# Privacy Injector — Automated Privacy Enforcement through Aspects

Chris Vanden Berghe[1,2] and Matthias Schunter[1]

[1] IBM Research, Zurich Research Laboratory
Säumerstrasse 4, CH-8803 Rüschlikon, Switzerland
{vbc,mts}@zurich.ibm.com
[2] Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
chrisvdb@cs.kuleuven.be

**Abstract**

Protection of personal data is essential for customer acceptance. Even though existing privacy policies can describe how data shall be handled, privacy enforcement remains a challenge. Especially for existing applications, it is unclear how one can effectively ensure correct data handling without completely redesigning the applications. In this paper we introduce Privacy Injector, which allows us to add privacy enforcement to existing applications.

Conceptually Privacy Injector consists of two complementary parts, namely, a privacy metadata tracking and a privacy policy enforcement part. We show how Privacy Injector protects the complete life cycle of personal data by providing us with a practical implementation of the "sticky policy paradigm." Throughout the collection, transformation, disclosure and deletion of personal data, Privacy Injector will automatically assign, preserve and update privacy metadata as well as enforce the privacy policy. As our approach is policy-agnostic, we can enforce any policy language that describes which actions may be performed on which data.

## 1 Introduction

An increasing number of enterprises make privacy promises to meet customer demand or to implement privacy regulations. As a consequence, enterprises aim at protecting data against accidental misuse, including unwarranted disclosure and over-retention. Recent approaches towards formalizing privacy regulations have addressed the issue of how permitted data usage can be formalized [3, 10, 14, 16]. However, in practice two major challenges remain. The first challenge is to assess the actual privacy status of an enterprise, i.e., what data is stored and what data has been collected under what policy. The second challenge is how to enforce the given privacy promise consistently throughout existing and new applications.

A first step in addressing these challenges has been the "sticky policy paradigm" as proposed in Karjoth et al. [16]. This paradigm requires that a privacy promise made to a data subject stick to the data to later identify how this data can be used. For cross-enterprise transfer, policy refinement can be used to enforce sticky policies [3]. However, for enterprise-internal use, there is no clear concept how policies can be reliably

associated with data and how policies can be managed. This holds in particular for existing enterprise applications in which privacy enforcement was not included as a non-functional design requirement.

In this paper we introduce Privacy Injector, which leverages the Aspect-Oriented Software Development (AOSD) [9] paradigm to modularize and encapsulate privacy enforcement. This allows us to add privacy enforcement functionality late in the software development cycle or even in the maintenance phase of applications. Privacy Injector consists of a *privacy metadata tracking* part and a *privacy policy enforcement* part. The former is a practical implementation of the aforementioned sticky policy paradigm, whereas the latter is responsible for the actual enforcement of the sticky policies.

The privacy metadata tracking part consists of three components. The first component, *privacy metadata assignment*, is responsible for assigning the appropriate privacy metadata to data that enters the system. This is achieved by instrumenting the input vectors of the execution platform, i.e., all functions responsible for collecting external data. The second component, *metadata-preserving data operations*, is responsible for preserving and updating this privacy metadata when operations are performed on this data. This is achieved by instrumenting all data operations to update the privacy metadata to reflect changes resulting from the operations. The third component, *metadata persistence*, is responsible for preserving, restoring, updating and removing privacy metadata when data is made persistent, retrieved, modified or removed, respectively. This is accomplished by leveraging the event systems exposed by persistence services.

The privacy policy enforcement part ensures that the appropriate tests and actions specified by the privacy policy are performed upon usage and disclosure of the data. The different ways in which data can be disclosed are called output vectors, which are again instrumented and retrofitted with the policy-enforcing functionality. For example, when an application calls the API for sending e-mail, this function is intercepted and the required conditions and obligations, as described by the privacy metadata attached to the function's parameters, are checked and the necessary actions performed.

We rely on the Aspect-Oriented Software Development paradigm for implementing the instrumentation, i.e., for making the connection between Privacy Injector and the target application. Our method is platform-independent, although we focus on Java to illustrate our concepts for this paper. Privacy Injector does not require the source code of the target applications; however it makes some assumptions about the target applications that will be discussed later in this paper.

The goal of this work is to show that Privacy Injector provides a useful methodology for implementing the sticky policy paradigm and enforcing privacy in applications in which privacy enforcement was not included as a design goal. In this paper we focus on preventing *unwarranted disclosure of personal data*; other possible and potentially valuable use-cases, e.g., preventing over-retention of personal data, are only touched upon briefly.

### 1.1 Outline

In Section 2 we discuss related work on privacy policies, privacy enforcement, flow control, and aspect-oriented software development. In Section 3 we discuss the life cycle of personal data. In Section 4 we introduce Privacy Injector conceptually, whereas

Section 5 is devoted the implementation details of our prototype. In Section 6 describes the benefits and limitations of our approach, and Section 7 concludes the paper.

## 2 Related Work

### 2.1 Privacy Policies and Policy Attachment

We distinguish privacy policies and privacy notices. Formalized privacy policies were described in [3, 10, 14, 16]. Privacy policy languages formalize which users can perform which operations on given data types for which purpose [5]. In addition, privacy policies can specify conditions (such as usage only during day-time; see [21]) or obligations (such as limited retention; see [4, 7, 13, 27]). Similarly to the approach described in this paper, privacy policies aim at enterprise-internal use.

Privacy notices, on the other hand, formalize the privacy promises of an enterprise to end-users. The World Wide Web Consortium has standardized the Platform for Privacy Preferences (P3P) that allows enterprises to declare which data is collected and how it will be used [22]. The adoption rate of P3P, however, remains relatively low [8].

