Trusted Declassification: Policy Infrastructure for a Security-Typed Language

Boniface Hicks
St. Vincent College
fatherboniface@acm.org
and
Dave King, Patrick McDaniel
Pennsylvania State University
{dhking,mcdaniel}@cse.psu.edu
and
Michael Hicks
University of Maryland, College Park
mwh@cs.umd.edu

Security-typed languages are a powerful tools for developing verifiably secure software applications. Programs written in these languages enforce a strong, global policy of *noninterference* which ensures that high-security data will not be observable on low-security channels. Because noninterference is typically too strong a property, most programs use some form of *declassification* to selectively leak high security information, e.g. when performing a password check or data encryption. Unfortunately, such a declassification is often expressed as an operation within a given program, rather than as part of a global policy, making reasoning about the security implications of a policy difficult.

In this paper, we propose a simple idea we call *trusted declassification* in which special *declassifier* functions are specified as part of the global policy. In particular, individual principals declaratively specify which declassifiers they trust so all information flows implied by the policy can be reasoned about in absence of a particular program. We formalize our approach for a Java-like language and prove a modified form of noninterference which we call *noninterference modulo trusted methods*. We have implemented our approach as an extension to Jif, a security-typed variant of Java, and provide our experience using it to build a secure email client, JPmail.

Categories and Subject Descriptors: CR-number [**subcategory**]: third-level—*fourth level*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints, Data types and structures, Frameworks*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Specification Techniques, Invariants, Mechanical verification*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security, Languages, Design, Theory

Additional Key Words and Phrases: Security-typed languages, declassification, Jif, security policy, information-flow control, *noninterference modulo trusted methods*, trusted declassification, FJifP, JPmail

1. INTRODUCTION

Even a brief glance at the cases prosecuted by the United States Federal Trade Commission reveals the damage that is continually caused by electronic information leakage [Federal Trade Commission 2002; Bangeman 2006]. In protecting sensitive information, including everything from credit card information to military secrets to personal, medical information, there is a pressing need for software applications with strong confidentiality guarantees.

In security-typed languages (STLs), each data item is labeled with its security policy. For example, Alice's password can be labeled to indicate that only Alice may read it:

```
String_Alice alicePwd;
```

Principals may delegate to other principals, so this label more precisely states that Alice and those principals who *act for* Alice may read `alicePwd`. The legal acts-for relationships are typically defined in a global policy kept separate from the program, as in Figure 3. Given this global policy and a particular program, standard type checking enforces the property of *noninterference*, which informally means that throughout the entire execution of the program, only those principals to which Alice (transitively) delegates may learn the contents of her data, whether directly or indirectly. This is quite convenient for the security analyst: to understand the security implications of a particular datum, the analyst needs only to examine the label on the datum and the global acts-for relationships; she does not need to examine the entire program.

Unfortunately, noninterference is too strong a property for real programs. Consider a password check in which a guess is compared with Alice's password:

```
boolean_??? check(String_public guess, String_Alice pwd) {
   return guess isEqualTo alicePwd;
}
```

What should be the label of the boolean return value? The problem is that this function reveals a small amount of information about Alice's password, which is whether or not it is equal to the guessed string. This program would not satisfy noninterference if ??? were public. On the other hand, if ??? were Alice the function would satisfy the type checker, but would be useless as a password checker, because it could not inform the public user whether the password were correct or not.

To remedy this problem, practical STLs support some form of *declassification*, in which high-security information is permitted to flow to a low-security observer. For example, we could rewrite the above function to support declassification selectively, based on a programmer annotation, as follows:

```
boolean_public check(String_public guess, String_Alice pwd) {
   return declassify(guess isEqualTo alicePwd, public);
}
```

Another useful example is when we want to encrypt some data to send it over a public channel:

```
String_public encrypt(String_Alice secret, String_Alice key) {
   return declassify(aesEncrypt(secret,key), public);
}
```

While the declassify operation is efficacious, the problem with such annotation-based declassification is that we have lost localized reasoning about data security. No longer can one simply examine a data label and the global acts-for relations; now one must also find and reason about each occurrence of declassification in the program; i.e., the global meaning of the policy Alice is lost. We can no longer reason about a global security policy (i.e., the acts-for relations) in absence of a program that uses it.

