Language-Based Enforcement of Privacy Policies

Katia Hayati and Martín Abadi

Department of Computer Science
University of California, Santa Cruz

Abstract. We develop a language-based approach for modeling and verifying aspects of privacy policies. Our approach relies on information-flow control. Concretely, we use the programming language Jif, an extension of Java with information-flow types. We address basic leaks of private information and also consider other aspects of privacy policies supported by the Platform for Privacy Preferences (P3P) and related systems, namely the notion of purpose and the retention of data.

1 Introduction

Entities with a Web presence should not only define and publish their privacy policies but also ensure that they comply with those policies. A recent online survey [2] conducted by the Privacy Place [3] indicates that users may not mind when a website uses their personal information to tailor their browsing, but that they care about the possible misuse of this information and support punishments for misbehaving websites.

The problem of enforcing privacy policies has recently been attacked from several angles and in various domains (general enterprises [6], financial institutions [4], etc.). However, there have been no solutions at the level of programming languages. A language-level modeling of privacy policies should help programmers in avoiding inadvertent implementation mistakes. It should also facilitate the auditing of code by independent entities.

In a different context, there has been much work on restricting flows of information in programs (e.g., [9]), and on programming languages that support such restrictions (e.g., [13, 18, 19, 20, 21]). These restrictions can serve for guaranteeing integrity and secrecy properties. Although secrecy and privacy are related, it does not seem straightforward to reduce privacy policies (of the kinds considered in the privacy literature, and used in websites and elsewhere) to standard secrecy properties. Moreover, while privacy concerns are sometimes mentioned in some papers on those programming language, the papers do not show how to apply their techniques to enforcing privacy policies.

In this paper, we start to bridge the gap between those two lines of work. We explore how to use an information-flow system embodied in a programming language for guaranteeing that programs abide by their stated privacy policies. We address basic leaks of private information and also consider other facets of privacy policies.

In order to ground our work, we base our concepts of privacy on the Platform for Privacy Preferences (P3P) [23]. Although the P3P project is a young one, it has generated much interest and ongoing research (for example, about its implementation [1]). The goal of the P3P effort is twofold: to allow websites to specify their privacy policies concisely and precisely, and to enable users to specify their privacy preferences in order to check them automatically against the published policies of websites. A P3P file is an XML document that describes which information a website collects, what it intends to do with it, and how long it will be kept. A P3P file should also describe a way for a user to resolve a conflict with the website (for example, if the user believes information was mishandled). However, P3P is limited in scope. In particular, it is outside its scope to verify that websites really do abide by their stated policies in the implementation of their Web applications.

As a programming language, we use Jif [16, 17, 18]. It is an extension of Java that includes an enriched type system for specifying and checking information-flow security properties. It has not previously been used for providing privacy guarantees in a systematic way, but we believe that it is quite attractive for this purpose.

We show how we can use Jif for modeling and verifying aspects of privacy policies. Specifically, we consider three aspects of privacy: data cannot be (inadvertently) leaked; data is used at most for the purpose for which it was collected; and data is retained no longer than promised. These aspects constitute a core subset of the notions addressed by P3P. Of course, these notions are not specific to P3P privacy policies. They are also present, with variants, in other contexts, even beyond the Web.

The rest of this paper is organized as follows. Next, in Section 2, we present some further background material. In Section 3 we show how Jif can be used to give basic privacy guarantees. In Section 4 we explain how to represent purposes with principals. In Section 5 we treat retention guarantees. We conclude in Section 6.