When comparing the two approaches, languages for enterprise-internal privacy practices and technical privacy policy enforcement offer finer-grained distinction of users, purposes, etc.

An open challenge is how to implement sound policy management for privacy policies and how policies can be enforced, namely, how policies can be attached to data and how they can be enforced automatically. Karjoth et al. [16] propose the “sticky policy paradigm” that defines that a privacy promise made to the data subject should stick to the data to later identify how this data may be used. As addressed in [2, 3], sticky policy transfer between enterprises can be implemented using policy comparison. Policy attachment and enforcement for legacy applications remain an open challenge.

### 2.2 Privacy Policy Enforcement

Privacy policy enforcement has several aspects, depending on the life cycle of the personal data that is collected and used (see Section 3). For data protection during collection, consistent use of privacy notices is essential. Privacy notices can be formalized using P3P [22], whereas their consistent use can be verified using the Watchfire tool [30]. Enforcement of privacy policies depends heavily on the systems that store and handle personal data. IBM has published technologies for declaring and enforcing privacy policies for Java Beans [12].

For policy enforcement inside databases, the concept of Hippocratic databases has been described in [1]. The core idea is to use SQL rewriting to include policy evaluation into the (modified) databased query that is actually being processed. This enables the database to automatically return the subset of records where usage is authorized. For newly built applications in which policy authorization can be delegated to an authorization engine, various authorization engines have been designed, including the one described in [21]. These engines enable an application to query whether a certain use of data is allowed.

### 2.3 Flow Control

Language-based information flow security was surveyed by Sabelfeld and Myers in [28]. The main focus of current research is to statically determine potential flows and whether a program complies with a desired flow-control policy. Flow-control approaches that perform static checking need to perform a worst-case analysis to catch all potential violations. To resolve this problem, Myers [19, 20] proposes a Java extension called JFlow that annotates Java code using flow-control constructs. This enables a precompiler that performs a static verification and then generates Java code that performs additional run-time checks.

The advantage of Privacy Injector over the existing approaches lies mainly in its practical applicability. For example, it does not require to have the application's source code available. Nor does it require a modified runtime or special language constructs; a simple configuration file specifying the policy suffices in most cases. Finally, no flow-control policy must be known at compile time, in contrast to the existing approaches where this forms a major obstacle in their acceptance because the privacy policy actually consented to is only determined at run-time.

Three design choices contribute to this improved practical applicability: the use of the AOSD paradigm, some assumptions made about the applications (discussed further) and a different adversary model. Rather than aiming at also identifying and preventing hidden information flow, we focus on an adversary model in which a non-malicious developer accidentally uses or discloses data in a manner violating the privacy policy.

### 2.4 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) [9] is a software engineering paradigm that enables the *Separation of Concerns* [23], i.e., the breaking down of applications into components that minimize the overlap in functionality. In particular, AOSD focuses on the modularization and encapsulation of cross-cutting concerns.

Cross-cutting concerns are areas of interest that cannot easily be separated and encapsulated through existing software development paradigms such as object-oriented or procedural programming. Logging is the archetypical example, as it touches every logged component, and existing software development paradigms force program code responsible for the logging functionality to be scattered throughout these components.

AOSD refers to software development as a whole, including design, testing, etc., whereas Aspect-Oriented Programming (AOP) [18] refers specifically to the programming part. Several distinct AOP implementations exist, each providing language constructs to express both the cross-cutting concerns (the advice) as well as the points in the application at which the advice should be integrated (the join points), and tools for performing the actual integration (weaving). In this work we will use AspectJ [17, 25], which is the most popular and widely supported AOP implementation for Java.

AspectJ provides a fine-grained qualification of the join points, called pointcuts, enabling instrumentation of methods, constructors, field access, etc., of which the selection is based on a combination of name, return type, parameters, etc. The actual weaving step of AspectJ is performed at the byte-code level, and thus no access to the target application's source code is required to provide it with additional functionality.

Although the principle of separation of concerns originally served mainly as a guideline for structuring application functionality, it has more recently been applied to separate the non-functional from the functional application requirements. One important non-functional application requirement that has proved difficult to separate or modularize with standard software development tools and methodologies is security. One reason for that is the cross-cutting nature of security. In [31] De Win et al. investigate the usefulness of AOP for secure software development.

## 3 The Life Cycle of Personal Data

Protecting personal data throughout its entire life cycle is essential to implement a given privacy notice. We now define a life cycle model of personal data as well as the corresponding privacy metadata that is required to track the life cycle and usage of personal data. Figure 1 shows the UML sequence diagram representation [6] of the life cycle of some personal data given out by data subject DS to enterprise A, which stores it in storage A and discloses it in turn to enterprise B. The arrows represent the direction of the data flow resulting from an action, the parameters represent the context required for enforcing the privacy of a particular action. The quotes represent changes to the parameters, e.g., *consent* is the consent given by the data subject to enterprise A, whereas *consent'* refers to the consent given to enterprise B (i.e., a subset of the original consent) and *consent"* refers to the additional consent requested by enterprise B. The boxed area shows the domain in which Privacy Injector is active, i.e., the enterprise-internal usage of personal data. Privacy enforcement for cross-enterprise transfer can however also benefit from it. In Section 4 we show how the life cycle phases are protected by Privacy Injector.