To remedy this problem, we propose the following simple idea. Rather than permit declassification on the granularity of program statements, declassification may only occur within special functions called *declassifiers*. The `check` and `encrypt` functions above are declassifiers. Then, individual principals indicate whether or not they trust a given declassifier as part of the global policy. For example, Alice may allow her data to be encrypted via the `encrypt` declassifier, or may wish to release her personal, medical records for scientific investigation, but only so long as the personal information is stripped out of them first by an `anonymizeMR` declassifier. On the other hand, even the small amount of information released by `check` and `encrypt` might be too much for some sensitive data. Likewise, different anonymization functions may be suitable for different users.

This paper presents a global security policy system, JPolicy, for managing principals in a security-typed language. Our system extends existing work by allowing each principal to indicate which declassifiers it trusts. We call our approach *trusted declassification*. We add functionality for a principal, `p`, to allow a function, `f`, to declassify any of its information to a new label, `lbl` (expressed in our policy language as `p` **allows** `f(lbl)`). With one of our policies in hand, the label on Alice's password regains a global meaning without having to inspect the code of the whole program. For example, if, according to the policy, Alice trusts no declassifiers, then we can be certain that `alicePwd` is only visible to principals who act for her. If, according to the policy, Alice trusts only `encrypt` and `check`, we can check the code and types for these two declassifiers, but not the entire program, to find that negligible information is leaked via the output from each encryption or password check. We have formalized our approach in a Java-like language called FJifP, and proven a noninterference property, called *noninterference modulo trusted methods*, and implemented it as an extension to Jif [Myers et al. 2001], a full scale implementation of a security-typed language based on Java.

There has recently been a proliferation of work seeking to formalize suitable logics for declassification in security-typed languages [Mantel and Sands 2004; Chong and Myers 2004; Broberg and Sands 2006; Matos and Boudol 2005; Myers et al. 2006; Li and Zdancewic 2005a] as detailed in a recent survey [Sabelfeld and Sands 2008]. The value of our approach is borne out of practical experience. In particular, we and others [Askarov and Sabelfeld 2005; Chong et al. 2007] have been trying to

build applications in Jif. Jif supports *selective declassification* [Myers 1999], similar in style to the examples we presented above. Based on existing experience, many uses of declassification—such as for encryption, anonymization, authentication, and filtering—fit nicely into the framework we have proposed. Indeed, when we used our framework to build an SMTP/POP3-compliant e-mail client, JPmail, we found that it made the process of reasoning about declassification and information flows far easier. This work takes a step toward making STLs more practical.

A key aspect of making STLs more practical for systems development lies in the application of systems-construction principles to application development tools for STLs [Hicks 2007]. This work makes security-typed languages more practical by applying a well-tested principle, first proposed for systems construction by Hansen [Hansen 1970]. Our policy infrastructure is the first to separate out an information flow policy with declassifiers into a policy file that may be analyzed and modified apart from an STL application. By separating the information-flow policy from a program in which it is used, the policy can be more easily analyzed for its compliance with systems policies [Hicks et al. 2007a] and also re-configured to work on different systems [Hicks et al. 2007b].

The structure of this paper is as follows: in Section 2 we give an example of a program and policy which we will use throughout the paper to describe our approach. In Section 3, we describe a basic object-oriented, security-typed language, FJifP with declassification and an external policy. We also give the security theorems we have proven about FJifP, namely *noninterference modulo trusted methods*. In Section 4, we describe an external, global policy definition for our system and an implementation of our system in the security-typed language, Jif. Section 5 describes our experience using this framework for building and testing a realistic application, JPmail. Following that, we consider related work in Section 6 and conclude in Section 7.

