \bullet$ associates $\bullet$ with rhs, since it is a local variable and has no policy.
   $\Gamma[x \mapsto \mathcal{P}] \implies \Gamma[text \mapsto \bullet]$
   Gamma for $text$ is updated with $\bullet$.
2. $Presc\ newp = (p, text);$ //Apply P-Func, P-LocalWrite
   (a) $read \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C, m))$
   $(\Gamma[p] \sqcap \Gamma[text]) \sqcap \mathcal{P}_{Presc}$
   $(\bullet \sqcap \bullet) \sqcap (\mathcal{P}_{C,m}@(C, m))$
   i.e., $(\bullet \sqcap \bullet) \sqcap \mathcal{P}_{Doc}$ since $(\mathcal{P}_{C,m}@(C, m)) = \mathcal{P}_{Doc}$
   which reduces to $\mathcal{P}_{Doc}$
   $read \sqsubseteq \mathcal{P}_{Doc}$ (*i.e.,* $read \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C, m))$)
   which reduces to $read \sqsubseteq full$, using the notation $A \sqsubseteq (I, R, A')$ when $A \sqsubseteq A'$, and $A \sqsubseteq \{(I, R, A')_i\}$ when $A \sqsubseteq (I, R, A')_i$ for some i (i.e., $A \sqsubseteq A'_i$).
   (b) $write \sqsubseteq \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C, m))$, since the constructor $(\_, \_)$ is sensitive
   $write \sqsubseteq \mathcal{P}_{Doc} \sqcap \mathcal{P}_{Presc}$
   which reduces to $write \sqsubseteq full$, and the policy of $(p, text)$ is $\mathcal{P}_{Presc}$
   (c) $\Gamma[newp \mapsto \Gamma[(p, text)]]$
   $\Gamma[newp \mapsto \mathcal{P}_{Presc}]$ //since pc is empty here

In the first statement, the policy set on *text* is $\{\bullet\}$ because it is not yet associated with a subject. But when non-sensitive *text* is combined with a subject identity, this is seen as construction of a sensitive data, and P-Func is used to ensure that the information can be read and constructed by the current context. The rest of the example can be checked in a similar way.

6 Related Work

Language-based mechanisms are techniques based on programming languages that are often used in developing secure applications. In particular, language-based security mechanisms are used in specification and enforcement of security policies. In recent years, various techniques (compilers, automated program analysis, type checking, program rewriting etc.) have been explored from the perspective of their applicability in enforcing security and privacy policies in programs. Privacy by Design (PbD) has been discussed and promoted from several viewpoints such as privacy engineering [1,11,12], privacy design patterns [13,14], and formal approaches [15,16,17]. Tschantz and Wing, in [17] and Daniel Métayer, in [15] discuss the significance of formal methods for foundational formalizations of privacy related aspects. In [16], Schneider discusses the main ideas of *Privacy by Design* and summarizes key challenges in achieving Privacy by Construction and probable means to handle these challenges. The paper calls for ways to ensure control of *purpose* integrated in programming languages. It is also indicated that in order to ensure that privacy-compliant code is sound and correct, formal methods would be helpful in proving soundness and completeness (with respect to a set of predefined privacy concepts). Privacy design strategies [13] focus on how to take privacy requirements into account from the beginning and make it a software quality attribute. The engineering aspects of privacy by design is addressed, but there is a lack on how to apply them in practice. In our work, we adhere to several privacy design strategies such as separating and hiding the data, and encapsulation in an object-oriented context.

Hayati and Abadi [4] describe a language-based approach based on information-flow control, to model and verify aspects of privacy policies in the Jif (Java Information Flow) programming language. In this approach data collected for a specific purpose is annotated with Jif principals and then the methods needed for a specific purpose are also annotated with Jif principals. Explicitly declaring purposes for data and methods ensures that the labeled data will be used only by the methods with connected purposes. Purposes are organized in a hierarchy, with sub-purposes. However, this representation of purpose is not sufficient to guarantee that principals will perform actions compliant with the declared purpose. But this can be checked statically in our approach, because the principal is restricted by a purpose-based access control.