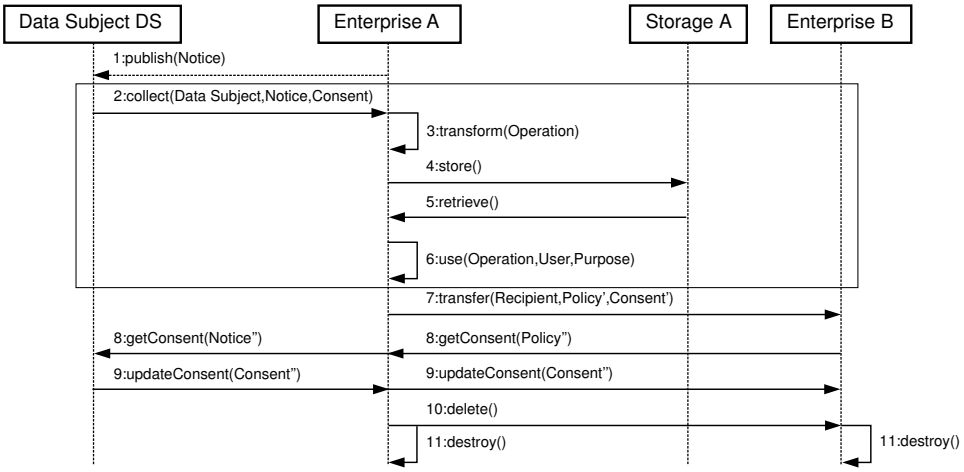


**Fig. 1.** Life cycle and flow of personal data.

*Publication (1):* Before actual collection of personal data, the enterprise publishes a privacy notice describing the intended use of the the data subject's personal data. This is typically done by displaying a notice on a website or as part of a web form. This flow is independent of the actual data flows and is therefore represented by a dotted line.

*Collection (2):* Data subject DS consents with the privacy notice and sends personal information to the enterprise. During this data-collection flow, the collecting enterprise associates the personal data with the privacy metadata required to enforce its privacy notice to the data subject. The privacy metadata includes the following:

*Data Subject*: The person whose personal data has been collected
*Privacy policy*: The privacy policy that governs the data's usage
*Consent*: Opt-in and opt-out choices as collected from the data subject

The identifier for the data subject is needed if the privacy notice promises notification under certain circumstances. The privacy policy formalizes the intended use of the data and governs the data's usage. The privacy policy is a refinement of the privacy notice that has been consented to by the data subject. For example, whereas a notice may say "we share your data with our partners for processing orders", the privacy policy should list the actual partners. It is complemented by the consent, which defines opt-in and opt-out choices that refine the usages defined in the privacy policy.

*Transformation (3):* Most data is subjected to several transformation operations throughout its lifetime. These transformation operations typically normalize, merge and extract data to make it suitable for its environment and purpose. Although these operations are very common, they constitute a challenge that has not been addressed by earlier approaches. In particular, ensuring that the associated privacy metadata remains consistent with the transformed data is non-trivial.

*Storage and Retrieval (4 & 5):* One common type of data operation is storage and retrieval using a database or other persistent storage. The privacy metadata associated with the personal data should "survive" such storage and retrieval operations. For this, the privacy metadata is stored and retrieved together with the data it belongs to. Persistent storage media, e.g., relational databases, are typically not metadata-aware, and this functionality will thus have to be provisioned by another system component.

*Usage (6):* When data is accessed by a certain user of the enterprise to perform a certain operation for a given purpose, this usage must be authorized. The context (operation, user and purpose) must be compared with the conditions described in the privacy policy that is configured by the consent associated with the data. The usage can then either be authorized and the obligations carried out, or else usage can be denied and potential mis-use logged for auditing purposes. The actual context information required to make the authorization decision is privacy-policy-dependent. In some cases the privacy policy applies to the entire enterprise and thus no user information is required. In other cases, additional context information, for example, about the country in which the operation takes place, might be required.

*Cross-enterprise transfer (7-11):* Transfer of personal data is a special type of usage in which there is a protocol between two enterprises. The sending enterprise first verifies whether the personal data may be transferred to the recipient. If the privacy policy and consent allow such a transfer, the data is transferred together with the original privacy policy and consent or else with a refined privacy policy and consent that adhere to the original privacy policy and consent (see [3, 2, 15] for a policy management framework for disclosures). For proper disclosure management, additional privacy metadata is needed:

*Sources*: Organizations where data has been obtained
*(Recipient,Policy',Consent')*: Recipients plus governing policies and consents

If an enterprise wants to use data for a purpose that has not been consented to during collection, it can recursively request consent from the party from whom the data was obtained (8 & 9). If a collecting enterprise has promised complete deletion, then it can trigger recursive deletion of the data by requesting deletion to all data recipients and then deleting its own data after obtaining appropriate acknowledgements (10 & 11).

## 4 Privacy Injector

In this section we introduce Privacy Injector, which leverages the AOSD paradigm to modularize and encapsulate privacy enforcement. This allows us to add privacy enforcement functionality late in the software development cycle or even to existing applications. Privacy Injector builds upon the idea behind context-sensitive string evaluation (CSSE) [24], which defines a metadata tracking and validation system used for preventing injection attacks. This paper shows how we can leverage and extend this idea to create a practical privacy enforcement. Privacy Injector consists of two complementary parts: a *privacy metadata tracking* part and a *privacy policy enforcement* part. The former is a practical implementation of the aforementioned sticky policy paradigm, whereas the latter is responsible for the actual enforcement of the sticky policies.

### 4.1 Privacy Metadata Tracking

Three components make up the privacy metadata tracking part: the *Privacy metadata assignment* component is responsible for assigning the appropriate privacy metadata to data that enters the system. The *metadata-preserving data operations* component is responsible for preserving and updating this privacy metadata when operations are performed on it. And the *metadata persistence* component is responsible for preserving, restoring, updating or removing privacy metadata when data is made persistent, retrieved, modified or removed, respectively.

Note that these components are mostly application- and enterprise-independent, and thus only need to be developed once per execution platform (e.g., Java, .NET) and can be used on a wide variety of applications and enterprises. Application- or enterprise-specific customizations are possible through configuration settings. In the remainder of this section, we describe the privacy policy-tracking components in more detail.