2. EXAMPLE

Consider the code in Figure 1. Medical records are parameterized by a principal (indicated with <>'s) and a medical record could be instantiated for Alice by writing the following (presuming an implicit constructor which takes arguments of the appropriate security levels to assign each of the member variables).

```
MedicalRecord<Alice> rec = new MedicalRecord<Alice>(...)
```

A medical record can release its history with the method `getHistory`, but the label on the return value, *p*:, ensures that it will remain protected after it is released. A medical record can also write its history to a public stream (a socket or a file, e.g.) via the `saveHistory` method, but because the stream is public, the history must be passed through a declassifier, in this case it is encrypted with AES. Finally, using the method `updateName`, the name on the medical record can be updated by someone other than `p`, but only if that principal knows the password. Here again, declassification is needed, because the result of comparing a public value, `guess`, and a secret value, `password`, is stored in a public boolean, `valid`. Thus, the declassifier `check` is used to do the comparison and declassify the result. Principals must authorize these declassifications explicitly in the global policy.

```
class MedicalRecord<p> {
  String_public name;
  String_p: history;
  Key_p: aesKey;
  String_p: password;

  String_p: getHistory() { return history; }

  void saveHistory(OutputStream_public out) { out.write(AES<p>.encrypt(history,aesKey)); }

  void updateName(String_public guess, String_public newName) {
     boolean_public valid = Passwd<p>.check(guess,password);
     if (valid) name = newName; }
}
```

Fig. 1. A simple example

```
Alice −> DrBob
Alice allows Passwd.check(public)
Alice allows AES.encrypt(public)
DrBob allows AES.encrypt(public)
DrBob −> DrJohn
Chuck −> DrBob
```

Fig. 2. A simple policy

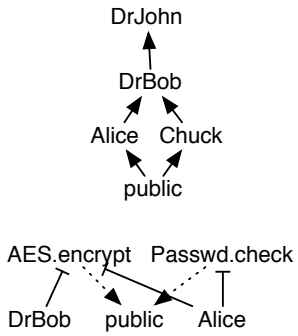

Fig. 3. Example acts-for hierarchy and declassifier context.

A simple global policy is shown in Figure 2. Global policies express both delegations, using −>, and trusted declassifiers, using **allows**. Given this policy, we can determine all the possible ways in which Alice's data can flow. Anything Alice can read can also flow to Dr. Bob, because Alice explicitly trusts him (indicated by `Alice −> DrBob`). It can also flow transitively to his partner, Dr. John. More interestingly, this policy contains all of the declassifiers which Alice will allow to operate on her data. Thus, we see that Alice's data can flow to a public output, but only if it is first encrypted with AES. This is asserted by the `Alice` **allows** `AES.encrypt(`**public**`)` policy statement. Alternatively, Alice's data might be leaked (a little bit at a time) via a password check. It is also possible for a principal to release information to a different security level through a declassifier, for example, `DrBob` **allows** `Doctor.patientReport(Alice)`. In this case, the declassifier only

releases the information to the patient, not to the public. The declassifier sanitizes the doctor's notes before the release.

In FJifP, security is enforced statically by the type-checker, by disallowing programs that violate their policy. Consider the two methods, `updateName` and `saveHistory`. These methods utilize declassifiers, `Passwd.check` and `AES.encrypt`, respectively. In order to instantiate a `MedicalRecord` with a principal `p`, we require that `p` allows the use of these declassifiers. Thus, given the policy in Figure 2, the above instantiation of `rec` for Alice will succeed, because Alice allows both declassifiers. On the contrary, attempting to instantiate a medical record for Chuck would cause a type error.

In this example, we can see how policy can be lifted out of a program and stored in an external file. In this way, when examining any fragment of code, we can understand the security guarantees of policy labels by consulting a centralized policy file. It is worth noting that a precise characterization of *how much* information can be leaked would also require inspecting the code of the declassifiers. For example, consulting the code for encryption and the code for password checks readily leads to the conclusion that very little information is leaked through these methods. Since the number of declassifiers for an application is generally small, it is not hard to inspect them by hand. Furthermore, a standard collection of declassifiers can be built up over time with careful analyses of the information leakage allowed by each.

The challenge of automatically quantifying information leakage is being studied elsewhere [Lowe 2002; Chen and Malacaria 2007; Clark et al. 2007]. We expect that as these results mature, they will integrate cleanly with our system.

3. SEMANTICS AND PROPERTIES OF FJIFP

3.1 Introduction to FJifP

We first describe FJifP (short for Featherweight Jif with Policy), a security-typed, object-oriented language. FJifP is an extension of Featherweight Java [Igarashi et al. 1999] that includes the essential security features of Jif as well as the option for certain methods to be used as declassifiers. We then give typing and evaluation rules for that system, show their soundness, and prove a theorem about the language's security, *noninterference modulo trusted methods*.

