Information flow control for static enforcement of user-defined privacy policies

Sören Preibusch
Computer Laboratory – University of Cambridge
Email: sdp36@cl.cam.ac.uk

***Abstract*—Information flow control (IFC) allows software programmers and auditors to detect and prevent the sharing of information between different parts of a program which, as a matter of policy, should be kept logically separate. However, the lack of widespread use of IFC suggests technology and usability barriers to adoption.**

The programming language JIF provides IFC on top of Java. To assess pragmatic issues and systematic limitations of using JIF for commercial privacy-preserving Web applications, we deliver the first Web-based case-study with customer-negotiated restrictions on data recipients and usage.

On a practical level, from our experience of programming in JIF, we assess its suitability for preventing accidental misuse of personal information and deduce recommendations for future implementations. On a theoretical level, we explore the compatibility between static analysis and privacy policies configured at runtime.

I. Privacy assurances with information flow control

Leakage of personal information at high-profile Web sites has heightened Web users' privacy concerns. As privacy is becoming a competitive advantage, enterprises need to secure users' personal information against malicious privacy breaches, but also against unintended data flows due to carelessness. As manual code inspection has proven laborious and error-prone, mechanised procedures are needed. Information flow control (IFC) is a technical approach to prevent the unintended leakage and manipulation of sensitive information, and could provide strong guarantees that deployed applications respect privacy and security policies [1]. However, the lack of real-world IFC deployments to enforce privacy constraints suggests technical and usability barriers towards adoption.

This paper reports on the first case-study of using IFC to enforce Web users' self-defined privacy constraints. The benefits and limits of JIF, an IFC programming language built on top of Java, are investigated by implementing privacy negotiations vis-à-vis an online merchant.

Our threat model assumes consumers who voluntarily provide their personal information to a Web site, and communicate with it restrictions on usage and recipients. Data recipients may maliciously try to re-purpose information they receive, or unintentionally process data beyond the constraints as programmers are fallible.

Our contribution is twofold. At a theoretical level, we explore systematic limitations of user-specified privacy policies and their encoding as IFC concepts at runtime, for enforcement with JIF's static analysis of information flow. At a practical level, we study pragmatic issues in producing privacy-preserving Web applications with JIF, comment on its suitability for preventing accidental misuse of personal information, and provide recommendations and software engineering guidance.

II. Brief background on IFC and JIF

A. Basic concepts in information flow control

In an imperative programming language, such as Java and JIF, which builds upon it, variables' values under information flow control are monitored to influence each other only in compliance with a policy, which may be attached to the variables by means of labels. Explicit flows (through assignments and various I/O operations) as well as implicit flows (through control flow and conditional program execution, including timing and exceptional program termination) are checked at compile time or runtime with static or dynamic analysis respectively. To preserve confidentiality, information is not allowed to flow towards a less restrictive variable, as established by an ordering over labels. Static analysis is preferable to enforce IFC constraints, but some restrictions may only be learned as the program is executed.

B. JIF programming language: "Java + information flow"

JIF [2] extends the Java programming language with support for information flow control. Labels are attached to variables when they are declared. The compiler performs static analysis to enforce IFC constraints: explicitly or implicitly, information is not allowed to flow but in accordance with a partial order over labels. Because the permissibility of information flow may depend on inputs not yet available at compile time, additional runtime checks are woven into intermediate, augmented Java code which is compiled by the standard Java compiler against JIF runtime classes.

The Java language syntax is augmented to allow literal labels and principals in the code, both of which can be treated as value types in JIF and which carry the IFC constraints. Written in curly brackets, JIF labels are composed of confidentiality policies (`owner->reader`), and integrity policies (`owner<-writer`), both establishing relations over principals. Multiple policies can be combined with join or meet operators to represent constraints that must be fulfilled simultaneously or alternatively. With certain limitations, principals

and labels can also be created and manipulated as objects, compared using a delegation respectively 'less restrictive than' (<=) ordering at runtime, and applied to other variables.

```
// create labels; dereference* and apply to variable
final label lEmail = new label {pCustomer->pp};
String{*lEmail} sCustomerEmail = "sdp36@cl";
final label lDatabase = new label {pp->};
String{*lDatabase} sDatabase = "";

// compare labels at runtime with <=
if(lEmail <= lDatabase)
  sDatabase += sCustomerEmail + ", ";
```

For Java operators, the impact on label propagation is built into the JIF compiler, but the programmer has to state the information flow introduced by method evaluation. JIF method signatures carry labels for the return value, arguments, exceptions thrown by the method and for bounds on side effects. Method signatures with JIF labels can also be supplied for native Java methods to expose their impact on information flow to the compiler. This way, a JIF programmer can make use of existing Java libraries.