**Privacy Metadata Assignment** The initial step in a policy-tracking system is the assignment of privacy metadata to personal data. In Privacy Injector, this is the responsibility of the privacy metadata assignment component and performed upon entry of personal data into the system through one of the input vectors. Typical input vectors include parameters from web requests, direct input, web services requests, e-mail, etc.

The assignment of the privacy metadata is achieved through instrumentation of the API functions exposed by the execution platform responsible for the input vectors. For example, by instrumenting the API functions for retrieving the contents of cookies contained in web requests, we ensure that all data returned by these functions will have the appropriate privacy metadata assigned to it. Input received from persistent storage is treated separately by the metadata persistence component.

The privacy metadata assignment component operates fully automatically; therefore it relies on a configuration file that specifies for each input vector the conditions under which certain privacy metadata is to be assigned to data of this input vector. A simple policy could specify that no user-provided web request parameters with the name "credit card" are to be made persistent. A more complex policy could add that the restriction is only applicable under certain conditions, e.g., if the request originates from a particular country. The relevant conditions depend on the input vector. A conservative default can be specified in case some input does not match the specified conditions.

As implementing privacy metadata assignment using traditional software development methodologies requires code scattered throughout the applications, it can be seen as a cross-cutting concern that is suited for modularization through AOSD. AOSD allows us to modularize this functionality into advice and pointcuts: the former contains the actual program code responsible for the policy assignment, whereas the latter describe how the input vectors are to be intercepted and instrumented with the advice.

The location and representation of the privacy metadata are implementation choices. Conceptually, the policy travels together with the personal data. In practical implementations, however, the privacy metadata can be either stored in a system-wide policy repository or truly be part of the data. The representation of the privacy metadata is also an implementation issue, and, depending on the needs, it is possible to assign arbitrary privacy metadata containing any combination of data (e.g., consent data, data subject) and privacy policy in either declarative or programmatic form.

**Metadata-Preserving Data Operations** During its life cycle, data undergoes a chain of operations that normalize and transform it into the desired form. A privacy metadata tracking system has to ensure that the privacy metadata assigned is not lost but correctly updated when such operations are performed on the data. In Privacy Injector, this is the responsibility of the metadata-preserving data operations component.

To achieve this, all data-manipulation operations are intercepted and instrumented to update the privacy metadata to reflect changes resulting from the operations on the data, i.e., to make them "metadata-preserving." For example, when two strings are concatenated, the privacy metadata of the resulting string will have to reflect to which policy its different fragments adhere. Note that we intercept data operations at the level of the primitive data types of the execution platform.

For this paper, we assume that personal data is stored in strings, and concentrate on the string-level representation of data and corresponding string-manipulation functions. This is not an inherent limitation, and our method can equally be applied to other data types and their corresponding data-manipulation functions. In the case of strings, we assign privacy metadata per string fragment, as opposed to per string as a whole. This allows fine-grained specification of which fragments adhere to which policy and hence execution of the checks defined by the policy on only the relevant fragments.

Different transformation operations will yield different effects on the privacy metadata. Some operations, for example, changing the case of textual data, will have no influence on the privacy status of the data and such operations do not affect the privacy metadata. Other operations, for example anonymization [29], yield data that is no longer personal and thus privacy metadata needs to be removed or updated to reflect this. Yet other operations will result in a more complex interaction with the privacy metadata. An example of a very interesting and common operation is the merging of data.

There are two complementary approaches for handling the metadata of merged personal data. The first is fine-grained policy association, in which different policies are associated with the individual parts that constitute the data. The second approach is policy algebras, in which the appropriate merged metadata is calculated. When no accurate policy algebra is known for the operation, a conservative approach is to have the merged data governed by the policies of all input data. In the exceptional case of contradictory policies the most conservative action has to be selected or human intervention has to be requested.

**Metadata Persistence** A particularly important data operation is persistence, most commonly in relational databases. Privacy Injector specifies a metadata persistence component responsible for preserving, restoring, updating or removing privacy metadata when data is stored, retrieved, modified or removed, respectively. One possibility to implement this component to use a technique similar to the other components, namely, the interception and instrumentation of the appropriate persistence functions. Another related technique is SQL rewriting as used in Hippocratic databases [1]. However, implementing either of these techniques is a daunting task as it would require parsing and syntactical analysis of each SQL query to ensure correct privacy metadata persistence.

As an alternative, we propose a new technique that leverages the event system exposed by persistence services. The goal of such services is to abstract data persistence away from the applications. Applications therefore no longer perform SQL queries directly, but rely on the persistence service to store, retrieve and update their objects. The developers only describe the mapping of a particular object to database tables, and it is the responsibility of the persistence service to perform the actual mapping between the objects and the database. The best known persistence service is Hibernate [26].

The proposed metadata persistence component builds upon the fine-grained event systems exposed by persistence services. Upon storing an object in the database, the metadata persistence component will receive an event and examine the privacy metadata of the object. When indeed privacy metadata is attached to this object, it will also be made persistent. Similarly, when an object is restored, updated or removed, a corre-

sponding event will be sent to the metadata persistence component, which will respectively restore, update or remove the persisted privacy metadata.

This rather unconventional use of the event system exposed by persistence services allows us to turn the difficult problem of policy persistence into a more manageable one as no parsing or syntactical analysis of SQL queries is required. On the other hand, this comes at the expense of a reduced applicability of our method. We believe, however, that the current trend towards the use of persistence services for enterprise applications will continue as such persistence services themselves are rapidly becoming more mature and the resulting applications prove more flexible and easier to maintain.