Basin et al. [18] propose an approach that relates a purpose with a business process and use formal models of inter-process communication to demonstrate GDPR compliance. Process collection is modeled as data-flow graphs which depict the data collected and the data used by the processes. Then these processes are associated with a data purpose and are used to algorithmically i) generate data purpose statements, ii) detect violation of data minimization, and iii) demonstrate compliance of some more aspects of GDPR. Since in GDPR, end-users should know the necessary purpose of data collection, some works such as [18] propose to audit logs and detect if a computer system supports a purpose. In a continuation of this work [19], Arfelt et al. show how such an audit can be automated by monitoring. Automatic audits and monitoring can be applied to a system like ours as a complementary step to verify how it complies with the GDPR. Besides, our work is more focussed on integrating such legal instruments during the design phase, using formal language semantics. In [20], Adams and Schupp consider black-box objects that communicate through messages. The approach is centered around algorithms that take as input an architecture and a set of privacy constraints, and output an extension of the original architecture that satisfies the privacy constraints. This work is complementary to ours in that it puts restrictions on the run-time message handling. In contrast to our work, the approach does not concern analysis of program code.

In [21], Ferrara and Spoto discuss the role of static analysis for GDPR compliance. The authors suggest combining taint analyses and backward slicing algorithms to generate reports relevant for the various actors (i.e., data protection officers, chief information officers, project managers, and developers) involved at various stages of GDPR compliance. In particular, taint analysis is performed on each program statement and then the data-flow of sensitive information is reconstructed using backward-slicing. These flows are then abstracted into the information needed by the compliance actors. However, they do not formalize nor check privacy policies (as we do).

In the sense of access control mechanisms such as RBAC that controls and restricts system access to authorized users, there are some common features. In addition to the hierarchies of roles and access rights supported by RBAC, our framework introduces hierarchies of purposes to control role access. However, our work uses static analysis while RBAC uses runtime analysis. Anthonysamy et al. [22] demonstrate a *semantic-mapping* approach to infer function specifications from semantics of natural language. This technique is useful in compliance verification as it aids in identification of program constructs that implements certain policies. The authors implement this technique in a tool, CASTOR, which takes policy statements (in natural language) and source code as input, and outputs a set of semantic mappings between policies and function specifications (function name, associated class, parameters etc.).

7 Conclusion

We have investigated challenges and opportunities in approaching privacy from the *by-design* perspective, i.e., embedding privacy design requirements into a language. We have considered a small core language supporting active objects, and extended it to integrate privacy policies. We chose three primary constituents of a privacy policy, i.e., *principal, purpose*, and *access right*. Policies are declared for methods and data types, and together restrict the usage of sensitive data.

We defined a language for formulating these policies, discussed static privacy polices, and formalized a concept of static privacy policies. We have formulated rules for policy compliance, given by an extended effect system. The problem of checking a program's compliance with privacy policies, reduces to efficient type-checking. The analysis is class-wise, which is a benefit in open object-oriented systems, and for scalability. Needless to mention that much work needs to be done, in terms of defining possibly new constructs and abstractions in order to formalize the essential data protection principles. In the future we would like to *i)* extend the policy definition language, to express a wider range of privacy restrictions, *ii)* work out a larger case study, and *iii)* in particular focus on the dynamic policy and consent management.

Acknowledgments. The work is supported by the projects IoTSec no. 248113 and by SCOTT no. 283085. (its-wiki.no/wiki/IoTSec:Home and www.scott-project.eu)

References

1. G. Danezis, J. Domingo-Ferrer, M. Hansen, J.-H. Hoepman, D. L. Métayer, R. Tirtea, and S. Schiffner, "Privacy and data protection by design-from policy to engineering," *arXiv preprint arXiv:1501.03726*, 2015.
2. European Parliament and Council of the European Union, "The General Data Protection Regulation (GDPR)." `https://eur-lex.europa.eu/eli/reg/2016/679/oj`. Accessed: 2019-12-12.