2 Background

The problem of making it easier to control how private information is handled has been considered from many perspectives. Dreyer and Olivier [10] describe a system based on graph theory where entities are vertices in a graph and information flow between entities is represented by edges between vertices. In order to determine whether an inadmissible flow occurs, they use a graph reachability algorithm. Ashley, Powers, and Schunter [6] describe a system in which privacy information is attached to data. He and Antón [12] describe a system based on role engineering for modeling privacy policies. The authors also discuss P3P and lattices of purposes. Antón and her collaborators deplore the lack of a solution coming from the security sphere (see for example He's technical report [11] in addition to the previously cited paper). Privguard [15] is a system for protecting private data based on the purpose for which it was collected. It uses encryption

to achieve security. The Enterprise Privacy Authorization Language (EPAL) in development at IBM [5] is an alternative to P3P. IBM has also developed the Declarative Policy Monitoring [8] and Reference Monitor technologies [14], designed to provide programming support for the enforcement of privacy policies written in EPAL or P3P. The enforcement appears to be done dynamically (for example in LDAP sniffers) and not at the language level. In addition to these projects, we are aware of some nascent efforts in this area (such as PAW [22]).

Type-based information-flow analysis for programming languages is a rich field, of which Jif is a prominent example. Work in the field is concerned with enforcing integrity and secrecy properties at the level of programs, relying on programming-language support. For example, Palsberg and Ørbæk [19] develop a λ-calculus with explicit trust operations, and equip it with a trust-type system; the SLam calculus [13] is a λ-calculus in which types express both integrity and secrecy properties.

While some of the foundational research in this area applies only to foundational calculi or "toy" programming languages, the techniques developed carry over to powerful, general-purpose programming languages. These include Jif (which, as mentioned in the introduction, is an extension of Java) and also Flow Caml (an extension of ML) [20, 21].

Jif provides mostly static information-flow checking via a type system based on the Decentralized Label Model [18]. The programmer must annotate variables, methods, and class declaration with a *label*. (Jif does not force the programmer to annotate every single variable: Jif infers labels not explicitly declared, and sets them to be as restrictive as possible.) A label specifies who owns data and who can read it. For example, the label {`Alice:`} means that `Alice` owns the data and only `Alice` can read it, and {`Alice: Bob`} means that `Alice` owns the data and `Bob` can read it too. The entities that own and read data are called *principals*. Principals are first-class objects (in the sense that they may be passed around). They are related to each other by the *acts-for* relation. If `Alice` acts-for `Bob`, then `Alice` can do everything `Bob` can do. The acts-for relation is reflexive and transitive. In addition, Jif supports a `declassify` operation, which enables the owner of a piece of data to give it a less restrictive label in certain circumstances.

3 Basic control of information leaks

At a very basic level, Jif can be used to ensure that sensitive data is not leaked. For this purpose, we can represent categories of data (such as categories of sensitive data) with principals.

As an example, let us consider two principals, named `SecretUserData` and `SharedUserData`, and assume that `SecretUserData` acts-for `SharedUserData`. Then anything owned by `SharedUserData` is accessible by `SecretUserData`, but the converse is not true. Thus, while secret data may depend on shared data, leaks of secret data into shared data can be caught, as illustrated in the following code fragment:

```
// This code does not (and should not) compile.

int{SecretUserData: } credit_rating = 3;
// owned by SecretUserData and readable by no one else.

int{SharedUserData: } rebate;
// owned by SharedUserData, and can be accessed
// by SecretUserData.

if (credit_rating > 4) {
  rebate = 1;
  // ERROR: the (visible) value of rebate depends on the
  // (supposedly secret) value of credit_rating.
} else {
  rebate = 0;
}
```

This code fragment does not compile in Jif, because the value of `rebate` depends on the value of `credit_rating`. This dependency constitutes an inappropriate flow of information. If the value of a public variable depends on the value of a secret variable, then by observing the output of a program a non-authorized entity could infer information about the secret data.

As this example demonstrates, some of the most basic privacy properties can be supported by the Jif type system. In more elaborate examples, finer-grained data classifications can be used to indicate the intended recipients of a piece of data and other additional information. In any case, with Jif, the programmer has fewer worries that correctly labeled data will flow in unexpected or forbidden ways.

4 Purpose

In this section, we tackle the problem of modeling purposes in Jif. First, we review the definition of the notion of "purpose". Then we discuss how to represent purposes with Jif principals. Finally we briefly discuss the assurance problem: how can we make sure a program does what it promises to do?