### 4.2 Privacy Policy Enforcement

The privacy policy enforcement part ensures that the appropriate tests and actions specified by the privacy policy are performed upon usage and disclosure of the data. The different ways in which data can be disclosed are called output vectors, which are again intercepted and instrumented with the policy-enforcing functionality. For example, when an application calls the API for sending e-mail, this function is intercepted and the required conditions and obligations, as described by the privacy metadata attached to the function's parameters, are checked and the necessary actions performed.

Privacy Injector is policy-agnostic and only responsible for ensuring that the specified policy is notified of all relevant uses of the personal data and provided with the correct contextual information. The policy itself is executable, and can either be programmatically defined or interpret a declarative privacy policy configured during the privacy policy assignment.

As such, the conditions and actions supported are only limited by the expressiveness of the programming or privacy policy language used. Conditions are typically specified in function of the context of the usage or disclosure (e.g., usage/disclosure type, time of day, recipient, ...) and the attached privacy metadata (e.g., data subject, consent, type of data, ...). Practically, actions are mostly limited by their effect on the target application. Typical actions include logging the request, blocking the request (e.g., by throwing an exception), notifying the data subject, delaying the request while asking for additional consent, etc. Yet another possible action is changing or removing the personal information being disclosed, e.g., obscuring all but 4 digits of a credit-card number. This can however impact the application in unforeseen ways, and should thus be done with utmost caution.

## 5 Prototype Implementation

In this section we introduce a prototype implementation of Privacy Injector and focus on the technical aspects of implementing the concepts introduced in Section 4. Implementing a complex system such as Privacy Injector involves making several important and less important design decisions. Although we describe many of the particular design decisions we made in the prototype, the goal of this section is not to convince the reader that these are the best ones possible. The goal is rather to demonstrate the practical feasibility of Privacy Injector, to learn and find limitations, and finally to provide a feeling of how a production system might look.

### 5.1 Architecture

Our prototype targets Java applications and is implemented in AspectJ; it consists of a combination of Java classes and aspects that together form a general-purpose and highly flexible privacy enforcement framework. In its current state, the prototype does not include support for metadata persistence.

All interception points (input vectors, string operations, and output vectors) target API calls of the applications to the underlying Java platform, and the prototype can thus be seen as a layer between these two. An advantage of this choice, compared with instrumenting the platform itself, is that this API layer is standard over all platform implementations. This make our prototype compatible with all Java implementations.

The datatype targeted is strings for the reasons discussed in Section 4.1. Privacy metadata (PM) are objects that contain the privacy context as well as the privacy policy. Strings and PM objects are linked together by means of a central repository.

The focus is on extensibility rather than completeness. For example, only a few input and output vectors are implemented, but implementations for more can easily be plugged in. The prototype does, however, contain its own minimalist policy language that allows full declarative configuration of the prototype for many common tasks.

We distinguish between four components, which will be detailed further below: configuration, PM assignment, PM preservation, and policy enforcement. These components map largely to the conceptual components of Section 4.

### 5.2 Configuration

The prototype supports configuration through an XML-based configuration file, which enables most common and many less common tasks be achieved without programming. At the same time, it also supports flexible programmatic configuration through user-provided interceptors for input and output vectors, or specialized PM factories, i.e., classes that generate PM objects.

Figure 2 depicts the set of Java classes that form the configuration component and the three steps that make up the configuration phase: reading of the configuration file, initialization of the specified PM factories, and registration of the PM factories with the relevant input vectors.

The first step entails reading and parsing the XML-based configuration file. This configuration file specifies the PM factories and their initialization context, as well as the conditions under which data received from certain input vectors is assigned PM from these factories. An example configuration file looks as follows:

```
<PIConfiguration>
  <PM id="onlyLocalEmail" factory="pi.pmfactory.Generic">
    <!-- initialization context -->
  </PM>
  <PMAssignment>
    <inputvector>pi.inputVector.Http</inputvector>
    <conditions>
      <regexp target="http.requestedUrl">some regexp</regexp>
      <test>com.company.pi.RequestorTest</test>
      <regexp target="http.requestedParam">another regexp</regexp>
    </conditions>
    <PM ref="onlyLocalEmail"/>
  </PMAssignment>
</PIConfiguration>
```

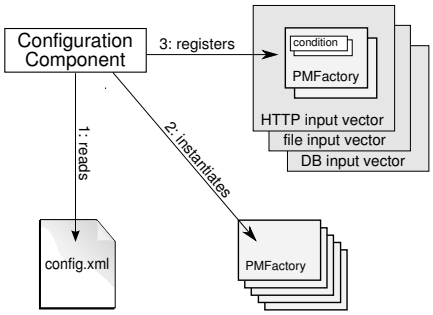


**Fig. 2.** Configuration phase of the prototype.

The second step is to initialize the specified PM factories described in the <PM> elements of the configuration file. A PM factory is a Java class that follows the factory design pattern [11] to create specialized PM objects. In this excerpt one PM factory of type "pi.pmfactory.Generic" and with id "onlyLocalEmail" is specified. The PM element also contains an optional initialization context (not shown here), which will be discussed as part of the PM assignment phase.

The third and final step is to configure the actual PM assignment by linking the different PM factories with the relevant input vectors. This link is described in the <PMAssignment> elements of the configuration file and contains a reference to the relevant input vector, the conditions under which PM should be assigned, and a reference to the PM factory responsible for creating the PM objects. Two types of conditions are supported: regular expressions on the context exposed by the input vector (e.g., the requestedURL in case of the HTTP input vector) and arbitrary user-provided programmatic tests. These conditions are then initialized (e.g., regular expressions are pre-compiled for efficiency) and together with the PM factory registered at the interceptor for the input vector specified.

After these three steps, Privacy Injector is fully configured, and assignment of PM can commence.