Featherweight Java (FJ) is a minimal subset of the Java programming language that models essential features of an object-oriented language such as field access, dynamic dispatch, inheritance, casting, and mutually recursive classes. It does not include many features of the full language, including mutable state, concurrency, and introspection. Conditionals can be implemented through inheritance and loops can be implemented through recursive method calls.

In giving the definition of FJifP, we seek to add security types and runtime principals to FJ in order to provide a basic framework for the Jif language. We omit some of the more complex features of Jif such as authority and unrestricted declassification. To additionally simplify the presentation of our system, we omit two mechanisms of FJ: non-default constructors and unrestricted type casts. These language features do not interfere with the security properties.

Figure 4 shows the Medical Record Example from Figure 1, modified to be a program in FJifP, extended with primitives for booleans and conditional expressions.

```
class MedicalRecord<α>{ξ ◁ public} ◁ Object {
  String_public name;
  String_α: history;
  Key_α: aesKey;
  String_α: password;

  String_α: getHistory() { return history; }

  OutputStream_public saveHistory(OutputStream_public out) {
     return out.write( new AES<α>_α:().encrypt( this.history, this.aesKey ) );}

  MedicalRecord<α>_public updateName( String_public guess, String_public newName ) {
     if ( declassify(new Passwd<α>_α:().check( guess, password ), public) )
         return new MedicalRecord<α>_public( this.newName, this.history,
                                             this.aesKey, this.password );
     else return this; }
}
```

Fig. 4. Figure 1, rewritten in FJifP

For the most part, the code in Figure 4 remains the same as the pseudo-code. We presume the standard encodings for **if** and the existence of `OutputStream`, `String`, `Key`, etc. The keyword Public is a special principal having the property that Public $\preceq$ p for all principals p and the label *public* being the policy {Public :}. There are also a few things to note involving the lack of state, static methods. First, when the original code called for modification of a medical record through an assignment statement, the new code instead returns a new medical record. Static methods (such as the call to `AES.encrypt`) have been replaced by creating new instances of the class and then calling that member function on them.

Because there is an illegal, implicit flow between the public string `guess` and the $\{\alpha :\}$-level password in `updateName`, this class cannot be type-checked without some notion of declassification. In this example, to correctly type the `updateName` method, we add a declassify statement around the `check` method. This is legal as long as the policy allows `updateName` to declassify data from Alice to Public.

There is one other detail of note in `updateName`. Our type system does not include a special label for the security level of `this`, and so to label-check the body of a method, we specify the maximum label that the class can be instantiated with. For example, the header of `MedicalRecord` is tagged with $\xi \triangleleft \mathit{public}$, which indicates that the only legal instances of the `MedicalRecord` class of the form $\texttt{MedicalRecord}\langle\alpha\rangle_{\mathit{public}}$ for some principal α.

3.2 Definitions

An FJifP program consists of a series of defined classes $\texttt{C}, \texttt{D}, \ldots$ and terms $\texttt{t}_1, \texttt{t}_2, \ldots$ that are to be evaluated under a series of class definitions. Terms might invoke methods, access fields, create new instances of classes, and perform casts (to name

Class Names	$\mathtt{C}, \mathtt{D}$	
Field Names	$\mathtt{f}, \mathtt{g}$	
Method Names	$\mathtt{m}$	
Variables	$\mathtt{x}, \mathtt{y}$	
Principals	p, q, r	
Policies	$d ::=$	$p_1 : \overline{q}_1; \ldots; p_k : \overline{q}_k$
Labels	$l ::=$	$\{d\}$
Param. Classes	$\mathtt{N} ::=$	$\mathtt{C}\langle\overline{p}\rangle$
Security Types	$\mathtt{S}, \mathtt{T} ::=$	$\mathtt{N}\{l\}$
Class Definitions	$\mathtt{CL} ::=$	$\mathtt{class\ C}\langle\overline{\alpha}\rangle \lhd \mathtt{N}\ \{\ \overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}$
Methods	$\mathtt{M} ::=$	$\mathtt{S\ m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return(t);}\ \}$
Terms	$\mathtt{t} ::=$	$\mathtt{x}$
		$\mid \mathtt{t.f}$
		$\mid \mathtt{t.m}(\overline{\mathtt{t}})$
		$\mid \mathtt{new\ S(t)}$
		$\mid \mathtt{(S)\ t}$
		$\mid \mathtt{actsfor}(p, q)\ \mathtt{in\ t}$
		$\mid \mathtt{declassify}^{\mathtt{m}}(\mathtt{t}, l)$
Values	$\mathtt{u}, \mathtt{v} ::=$	$\mathtt{new\ S}(\overline{\mathtt{v}})$
Actsfor Hierarchy	$(p, q) \in \Delta$	
Declass. Policy	$(\mathtt{m}, p, q) \in \Upsilon$	
Security Contexts	$\Theta = (\Delta, \Upsilon)$	