III. System architecture and deployment

We implement an electronic retailing scenario to investigate the usefulness of JIF as a representative of IFC programming languages, for enforcing privacy policies on a production system. This work presents the first case-study on a statically analysed IFC system where the involved principals and their privacy requirements are not known beforehand. Existing language features are put into practice, rather than extended.

A Linux Apache Web server with the Java runtime but not the compiler, serves HTTP requests for Web resources such as stylesheets, images or JavaScript files directly. They act as auxiliary technologies and become part of the trusted computing base. The pre-existing JIF runtime classes are made available to the Java runtime but no further JIF binaries are required (in particular, not the JIF compiler). A shell script wraps application-specific JIF programs which are then invoked via CGI.

Users' privacy constraints are enquired through the browser, using a Web form in our case. A textual representation of their choices is attached to the form data upon submission. Once arrived at the Web server, this representation is parsed by dynamically creating and applying one JIF label per data item received. The enforcement of privacy constraints through IFC can only happen thereafter.

Given the inability to attach meta-data to a form submission in HTML, we have chosen to embed users' choices regarding recipients and purposes of their data in the `name` attribute of input elements, like this: `<input name="email&gt;ShippingDept[OrderNotif]"/>`, with the following advantages: (1) the `name` attribute is the key to the data value and constituent part of the submission; (2) the value of the input element remains intact or could be encrypted whilst leaving the intended recipient in plaintext; (3) the choice of input element names lies with the programmer who can designate appropriate delimiters for the meta-data; (4) restrictions in the HTML specifications on permissible element names are compatible with embedding string representations of privacy choices; (5) because Web browsers auto-complete/pre-fill input elements with values previously supplied to input elements of the same name, auto-complete will only work across fields with the same privacy requirements attached; (6) and similarly, an augmented name breaks access to the value for Web server applications unaware of the privacy constraints.

The JIF libraries were augmented to parse the string representations of constraints on recipients and purposes, received as part of a Web form submission, into a dynamic label with corresponding confidentiality and integrity policies respectively. The orthogonal nature of these privacy restrictions is, thus, paralleled in the IFC domain. For versatile data items, such as email, the resulting restrictions are disjoined with the meet operator, for instance `{pShip->pShip meet pNews->pNews; pShip<-usageNotif meet pNews<-usageAny}`.

IV. Enforcing privacy labels

The Web server principal associated with standard input and output combines Web form values and their corresponding, parsed JIF labels into a wrapper object. When the labelled personal information is later used in the program, a runtime check has to be performed that asserts to the compiler that the label representing restrictions on customer data is less restrictive the label of variables acting as data sinks. Otherwise, compilation fails.

```
// wrapper object: customer data and its label
final label lEmail = oEmail.getLabel();
String{*lEmail} sEmail = oEmail.getVal();

// required minimum permissions on data
final principal usageNotif = new Usage("OrderNotif");
final NamedDataRecipient pShip = oPrincipalFactory
  .createRecipient("ShippingDept");
final label lRequShipNotif =
  new label {pShip->pShip; pShip<-usageNotif};
String{*lShipNotif} sQueueShipNotif = "";

if(lEmail <= lRequShipNotif)
  sQueueShipNotif += sEmail; // violates static analysis
  // if runtime check on preceding line is omitted
```

V. Experience of programming in JIF

By example, this case-study has demonstrated the feasibility of using IFC to enforce user-supplied privacy constraints in commercial systems. However, the considerable effort required for a functionally simple application casts doubt on the practical suitability of JIF.

Concerning the *system architecture*, the observed growth of the trusted computing base beyond the JIF compiler and runtime, to include various Web resources, is unfortunate. Attempts have been made to push the guarantees that JIF

provides further towards the client [3], but some code portions always need to be trusted.