### 5.3 PM Assignment

The PM-assignment component is responsible for assigning the specified PM objects to data received by the target application through one of its input vectors. As this requires the ability to intercept input-vector API calls made by the target application, the PM-assignment component consists of AspectJ aspects rather than plain Java classes. The core part is an abstract aspect that is subclassed by concrete aspects, which are organized per input vector and responsible for the actual interception and PM assignment. The framework can easily be extended further by plugging in a new aspect targetting the desired input vector. Figure 3 shows the four steps involved: interception of the input, validation of the conditions, creation of the PM, and assignment of the PM to the data.

The interception step is driven by the pointcuts and advice (cf. Section 2.4) declared in the aspects. The pointcuts specify join points for all the relevant

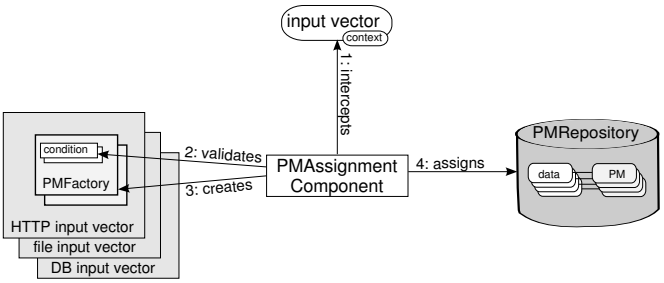


**Fig. 3.** Privacy metadata assignment phase of the prototype.

methods pertaining to the input vector. For the HTTP input vector, on which is the prototype focuses, this means all methods for extracting data from an HTTP request object. For example, `javax.servlet.Servlet.getParameter()` or `javax.servlet.http.Cookie.getValue()`. The advice used is so-called *after advice*, which is executed after the call to the API function returns, and is capable of inspecting the returned value before it is passed on to the calling application.

Each input vector has a (possibly empty) set of PM factories that were registered with it during the configuration phase. In the second step the conditions associated with these PM factories are validated. These conditions are typically tests on the context exposed by the aspect, i.e., on the set of parameters relevant to that particular input vector. For the HTTP input vector, such a condition could be related to the resource requested or to the authentication status of the requester. Only if all conditions of a PM factory hold, is the third step performed on that PM Factory. Otherwise, control will be returned to the application without assigning PM.

In the third step, the actual PM object is created by calling the creation method of the PM factory. A PM factory is a factory that creates objects that subclass the PM class. The configuration file can specify an arbitrary user-provided factory or the generic one as in our example. The latter provides less flexibility, but requires only configuration and no programming from the user's part. The configuration excerpt shows the PM factory initialization context used to configure the generic factory:

```
<PM id="onlyLocalEmail" factory="pi.pmfactory.Generic">
  <context>
    <param name="inputVector"/>
    <param name="http.requester" as="dataSubject"/>
    <param name="http.requestTime" as="timestamp"/>
    <text as="comment">some comment</text>
  </context>
  <policies>
    <!-- description of policies -->
  </policies>
</PM>
```

This initialization context is passed on to the PM factory during its initialization in the configuration phase. The syntax of the initialization context is PM-factory-specific; shown here is the syntax for the generic PM factory. The initialization context contains two parts: the context, which describes the metadata that should be maintained, and the

policies, which describes the policy (cf. Section 5.5). In our example, the generic PM factory is configured to maintain four pieces of metadata in the PM objects it generates: the input vector, the requester, the time of the request, and a comment. The `as` specifies the name under which the metadata is accessible.

In the final step, the PM generated is packaged into a PM container and stored in the central PM repository. A PM container is a data structure that allows one to associate multiple PM objects with possibly overlapping string fragments. It also provides an API for convenient and efficient lookup and manipulation of the associated PM. This PM container is then added to the central PM repository, which is essentially a weak hash table enabling efficient lookup through an specialized API. By using a *weak* hash table [3] the PM of data that is no longer in use will be removed automatically.

After this step all input data that fulfilled the specified conditions will have the appropriate PM associated with it.

### 5.4 PM Preservation

The goal of this phase is to preserve and update the PM assigned in the assignment phase. Efficiency is a major concern of this phase as almost every string operation is affected, even if none of the operands contain PM. And, as in a typical application only a small fraction of the strings has PM attached, special care should be taken to make operations on these strings particularly low-overhead. Figure 4 shows the four steps of the PM preservation phase: interception of the data operations, retrieval of the PM, updating of the PM, and storage of the PM.

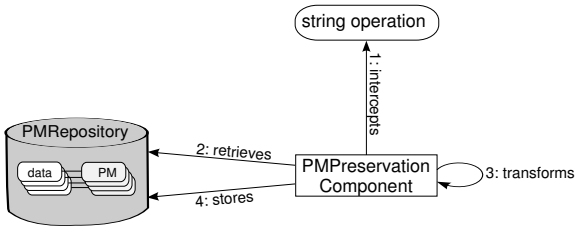


**Fig. 4.** Metadata-preservation phase of the prototype.

In the interception step, all relevant string operations are intercepted. Java has two types of string-like data, String and StringBuffer, with the difference between the two being that the former is immutable whereas the latter is not. This means that the relevant operations are operations that have at least one stringbuffer operand or that return a string and have at least one string operand. The PM preservation component provides pointcuts for all calls to these functions.

In the second step, the PM of all operands is retrieved from the PM repository. The repository has an API for doing this, requiring only one efficient hash table lookup per operand. If no PM is found, control is returned to the application.

[3] A special type of hash table whose elements do not count as referents for the garbage collector.