Fig. 5. FJifP Language Syntax

a few possibilities). Classes contain fields $\mathtt{f}$ and methods $\mathtt{m}$. Instantiated classes are parameterized by principals p and given security labels l for security. The notation $\overline{\mathtt{x}}$ represents a list: $\overline{\mathtt{x}}$ is a list of variables, parameterized $\mathtt{x}_1, \ldots, \mathtt{x}_n$.

The language syntax for FJifP is given in Figure 5. FJifP adds new constructs for $\mathtt{actsfor}(p, q)\ \mathtt{in\ t}$, which checks that p acts for q in $\mathtt{t}$, and $\mathtt{declassify}^{\mathtt{m}}(\mathtt{t}, l)$, declassifies the term $\mathtt{t}$ to security level l under the policies of $\mathtt{m}$. Statements of the form $\mathtt{declassify}^{\mathtt{m}}(\mathtt{t}, l)$ can only occur inside the body of a method $\mathtt{m}$.

Our security labels follow the decentralized label model (DLM) [Myers 1999], which permits multiple policies on values. A label l is made up of policies. Each policy consists of an owning principal p together with reader lists allowed by that principal (implicitly including p). The type system ensures that all of the policies in a label are enforced, requiring a reader to appear in all policies in order to read the data. For example, let l be the label $\{\mathsf{Alice} : \mathsf{Bob}, \mathsf{Charlie}; \mathsf{Charlie} : \mathsf{Bob}\}$. Alice owns the first policy, and is implicitly a reader. Bob, and Charlie are also readers in this policy. The second policy is owned by Charlie and readable by both Bob and Charlie. If a value $\mathtt{v}$ has been instantiated and tagged with l, then either Bob or Charlie can read $\mathtt{v}$; though Alice owns a policy on $\mathtt{v}$, she is not a reader in Charlie's policy. A security context Θ then has two primary judgements: the first tests if the principal q is trusted to act for p, written $\Theta \vdash p \preceq q$. The second tests if a label l_2 is at least as restrictive as l_1 and is written $\Theta \vdash l_1 \sqsubseteq l_2$. The metavariable d represents a list of policies $p : \overline{q}$. These rules are given in Figure 6.

FJifP classes and terms are assigned a type under a global security context $\Theta = (\Delta, \Upsilon)$. The trust relations between principals are given in the acts-for hierarchy

Actsfor Checking

$$\frac{}{\Theta \vdash p \preceq p} \text{ (PLT-REFL)}$$

$$\frac{(p,q) \in \Theta(\Delta)}{\Theta \vdash p \preceq q} \text{ (PLT-ACTSFOR)}$$

$$\frac{\Theta \vdash p \preceq r \quad \Theta \vdash r \preceq q}{\Theta \vdash p \preceq q} \text{ (PLT-TRANS)}$$

Label Comparison

$$\frac{\forall p : \overline{q} \in d_1 \;.\; \exists p' : \overline{q'} \in d_2 \;.\; \Theta \vdash p : \overline{q} \sqsubseteq \overline{p'} : \overline{q'}}{\Theta \vdash \{d_1\} \sqsubseteq \{d_2\}} \text{ (SEC-LAB)}$$

$$\frac{\Theta \vdash p \preceq p' \quad \forall q'_i \in \overline{q'} \;.\; \exists q_j \in \overline{q} \;.\; \Theta \vdash q_j \preceq q'_i}{\Theta \vdash p : \overline{q} \sqsubseteq p' : \overline{q'}} \text{ (SEC-LIST)}$$