We encountered *systematic limitations* regarding JIF. For our application domain, the single most important issue is the incompatibility between genericity in parsing and exposing privacy constraints on form data and their specificity. An array `String{*li}[] sFormData` would apply the same label `li` across all data items; a function `String{*lr} getFormData(String sKey)` requires a static label `lr`. Data items, thus, have to be processed and accessed one by one. Further, the interoperability of dynamic and literal labels is insufficient, requiring duplication of label expressions and preventing runtime reasoning over a method's argument labels. One cannot inspect labels applied to arbitrary variables.

One of JIF's strongest advantages, the ability to call methods from existing libraries by providing IFC signatures for them, comes at the price of introducing potential security holes by erroneous signatures. Care is required to deduce the appropriate signature from the relating library documentation. The method duo `appendTail`/`appendReplacement` from the `java.util.regex.Matcher` class and I/O methods in general are particularly challenging examples. Unfortunately, the few signatures that come bundled with the JIF installation are incomplete and occasionally do not match with the actual Java class structures. Provided the source code of imported Java libraries exists, the compiler may infer suitable method labels; in practice, however, those were found to be not tight enough, requiring manual intervention.

We also noticed an incompatibility between JIF and some software engineering principles. Refactoring repeated functionality into methods is burdensome as the compiler may frown upon information flow introduced by the revised control flow. Similarly, changes to output channels (e.g., console vs. file) may result in vast changes to the label logic. JIF makes it impossible to adhere to a strict factory pattern with private constructors and the Java access modifiers (public/private) are not considered as an implicit label structure.

The author's *personal experience* of programming in JIF includes periods of frustration and a lack of early successes even for simplistic programs, which could have provided positive feedback in the learning phase. Admittedly, some of the encountered difficulties may not apply for a more advanced JIF programmer. Indeed, the author was able to reach a "plateau of productivity" where coding in JIF and Java went equally smoothly. This notably happened after all variables and method declarations were suitably equipped with JIF labels and implicit information flows were mostly eliminated by reordering conditional statements.

The first major issue is the lack of helpful documentation. The theory-heavy JIF reference manual provides little help for getting started and the sample program shipped with the installation package is far too complicated for a JIF novice. Contrarily to popular programming languages, this lack is not compensated by community content, except the JIF mailing list. The JIF runtime is practically undocumented; the author ultimately guessed its intended use from the source code and explored language features through the compiler test cases. The under-documentation of the runtime is particularly troublesome, as JIF functionality intended to replace potentially insecure Java functionality is thus not used. Also regarding documentation, reading the JIF compiler error messages requires experience. Misleading output, compiler crashes, or errors reported in the intermediate Java code were encountered.

The second major issue is the difficulty in debugging JIF programs, which has also been reported in other case-studies (Section VI). Eventually, a Java class was developed by the author to provide a controlled debug statement that ignores IFC constraints to print arbitrary content to the console – in exploiting the aforementioned security hole of inaccurate method signatures.

It also seems that the 'syntactic sugar', which is expanded during compilation, causes further trouble. Occasionally, debugging requires one to inspect the intermediate compilation. The ban on some Java constructs by the JIF compiler such as inner classes was less of an issue, but generics were missed, for their ability to collect strongly typed data, including principals or labels.

Recommendations and advice. With poor usability and education as the main barriers towards wider adoption of IFC-equipped programming, best practices could help developers starting anew with JIF. We add to the advice given in [4], whilst noting that it seems superior to program directly in JIF instead of enriching an initial Java program with IFC. In the Web companion, available online at http://privacy-calculus.net/, code snippets are provided for accessing standard input and standard output for read/write access to the console, for declassifying data, and for integrating JIF with the Web server. We further recommend making intensive use of the ability to provide own principal implementations for custom privacy needs.

A wish-list to improve to usefulness of JIF for production environments would include: (1) a starters' guide, best practices, and ways out of common compiler errors; (2) access to inferred JIF labels, including program counter labels representing information carried through the program flow. Compiler messages relating to label errors for which the violating constraints are not represented in the program but only in the parse tree of the compiler are very difficult to debug (a common issue with source-expanding pre-compilation); (3) easy access to JIF's internal library classes and constants, at compile time and runtime, such as top and bottom principals; (4) the support for true reader principals, who are allowed to read information but not to pass it on any further. With respect to the overall infrastructure, in which JIF applications are embedded, the ability for Web users to submit meta-data along with their form submissions, would be a valuable addition to the HTML standards.