The third step requires by far the largest development effort as it needs the ability to reflect the semantics of all relevant string operations on the PM correctly. For our prototype, we initially focused on the most common operations. Many of the operations fall into the class of operations that return string(buffers) that merely have a copy of the original PM of their operands attached or no PM at all. In the former case, the original PM is cloned, in the latter control is returned to the application. Other operations require more complex manipulation of the PM, e.g., a merge of the PM of two operands or a selection of a fragment of the PM. For this we leverage the API provided by the PMContainer class, which facilitates common operations such as retrieval of the PM of string fragments and merging of PM containers.

The final step is storing the updated PM in the PM repository.

### 5.5 Policy Enforcement

In this last phase, the actual enforcement of the privacy policy is performed. The framework is responsible for notifying and providing the context to the PM objects attached to data passed to an output vector. The actual validation of the conditions and execution of the appropriate actions are the responsibility of the PM objects.

Recall that PM objects are created by PM factories that can be either user-provided or of the generic type. Factories can contain a hard-coded programmatic policy or a declarative one, described in a factory-specific manner in the <PM> part of the configuration file. By plugging in a policy language interpreter, the framework can be extended to support arbitrary policy languages. The following XML shows an example of the configuration of the generic PM factory:

```
<PM id="onlyLocalEmail" factory="pi.pmfactory.Generic">
  <context>
    <!-- stored privacy context, discussed in PM Assignment -->
  </context>
  <policies>
    <policy>
      <outputvector>email</outputvector>
      <conditions>
        <regexp target="email.recipient">^.+@mycompany.com$</regexp>
      </conditions>
      <actions>
        <log file="/path/to/log/file"/>
      </actions>
    </policy>
    <default>
      <actions>
        <custom method="pi.Actions.pageOperator"/>
        <exception class="pi.IllegalDisclosureException">
          disclosure not in accordance with privacy policy
        </exception>
      </actions>
    </default>
  </policies>
</PM>
```

The <policy> element pertains to the email output vector, and describes which actions have to be performed under which conditions. The conditions are specified using a syntax identical to that of the PM assignment conditions and can refer to both context exposed by the output vector and metadata stored in the PM object. In the example, the

condition specifies a limitation on the domain of the email recipients. If all conditions hold, the specified action, in this case log, is performed. If for none of the specified policies, both output vector and conditions match, the optional default policy actions are performed. In the example, a custom method is executed and an exception is thrown.

Figure 5 shows the three steps of the policy-enforcement phase: interception of the output vectors, retrieval of the PM, and execution of the specified policy.

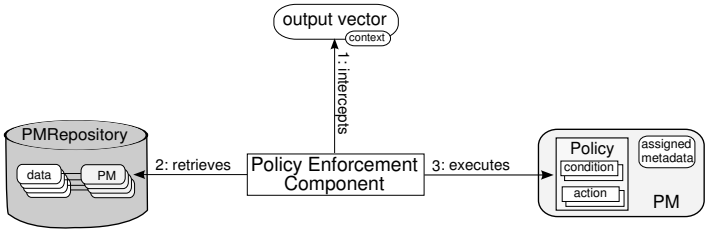


**Fig. 5.** Policy-enforcement phase of the prototype.

Our primary focus is prevention of unwarranted disclosure of personal data. The first step of the policy-enforcement step will then also be to intercept all calls to output vectors, as this is where disclosure takes place. The output vector in our example is the Java mail API. We defined aspects grouped per output vector that provide pointcuts for all calls to methods used for sending data through the output vector. The advice is so-called *around advice* that allows us to alter the program flow, for example by throwing exceptions or altering the data passed to the output vector.

The next step is to verify whether the data being sent has PM attached to it. This is done in the same way as in the PM preservation phase by checking the PM repository. If the data has no PM attached to it, control is returned to the program and the disclosure is allowed to take place unchanged.

In the final step the policy specified by the PM is executed. For this, a pre-defined method is invoked which receives the context surrounding the call to the output vector, e.g., which output vector, the parameters, etc. Using the provided context together with the metadata stored in the PM object, the appropriate actions are performed. The generic PM factory supports logging, exception throwing, and executing arbitrary commands.

## 6 Discussion

### 6.1 Assumptions and Limitations

In this section we discuss limitations and assumptions related to Privacy Injector.

The most important assumption made by Privacy Injector is the use of a persistence service by the target application. This is because, as mentioned in Section 4, the metadata persistence component relies on the event system of the persistence service for correctly preserving metadata when data is stored and retrieved. The impact of the assumption is eased by the momentum persistence services currently enjoy.

We also assume that for privacy enforcement we are mainly interested in textual data and hence that, at finest granularity, privacy-related data is contained in variables of the string type. This can be considered a pragmatic consideration that will suffice for many real-world applications. Related to this is the assumption that applications do not access strings in a low-level manner, but through the string API functions. This holds in almost all cases for high-level languages such as Java, but might not hold for C.

Privacy Injector is inherently platform-independent, but assumes a provision for intercepting library functions, either at the platform layer (instrumentation of the library functions) or at the application layer (instrumentation of the library function calls).

Finally, Privacy Injector is particularly suited for preventing unwarranted disclosure of personal data, as the focus on the limited and well-defined set of input and output vectors provided by the platform allows the creation of a reusable framework. For use-cases with other requirements, it currently remains an open question whether the desired functionality can also be made generally applicable.

### 6.2 Correctness

When implementing a privacy-enforcement system, it is reasonable to expect a certain level of guarantee that the system enforces the policy as specified. As Privacy Injector is a complex system that interacts with all components of a system, a formal correctness proof is beyond the state of the art. When assessing the possibility to create a complete and correct Privacy Injector implementation, one has to keep two properties in mind.

The first is that the Privacy Injector framework is largely reusable and requires implementation only once. This has the advantage that it can be done by experts that submit it to rigorous testing, resulting in high-quality code.