Fig. 6. Security Context Judgements

Δ. For example, if Alice trusts Bob to act for her, then $(\mathsf{Alice}, \mathsf{Bob}) \in \Delta$. The declassification policy Υ allows for users to specify trust relationships with higher granularity. If the triple $(\mathtt{m}, p, q) \in \Upsilon$, then the trust relation (p, q) is added to the acts-for hierarchy Δ when type-checking the method `m`. `m` then acts as an information flow from p's data to q. We define the function $\mathsf{policyflows}(\Upsilon, \mathtt{m})$ as follows:

$$\mathsf{policyflows}(\Upsilon, \mathtt{m}) = \{\ (p,q) \mid (\mathtt{m}, p, q) \in \Upsilon\}$$

We overload the $\mathsf{extract}$ function on security contexts in the natural way: $\mathsf{policyflows}(\Theta, \mathtt{m}) = \mathsf{policyflows}(\Upsilon, \mathtt{m})$ if $\Theta \equiv (\Delta, \Upsilon)$, while if $\Theta \equiv (\Delta, \Upsilon)$, the notation $\Theta \cup \Delta'$ represents the security context $(\Delta \cup \Delta', \Upsilon)$.

While $\mathsf{policyflows}$ defines the principal flows that a method allows, the function $\mathsf{flows}(\Theta, \mathtt{m}, l)$ contains the security labels l' that a method may declassify data to:

$$\mathsf{flows}(\Theta, \mathtt{m}, l) = \{l' \mid \Theta \cup \mathsf{policyflows}(\Theta, \mathtt{m}) \vdash l \sqsubseteq l'\}$$

In FJifP, classes can be templated by principals, which introduces a principal variable α that can be used within the class. When we create a new instance of a class, the templated principals are then substituted in for the principal variables of a class. Templated classes, $\mathtt{C}\langle\overline{p}\rangle$, are represented by the meta-variable `N`. Security types, $\mathtt{C}\langle\overline{p}\rangle\{l\}$, are templated classes with labels attached, and they are ranged over by $\mathtt{S}, \mathtt{T}$. The function lab returns the label associated with a security type, while the expression $\mathtt{S} \sqcup l$ represents the security type `S` raised to the security level $\mathsf{lab}(\mathtt{S}) \sqcup l$. The definitions for these are as follows:

$$\mathsf{lab}(C\langle\overline{p}\rangle\{l\}) = l$$
$$C\langle\overline{p}\rangle\{l\} \sqcup l' = C\langle\overline{p}\rangle\{l \sqcup l'\}$$

As in FJ, there is a special class, `Object`, that has no principal variables, no fields, and no methods.

FJifP contains a class table CT that contains each class's definition. A class `C` has the following definition:

$$\mathrm{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle\{\xi \lhd l\} \lhd \mathtt{D}\langle\overline{q}\rangle\ \{\ \overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}$$

Here, C is a class with principal parameters α (the bar indicates a list) that which inherits from the class $\mathtt{D}\langle\overline{q}\rangle$ (some of the q_i might be in $\overline{\alpha}$). The label parameter $\xi \lhd l$ indicates that C can be tagged with any label l_0 that is no more secure than the label l. When type-checking the methods of C, the `this` pointer is assumed to have security level l.

The class C contains the fields and methods declared in its parent, along with the locally defined fields $\overline{\mathtt{S}}\ \overline{\mathtt{f}}$ and methods $\overline{\mathtt{M}}$; these methods may override its parent's methods, but are required to have the same type signatures.

We define a few simple functions for future reference.

—$\mathsf{parent}(\mathtt{C}) = \mathtt{D}\langle\overline{q}\rangle$: the parent of a class.

—$\mathsf{pvars}(\mathtt{C}) = \overline{\alpha}$: the principal variables of a class.

—$\mathsf{localfields}(\mathtt{C}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}$: fields declared locally. Each field has a security type associated with it.

—$\mathsf{localmethods}(\mathtt{C}) = \overline{\mathtt{M}}$: methods declared locally. Each method specifies the security type of its arguments and the security type of the returned value.