4.1 What is a purpose?

Data is collected to fulfill a *purpose*. A purpose describes what the system intends to do with a piece of data. Examples include "tailoring the homepage of a website to the tastes of a particular client", "enabling a third-party shipping service to ship the goods to the client", "providing adequate medical care to a patient", and the like.

A purpose should be interpreted as an "upper bound", so the goal of a verifier is to make sure that the system does *at most* what it promised to do

with the piece of data. To clarify this point, consider the example of an online bookseller that collects the user's mailing address for the purpose of shipping the purchased goods to the user. Then it is acceptable if the website actually does nothing with the address. For example, the user might have entered his address but then changed his mind about the particular purchase. On the other hand, the bookseller should not be allowed to do *more* than it promised. For example, it should not be allowed to sell the user's address to a telemarketing company.

Purposes can have a hierarchical structure. For example, the purpose of "traffic analysis" can be subsumed under the purpose of "website administration". Therefore, if a piece of data was collected with the purpose of helping with website administration, the system should be allowed to use it for the specific sub-purpose of "traffic analysis". The opposite, however, should not be true.

The P3P specification describes eleven specific purposes, and one catch-all "other" purpose which must be accompanied with a human-readable description. However, the notion of purpose can be more general than allowed in the P3P definition (which, for example, does not talk about sub-purposes). The model that we propose can adequately handle the P3P notion of purpose, and it is also powerful enough to describe a collection of purposes with sub-purpose relations, more broadly.

4.2 Modeling purpose with principals

The model We choose to represent a purpose with a principal in Jif. The programmer must create a principal for every purpose found in the policy. Data which is collected for a specific purpose is annotated as being owned by the corresponding principal. Methods which are needed for a specific purpose are annotated as bearing the authority of the corresponding principal.

This modeling achieves a number of goals. First of all, it ensures that correctly labeled data is going to be used only by the principals that have been explicitly granted authorization to use it. It also enables the programmer to make the program more explicit, as the purpose of methods is declared alongside them. In practice, the programmer does not need to annotate every single method, as Jif does some type inference. When type information is missing, type inference aims to find the most conservative label.

Consider the following code fragment. It shows a slightly more involved example than that of Section 3, and illustrates again how data cannot be misused. In this example `WebAdmin` and `Marketing` are two unrelated principals.

```
class LogProcessor {

  // the return type of total_hits is an int which
  // should be owned by WebAdmin and readable
  // by no one else.
  public int{WebAdmin: } total_hits() {
    ...
  }
```

```
  ...
}
...
int{Marketing: } hits = (new LogProcessor(...)).total_hits();
// ERROR: the label of hits is incompatible with
// the return label of total_hits().
}
```

An error is raised at compile time, preventing that data for `LogAmin` be made accessible to `Marketing`.

Sub-purposes Sub-purposes are also easy to model via the acts-for relation. For example, if `LogAdmin` is a sub-purpose of `Admin`, then we can let the principal `LogAdmin` act-for `Admin`. So if a piece of data is collected for the (generic) purpose of `Admin`, it may in particular be used for the purpose of `LogAdmin`.

Suppose we have defined these two principals. We can then write the following code fragment:

```
InetAddress{Admin: } client_address = ...;
...
int{LogAdmin: } client_uid = client_address.hashCode();
```

The second assignment is legal even though `LogAdmin` uses a variable owned by `Admin`.

Multiple purposes Multiple purposes can be understood as another facet of sub-purposes. We can use the acts-for relation again to model data which is collected for several different purposes. The following construction is similar to the modeling of groups and roles in the Decentralized Label Model [18].

As an example, consider the purposes `LogAdmin`, `WebAdmin`, and `Admin` introduced above. Both `WebAdmin` and `LogAdmin` are sub-purposes of `Admin`, so we let both `LogAdmin` and `WebAdmin` act-for `Admin`. Suppose that the variable `client_address` is collected for both `LogAdmin` and `WebAdmin` purposes. Then we can label `client_address` with `Admin`, and it will be usable by both `LogAdmin` and `WebAdmin`.