The second is that Privacy Injector can implement a fail-safe mode, by requiring that metadata be assigned to all data and failing when this assertion does not hold. When no real privacy metadata is attached to the data, a placeholder indicating that the metadata framework functioned correctly is attached. Related to this, a policy should always specify a safe default action when none of the conditions specified hold (as in the example provided in Section 5).

## 7 Conclusions and Future Work

In this paper we introduced Privacy Injector, which leverages the Aspect-Oriented Software Development paradigm to create a fully modularized privacy-enforcement system. Privacy Injector can be seen as a practical implementation of the sticky policy paradigm and conceptually consists of two complementary parts, namely a privacy metadata tracking and a privacy policy enforcement part.

We showed how these two parts together protect the entire life cycle of personal data. Throughout the collection, transformation, disclosure and deletion of personal data, Privacy Injector will automatically assign, preserve and update privacy metadata as well as enforce the privacy policy specified. Through the use of aspects, Privacy Injector can be used to add privacy enforcement to existing applications as well as a framework for building privacy-aware applications.

We also described a prototype implementation of Privacy Injector, aimed at demonstrating the practical feasibility of the concepts introduced in the paper. For this prototype we focused on Java and relied on the AspectJ flavor of the Aspect-Oriented Software Development paradigm. We focused on disclosure control to prevent unwarranted disclosure of personal data.

Currently, we are further extending our prototype to include support for metadata persistence through the use of the fine-grained event system provided by Hibernate. This will allow us to test our prototype in a real-world environment. As future work, we will add a provision for storing accounting information in the privacy metadata, providing us with the ability to keep a detailed history of all operations performed on any piece of personal information.

## References


1. Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *Proceedings of the 28th Int'l Conf. on Very Large Databases (VLDB), Hong Kong*, 2002.
2. Michael Backes, Walid Bagga, Günter Karjoth, and Matthias Schunter. Efficient comparison of enterprise privacy policies. *19th ACM Symposium on Applied Computing, Special Track Security, Nicosia, Cyprus*, 2004.
3. Michael Backes, Birgit Pfitzmann, and Matthias Schunter. A toolkit for managing enterprise privacy policies. *8th European Symposium on Research in Computer Security (ESORICS 2003), Lecture Notes in Computer Science*, 2808:162–180, 2003.
4. Claudio Bettini, Sushil Jajodia, X. Sean Wang, and Duminda Wijesekerat. Obligation monitoring in policy management. In *Proceedings of the 3rd IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, pages 2–12, 2002.
5. Piero A. Bonatti, Ernesto Damiani, Sabrina De Capitani di Vimercati, and Pierangela Samarati. A component-based architecture for secure data publication. In *Proceedings of the 17th Annual Computer Security Applications Conference*, pages 309–318, 2001.
6. Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
7. N. Damianou, N. Dulay, E. Lupo, and M. Sloman. The ponder policy specification language. *Policies for Distributed Systems and Networks (Policy 2001), Lecture Notes in Computer Science 1995*, pages 18–39, 2001.
8. S. Egelman, L. Cranor, and A. Chowdhury. An analysis of p3p-enabled web sites among top-20 search results. In *Proceedings of the Eighth International Conference on Electronic Commerce*, 2006.
9. Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
10. Simone Fischer-Hübner. *IT-security and privacy: Design and use of privacy-enhancing security mechanisms*, volume 1958 of *Lecture Notes in Computer Science*. Springer, 2002.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
12. IBM. Declarative privacy monitoring. Web page at http://alphaworks.ibm.com/tech/dpm.
13. Sushil Jajodia, Michiharu Kudo, and V. S. Subrahmanian. Provisional authorization. In *Proceedings of the E-commerce Security and Privacy*, pages 133–159. Kluwer Academic Publishers, 2001.

14. Günter Karjoth and Matthias Schunter. A privacy policy model for enterprises. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW)*, pages 271–281, 2002.
15. Günter Karjoth, Matthias Schunter, and Els Van Herreweghen. Enterprise privacy practices vs. privacy promises - how to promise what you can keep. *4th IEEE International Workshop on Policies for Distributed Systems and Networks (Policy '03), Lake Como, Italy*, pages 135–146, 2003.
16. Günter Karjoth, Matthias Schunter, and Michael Waidner. The platform for enterprise privacy practices – privacy-enabled management of customer data. In *Proceedings of the Privacy Enhancing Technologies Conference*, volume 2482 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2002.
17. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
18. Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
19. A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, pages 410–442, 2000.
20. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 228–241, 1999.
21. Oasis. eXtensible Access Control Markup Language (XACML). Web page at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.
22. Platform for Privacy Preferences (P3P). W3C Recommendation, April 2002. `http://www.w3.org/TR/2002/REC-P3P-20020416/`.
23. David L. Parnas. On the criteria to be used in decomposing systems into modules, 1972.
24. Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID2005)*, pages 124–145, 2005.
25. AspectJ Project. The AspectJ home page. Web page at http://eclipse.org/aspectj/.
26. Hibernate Project. Hibernate. Web page at http://hibernate.org/.
27. Carlos Ribeiro, Andre Zuquete, Paulo Ferreira, and Paulo Guedes. SPL: An access control language for security policies with complex constraints. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2001.
28. A. Sabelfeld and A. Myers. Language-based information-flow security, 2003.
29. Latanya Sweene. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002.
30. Watchfire. Watchfire. Web page at http://watchfire.com/.
31. Bart De Win, Frank Piessens, Wouter Joosen, and Tine Verhanneman. On the importance of the separation-of-concerns principle in secure software engineering. In *Proceedings of the ACSA Workshop on the Application of Engineering Principles to System Security Design*, pages 1–10, 2003.