VI. Related IFC implementations

JIF's strengths have attracted the highest number of software deployments of current IFC-enhanced programming languages. The four deployments to date are discussed:

The JPmail email client [5] relies heavily on certificates to identify extended, cryptographic principals. Textual representations of policies are parsed and applied as labels at compile-time (sic!), as the claimed dynamics in creating new confidentiality requirements are indeed implemented by recompiling the entire system from pre-generated code every time an email is sent. In response to the difficulties experienced with debugging JIF programs, the Eclipse IDE was augmented to support JIF, which was perceived as "not yet ready for industrial development" as of 2006.

A case-study in implementing cryptographic protocols made two JIF programs communicate via console piping, at a total overhead of 400% compared to the original Java program with a single process [4]. Again, only literal principals and labels were used. Although JIF was found useful to detect and to prevent insecure information flows, its all-or-nothing approach to declassification was found too naïve to fit the multi-faceted declassification needs in practice. Encryption was again considered in conjunction with declassification. The deduced patterns to facilitate secure program development and the uncovered insecurities in JIF were also confirmed in our work.

At the same time, an endeavour to implement secure handling of medical information and database security concluded an "impracticality of programming in JIF" [6]. These "explorations [of the language] were stopped short because of the overly burdensome complexity of programming in Jif" [6]. The inadequacy of existing declassification mechanisms and difficulties in debugging were re-iterated.

Finally, the JIF team developed SIF, a Web application framework on top of JIF [3]. SIF renders an HTML page from Java objects on the server. Their labels are woven into the resulting Web page and read back as further requests are received. Together with the control flow resulting from hyperlink navigation, they become a lower bound for the labels of request parameters and received form data. The authors do not share their experiences from programming and deploying SIF.

VII. Conclusion and critical outlook

With dynamic labels, information flow control provides the technical means to enforce runtime-negotiated privacy constraints. The JIF programming language is a powerful tool to engineer IFC-supported privacy compliance – and as such requires expertise. In its current state, novice JIF developers become easily discouraged and frustration hinders adoption. This electronic commerce case-study confirmed a spectrum of usability traps as a major hindrance towards adoption. It also adds to the body of best practices to help novice programmers getting a better start with JIF. However, we foresee further systematic limitations.

Providing JIF signatures for external methods at compile-time may not be compatible with newly discovered and invoked Web services as the program runs. Also, whilst some information flow channels are outside JIF's threat model (e.g., timing attacks), JIF's static analysis is inherently limited to code that has been written before it gets executed. Implicit calls to Java methods such as `toString` or `finalize` are under the radar of the JIF compiler.

JIF can help a benevolent company to substantiate its privacy claims but is not going to stop a malicious company from misusing personal information. Ongoing work thus focuses on further commercial data processing with JIF, such as databases, and to improve auxiliary Web technologies on the consumer side, to facilitate the configuration of privacy choices and their communication to the data controller.

Acknowledgement

Alastair Beresford and Florian Kammüller have provided helpful comments on earlier versions of this work. This project is part of research into a "Privacy Calculus", jointly supported by the British Council and the Deutscher Akademischer Austausch Dienst (ARC Project 1351).

References

[1] S. Preibusch, "Experiments and formal methods for privacy research," in *Privacy and Usability Methods Pow-wow (PUMP)*, August 2010.

[2] A. C. Myers. (1999–2010) Jif: Java + information flow. [Online]. Available: http://www.cs.cornell.edu/jif/

[3] S. Chong, K. Vikram, and A. C. Myers, "SIF: Enforcing Confidentiality and Integrity in Web Applications," in *Proceedings of USENIX Security Symposium 2007*, August 2007, pp. 1–16.

[4] A. Askarov and A. Sabelfeld, "Security-typed languages for implementation of cryptographic protocols: A case study," in *Computer Security - ESORICS 2005*, ser. Lecture Notes in Computer Science, S. d. C. di Vimercati, P. Syverson, and D. Gollmann, Eds. Springer Berlin / Heidelberg, 2005, vol. 3679, pp. 197–221.

[5] B. Hicks, K. Ahmadizadeh, and P. McDaniel, "From languages to systems: Understanding practical application development in security-typed languages," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, Dec. 2006, pp. 153–164.

[6] B. Hicks, P. McDaniel, and A. Hurson, "Information flow control in database security: A case study for secure programming with Jif," Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, Tech. Rep. NAS-TR-0011-2005, April 2005.

Web companion

For code snippets and live demo, please see: `http://privacy-calculus.net/`