Member methods m are declared as follows: $\mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})$. This method m takes arguments $\overline{\mathtt{x}}$ of security type $\overline{\mathtt{S}}$ and returns a value of the security type $\mathtt{S}_0$. The definitions for field lookup, method lookup, and a number of other utility functions are given in 3.2 and closely follow the analogous functions in FJ. The notation $\mathtt{t}[\mathtt{v}/\mathtt{x}]$ represents a simultaneous substitution being performed: in this case the value v is substituted for the free variables x in the term t.

3.3 Typing Rules

We now present the typing rules for terms and classes; these are given in Figure 8. Let Γ be an environment mapping variables to security types. The judgement for term typing $\Theta; \Gamma \vdash \mathtt{t} : \mathtt{S}$ assigns security types S to terms t under security context Θ and environment Γ.

The typing rules for terms are standard adaptations of their corresponding rules in FJ, except for two cases: (TP-NEW) and (TP-DECLASSIFY). When we create a new instance of a class using (TP-NEW), we must check that it has been parameterized with legal principals and assigned a legal security label. The typing rule for declassification is the only rule that allows explicitly lowering the security label on data. A declassify term t can use the flows that the declassification policy Υ allows for the method m.

There are three well-formedness checks, for types, classes, and methods. Respectively, these are $\Theta \vdash \mathtt{S}$ OK, $\Theta \vdash \mathtt{C}\langle\overline{\alpha}\lhd\overline{p}\rangle\{\xi\lhd l\}$ OK, and $\Theta \vdash \mathtt{m}$ OK IN $\mathtt{C}\langle\overline{p}\rangle\{\xi\lhd l\}$. We define these rules in Figure 9. Checking that a type is well-formed requires it to be instantiated with principals and labels compatible with the declared principal and label upper bounds on the class. For a class to be well-formed, all of its methods must be well-formed. For a method to be well-formed, its body must be typeable to the declared security type and not conflict with the type of its superclass as specified by the override function.

3.4 Subtyping

In FJ, a class C is a subtype of another class D if D is C, C inherits from D, or there is a $\mathtt{C}'$ such that C is a subtype of $\mathtt{C}'$ and $\mathtt{C}'$ is a subtype of D. For FJifP, we

Field Lookup

$$\mathsf{fields}(\texttt{Object}\{l\}) = \bullet$$

$$\frac{\begin{array}{c}\mathsf{localfields}(\texttt{C}) = \overline{\texttt{S}}\ \overline{\texttt{f}} \\ \mathsf{parent}(\texttt{C}) = \texttt{D}\langle\overline{q}\rangle \quad \mathsf{pvars}(\texttt{C}) = \overline{\alpha} \\ \mathsf{fields}(\texttt{D}\langle\overline{q}[\overline{p}/\alpha]\rangle) = \overline{\texttt{T}}\ \overline{\texttt{g}}\end{array}}{\mathsf{fields}(\texttt{C}\langle\overline{p}\rangle) = (\overline{\texttt{T}}\ \overline{\texttt{g}}, \overline{\texttt{S}}[\overline{p}/\overline{\alpha}]\ \overline{\texttt{f}})}$$

Method Typing

$$\frac{\begin{array}{c}S\ \texttt{m}(\overline{\texttt{S}}\ \overline{\texttt{x}})\ \{\ \texttt{return(t);}\ \} \in \mathsf{localmethods}(\texttt{C}) \\ \mathsf{pvars}(\texttt{C}) = \overline{\alpha}\end{array}}{\mathsf{mtype}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle) = (\overline{\texttt{S}} \to \texttt{S})[\overline{p}/\overline{\alpha}]}$$

$$\frac{\begin{array}{c}\texttt{m} \text{ not defined in } \mathsf{localmethods}(\texttt{C}) \\ \mathsf{parent}(\texttt{C}) = \texttt{D}\langle\overline{q}\rangle \quad \mathsf{pvars}(\texttt{C}) = \overline{\alpha} \\ \mathsf{mtype}(\texttt{m}, \texttt{D}\langle\overline{q}[\overline{p}/\overline{\alpha}]\rangle) = \overline{\texttt{S}} \to \texttt{S}_0\end{array}}{\mathsf{mtype}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle) = \overline{\texttt{S}} \to \texttt{S}_0}$$