3. S. Tokas, O. Owe, and T. Ramezanifarkhani, "Static Checking of GDPR-Related Privacy Compliance for Object-Oriented Distributed Systems," under review, 2019.
4. K. Hayati and M. Abadi, "Language-based enforcement of privacy policies," in *Intern. Workshop on Privacy Enhancing Technologies*, pp. 302–313, Springer, 2004.
5. O. Nierstrasz, "A tour of Hybrid – a language for programming with active objects," in *Advances in Object-Oriented Software Engin.*, pp. 67–182, Prentice-Hall, 1992.
6. C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *Proc. of the Third International Joint Conference on Artificial Intelligence*, IJCAI'73, pp. 235–245, Morgan Kaufmann Publishers Inc., 1973.
7. E. B. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," *Software & Systems Modeling*, vol. 6, pp. 39–58, Mar 2007.
8. T. Ramezanifarkhani, O. Owe, and S. Tokas, "A secrecy-preserving language for distributed and object-oriented systems," *Journal of Logical and Algebraic Methods in Programming*, vol. 99, pp. 1–25, 2018.
9. S. Tokas and O. Owe, "A formal framework for consent management," Proc. 31st Nordic Workshop on Programming Theory, NWPT'19, Nov. 2019. `https://cs.ttu.ee/events/nwpt2019/abstracts/nwpt19-abstracts-draft.pdf`.
10. F. Nielson and H. R. Nielson, "Type and effect systems," in *Correct System Design: Recent Insights and Advances*, pp. 114–136, Springer, 1999.
11. S. Gürses, C. Troncoso, and C. Diaz, "Engineering privacy by design reloaded," in *Amsterdam Privacy Conference*, pp. 1–21, 2015.
12. N. Notario, A. Crespo, Y.-S. Martín, J. M. Del Alamo, D. Le Métayer, T. Antignac, A. Kung, I. Kroener, and D. Wright, "PRIPARE: integrating privacy best practices into a privacy engineering methodology," in *2015 IEEE Security and Privacy Workshops*, pp. 151–158, IEEE, 2015.
13. J.-H. Hoepman, "Privacy design strategies," in *IFIP International Information Security Conference*, pp. 446–459, Springer, 2014.
14. M. Colesky, J.-H. Hoepman, and C. Hillen, "A critical analysis of privacy design strategies," in *2016 IEEE Security and Privacy Workshops (SPW)*, pp. 33–40, 2016.
15. D. Le Métayer, "Formal methods as a link between software code and legal rules," in *Int. Conf. on Software Engineering. and Formal Methods*, pp. 3–18, Springer, 2011.
16. G. Schneider, "Is privacy by construction possible?" in *International Symposium on Leveraging Applications of Formal Methods*, pp. 471–485, Springer, 2018.
17. M. C. Tschantz and J. M. Wing, "Formal methods for privacy," in *International Symposium on Formal Methods*, pp. 1–15, Springer, 2009.
18. D. Basin, S. Debois, and T. Hildebrandt, "On purpose and by necessity: compliance under the GDPR," *Proceedings of Financial Cryptography and Data Security*, vol. 18, pp. 20–37, 2018.
19. E. Arfelt, D. Basin, and S. Debois, "Monitoring the GDPR," in *European Symposium on Research in Computer Security*, pp. 681–699, Springer, 2019.
20. R. Adams and S. Schupp, "Constructing independently verifiable privacy-compliant type systems for message passing between black-box components," in *Verified Software. Theories, Tools, and Experiments*, pp. 196–214, Springer, 2018.
21. P. Ferrara and F. Spoto, "Static analysis for GDPR compliance," in *Proceedings of the Second Italian Conference on Cyber Security, Milan,* no. 2058 in CEUR Workshop Proceedings, 2018. `http://ceur-ws.org/Vol-2058/paper-10.pdf`.
22. P. Anthonysamy, M. Edwards, C. Weichel, and A. Rashid, "Inferring semantic mapping between policies and code: the clue is in the language," in *Intern. Symposium on Engineering Secure Software and Systems*, pp. 233–250, Springer, 2016.