Other examples may combine apparently unrelated purposes, such as the purpose `Marketing` defined above and the purpose `WebAdmin`. Suppose that we thought that a datum `unique_hits` should be used for both of these purposes. We can construct a new principal `Marketing_or_WebAdmin`, let `Marketing` and `WebAdmin` act-for the new principal, and use `Marketing_or_WebAdmin` in the annotation for `unique_hits`. In Jif this would not involve changing the already existing definition of `Marketing` or `WebAdmin`, because the superiors of a principal (those who act-for the principal) are declared alongside it.

While in principle one could construct an exponential number of compound purposes from a set of basic purposes, only those necessary for a particular

program would have to be declared and used in that program. We expect these tasks to be of manageable complexity.

Conditional purposes Next, we discuss the situation (absent in P3P but present in other contexts [12]) where a piece of data `val` was collected for a purpose `P` but could be used for a purpose `Q` under certain well-defined circumstances. In Jif, `val` can have the label {`P:` }, but `P` can use the `declassify` operation when necessary.

The following code fragment illustrates this concept:

```
class Log authority(LogAdmin) {
// The class is annotated as bearing LogAdmin's authority.

  double{LogAdmin: } computeAvg(...) {
    ...
  }

  int{LogAdmin: } orderOfMagnitude(double{LogAdmin: } x) {
    ...
  }

  int{LogAdmin: Marketing} showAvg() throws SecurityException
    where authority(LogAdmin) {
    // The method is annotated as bearing LogAdmin's authority
    // so it is allowed to declassify data owned by LogAdmin.

    int{LogAdmin: } magn = orderOfMagnitude(computeAvg(...));

    if (certain_well_defined_circumstances) {
      return declassify(magn, {LogAdmin: Marketing});
      // returns magn with the new label
      // {LogAdmin: Marketing}.

    }
    else {
      throw new SecurityException();
    }
  }
}
```

The places in a program where the `declassify` operation appears constitute clear targets for detailed auditing. Thus, although the use of `declassify` may weaken the guarantees that can be established at the programming-language level, a Jif program with a few careful declassifications offers a better basis for enforcement than an arbitrary Java program.

Roles and purposes The principals that we use for representing purpose are regular principals in Jif. Myers and Liskov [18] have shown how to use principals and the acts-for relation to model *roles*. A principal may have several roles in an organization and wish to keep them separate. Roles are important in other contexts [12], and this modeling of purposes could be combined with the straightforward expression of roles to yield a richer system.

4.3 Assurance

It is important to realize that Jif, with our representation of purposes, will *not* guarantee that principals will perform only those actions that are necessary for the declared purposes. For example, we obtain no guarantee that a principal called `WebAdmin` will only administer a website, nor that a principal called `Statistics` will perform only statistical actions.

Nevertheless, information-flow checking does help. It can reduce the size of the code that needs to be examined in order to ensure that data is used only for the declared purposes.

In order to achieve higher assurance, one may combine the formal reasoning of the type system with a statement from the programmer (or some other responsible entity) certifying that the code does what it is supposed to do, and no more. This statement may in particular address any use of declassification operations. Ideally, it would be accompanied with a formal proof.

5 Retention

Another dimension of privacy is controlling how long user data may be retained. Although Jif does not have a built-in mechanism for expressing time or retention, in this section we show a treatment of retention that works within the existing Jif label system. We complete the section with another brief discussion of assurance.

5.1 Retention periods

Our treatment addresses P3P's retention model, which defines five possible retention periods: `no-retention`, `stated-purpose`, `legal-requirement`, `business-practices`, and `indefinitely`. The label `no-retention` means that the data should be used only to complete the current action and should never be stored. Similarly, `stated-purpose` indicates that the data should be destroyed once the purpose for which is was collected is finished. This period could be longer than just the current transaction. For example, if a website collects a mailing address to share with an expediter, it might take a while to communicate the address to the expediter and complete the shipping. The retention `legal-requirement` indicates that the data will be retained for as long as is required by law. This period can vary, but for example the law sometimes requires banks to hold financial information for a year. The annotation `business-practices` is similar in intent, but here the considerations pertain to business rather than the law. The

annotation `indefinitely` is the least restrictive: it indicates that the website may keep the data for any amount of time, but imposes no requirement.

It is of course possible that dealing with other models of retention would require changes and extensions to our approach.

5.2 Retention labels

We can relate the idea of retention to information flow. A datum that may be retained only for a short time should not influence a datum that is retained for a very long time. For example, there should be no information flow from a variable with the label `no-retention` to a variable with the label `legal-requirement`.

Using this idea, we can represent retention periods within the existing Jif label system. We represent them as principals, much in the same way as purposes. Next we illustrate this encoding through an example; other encodings are possible.

In our example, `total_page_views` and `credit_rating` are variables intended for long-term retention, while `temp_cookie`, `credit_report_cookie`, and `viewed_credit_report` are variables intended for short-term use only (perhaps simply for displaying one webpage to the user). We omit the code that initializes the variables. For simplicity, we assume that these variables are all for the purpose of log administration, represented by the principal `LogAdmin`.

We introduce a principal for each retention period. In the example, we have two such principals, named `NoRetention` and `Indefinitely`, and we are concerned with preventing information flows from `NoRetention` to `Indefinitely`. Since flows in the opposite direction are admissible, we let `NoRetention` act-for `Indefinitely`.

Each of the variables has two owners: the purpose principal `LogAdmin` and a retention principal (one of `NoRetention` and `Indefinitely`). Labels with multiple owners, such as these ones, are supported in Jif. Intuitively, a component of a label with an owner `A` indicates `A`'s policy. A label with multiple owners can be understood as the conjunction of the policies of all the owners.

```
int{LogAdmin: ; NoRetention: } temp_cookie;
boolean{LogAdmin: ; NoRetention: } viewed_credit_report;
int{LogAdmin: ; NoRetention: } credit_report_cookie;
int{LogAdmin: ; Indefinitely: } total_page_views;
int{LogAdmin: ; Indefinitely: } credit_rating;

// This assignment is OK (this ‘‘if’’ block typechecks).
if (viewed_credit_report) {
  credit_report_cookie = NO_SHOW_AD;
}
else {
  credit_report_cookie = SHOW_AD;
}
```

```
// This block typechecks too.
actsFor(NoRetention, Indefinitely) {
  // This ‘‘if’’ block executes only if it is the case that
  // NoRetention acts-for Indefinitely.
  // The reason we have to add a runtime check of this fact is
  // that acts-for relationships may change.
  if (credit_rating > 5) {
    temp_cookie = 1;
  }
  else {
    temp_cookie = 2;
  }
}

// ERROR: short-term information used in
// long-term variable.
if (viewed_credit_report) {
  total_page_views++;
}
```

Here, the value of `viewed_credit_report` is not allowed to influence the value of `total_page_views`, which may be kept indefinitely. Thus, the information-flow analysis addresses both purposes and retentions, simultaneously and independently.

5.3 Assurance

Much as for purposes, the Jif type system offers helpful support for retentions, but no actual "real-world" guarantees. For example, Jif does not have any independent information on the legal requirements associated with the label `legal-requirement`, and the Jif type system need not forbid storing data with the label `no-retention` on disk.

On the other hand, many such difficulties may be prevented if retention labels are correctly associated with system interfaces. In particular, the file-system interface could simply prevent the writing of data with the label `no-retention`. Thus, it may be possible to guarantee that data with the label `no-retention` is indeed ephemeral.

6 Conclusion

In this paper we present an approach for modeling and verifying some privacy properties in Jif, a programming language with an information-flow type system. We show how purposes and retention periods, in the sense of P3P, may be represented in Jif. We believe that this approach is rich enough to support

additional privacy properties. In particular, we have developed some preliminary techniques for expressing the anonymous use of data.

So far we have focused on the checking of specific privacy properties on small pieces of code. We have not considered how our approach could apply to large software-engineering projects; we can only speculate on this question. Neither have we considered how those properties are assembled and expressed as a full policy. This policy may be written in P3P or a similar language, but it could also be represented by a Jif interface (analogous to a Java interface). In this case, the problem of checking compliance with the policy reduces to Jif typechecking. For other policy languages, the problem of checking compliance may also be tractable provided those languages are given a precise semantics.

Acknowledgments

We thank Andrew Myers for encouragement, help with Jif, and comments on a draft of this paper, which in particular led to improvements in the treatment of retention periods. We also thank Stephen Chong for clarifying some tricky features of Jif, and Nathan Whitehead and the anonymous reviewers for many helpful comments.

This work was partly supported by the National Science Foundation under Grants CCR-0204162 and CCR-0208800.

References

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Implementing P3P using database technology. In *Proceedings of the 19th International Conference on Data Engineering*, pages 595–606, March 2003.

[2] Annie I. Antón. The Privacy Place 2002 privacy values survey. `http://william.stufflebeam.cc/privacySurvey/results/resultsPage.php`, April 2003.

[3] Annie I. Antón. The Privacy Place. `http://www.theprivacyplace.org`, 2004.

[4] Annie I. Antón, Julie B. Earp, Davide Bolchini, Qingfeng He, Carlos Jensen, and William Stufflebeam. The lack of clarity in financial privacy policies and the need for standardization. Technical Report TR-2003-14, North Carolina State University, 2003.

[5] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. Enterprise Privacy Authorization Language (EPAL 1.1). `http://www.zurich.ibm.com/security/enterprise-privacy/epal/Specification/`, 2003.

[6] Paul Ashley, Calvin Powers, and Matthias Schunter. From privacy promises to privacy management: A new approach for enforcing privacy throughout an enterprise. In *Proceedings of the 2002 Workshop on New Security Paradigms*, pages 43–50, 2002.

[7] Michael Backes, Birgit Pfitzmann, and Matthias Schunter. A toolkit for managing enterprise privacy policies. In *8th European Symposium on Research in Computer Security (ESORICS 2003)*, number 2808 in LNCS, pages 162–180. Springer-Verlag, 2003.

[8] Kathy Bohrer, Satoshi Hada, Jeff Miller, Calvin Powers, and Hai fun Wu. Declarative Privacy Monitoring for Tivoli privacy manager. `http://alphaworks.ibm.com/tech/dpm`, October 2003.

[9] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, Mass., 1982.

[10] Lucas C.J. Dreyer and Martin S. Olivier. An information-flow model for privacy (InfoPriv). In Sushil Jajodia, editor, *Database Security XII: Status and Prospects*, pages 77–90. Kluwer, 1999.

[11] Qingfeng He. Privacy enforcement with an extended role-based access model. Technical Report TR-2003-09, North Carolina State University, February 2003.

[12] Qingfeng He and Annie I. Antón. A framework for modeling privacy requirements in role engineering. In *Proceedings of the 9th International Workshop on Requirements Engineering: Foundations for Software Quality*, pages 137–146. Essener Informatik Beiträge, 2003.

[13] Nevin Heintze and John G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 365–377, 1998.

[14] Richard Kevin Hill and Phil Fritz. Reference Monitor for Tivoli privacy manager. `http://alphaworks.ibm.com/tech/refmon`, July 2003.

[15] F.A. Lategan and M.S. Olivier. Privguard: A model to protect private information based on its usage. *South African Computer Journal*, 29:58–68, 2002.

[16] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[17] Andrew C. Myers. *Mostly-Static Decentralized Information Flow.* PhD thesis, Massachussets Institute of Technology, 1999.

[18] Andrew C. Myers and Barbara Liskov. Protecting privacy using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.

[19] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997.

[20] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003.

[21] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[22] Jan C.A. van der Lubbe. PAW: Privacy in an Ambient World. `http://www.cs.kun.nl/paw`, 2004.

[23] World Wide Web Consortium (W3C). The Platform for Privacy Preferences Specification. `http://www.w3.org/TR/P3P`, April 2002.