Method Body Lookup

$$\frac{\begin{array}{c}\texttt{S}\ \texttt{m}(\overline{\texttt{S}}\ \overline{\texttt{x}})\ \{\ \texttt{return(t);}\ \} \in \mathsf{localmethods}(\texttt{C}) \\ \mathsf{pvars}(\texttt{C}) = \overline{\alpha}\end{array}}{\mathsf{mbody}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle) = (\overline{\texttt{x}}, \texttt{t}[\overline{p}/\overline{\alpha}])}$$

$$\frac{\begin{array}{c}\texttt{m} \text{ not defined in } \mathsf{localmethods}(\texttt{C}) \\ \mathsf{parent}(\texttt{C}) = \texttt{D}\langle\overline{q}\rangle \quad \mathsf{pvars}(\texttt{C}) = \overline{\alpha} \\ \mathsf{mbody}(\texttt{m}, \texttt{D}\langle\overline{q}[\overline{p}/\overline{\alpha}]\rangle) = (\overline{\texttt{x}}, \texttt{t})\end{array}}{\mathsf{mbody}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle) = (\overline{\texttt{x}}, \texttt{t}[\overline{p}/\overline{\alpha}])}$$

Declared Methods

$$\frac{\mathsf{mbody}(\texttt{m}, \texttt{S}) = (\overline{\texttt{x}}, \texttt{t})}{\texttt{m} \in \mathsf{methods}(\texttt{S})}$$

$$\frac{\mathrm{CT}(\texttt{C}) = \texttt{class C}\langle\overline{p}\rangle\{\xi \lhd l\} \lhd \cdots \{\cdots\}}{\mathsf{container}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle) = \texttt{C}\langle\overline{p}\rangle\{l\}}$$

Method Overriding

$$\frac{\begin{array}{c}\mathsf{mtype}(\texttt{m}, \texttt{D}\langle\overline{q}\rangle) = \overline{\texttt{T}} \to \texttt{T}_0 \text{ implies} \\ \overline{\texttt{S}} = \overline{\texttt{T}} \text{ and } \texttt{S}_0 = \texttt{T}_0\end{array}}{\mathsf{override}(\texttt{m}, \texttt{D}\langle\overline{q}\rangle, \overline{\texttt{S}} \to \texttt{S}_0)}$$

Overloaded Functions for Security Types

$$\mathsf{fields}(\texttt{C}\langle\overline{p}\rangle\{l\}) = \mathsf{fields}(\texttt{C}\langle\overline{p}\rangle)$$

$$\mathsf{mbody}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle\{l\}) = \mathsf{mbody}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle)$$

$$\mathsf{mtype}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle\{l\}) = \mathsf{mtype}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle)$$

$$\mathsf{container}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle\{l\}) = \mathsf{container}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle)$$

$$\frac{\mathsf{override}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle, \overline{\texttt{S}} \to \texttt{S}_0)}{\mathsf{override}(\texttt{m}, \texttt{C}\langle\overline{p}\rangle\{l\}, \overline{\texttt{S}} \to \texttt{S}_0)}$$

Fig. 7. Auxiliary Definitions

need to define exactly what it means for a security type $\texttt{C}\langle\overline{p}\rangle\{l\}$ to be a subtype of $\texttt{D}\langle\overline{q}\rangle\{l\}$. The combination of two observations forms our subtyping rules, given for the subtyping judgement $\Theta \vdash \texttt{S <:T}$ in Figure 10.

If $\mathrm{CT}(\texttt{C}) = \texttt{class C}\langle\alpha\rangle\{\xi \lhd l\} \lhd \texttt{D}\langle\alpha\rangle\ \{\cdots\}$, then $\texttt{C}\langle\mathsf{Alice}\rangle\{l_0\}$ is a subtype of $\texttt{D}\langle\mathsf{Alice}\rangle\{l_0\}$ for all l_0 that are no more secure than l. Following Jif, we do not extend subtyping within class parameters: $\texttt{C}\langle\mathsf{Alice}\rangle\{l_0\}$ is not a subtype of $\texttt{C}\langle\mathsf{Bob}\rangle\{l_0\}$ even if $\Theta \vdash \mathsf{Alice} \preceq \mathsf{Bob}$. As we can always safely raise the security level of data, $\texttt{C}\langle\overline{p}\rangle\{l_1\}$ is a subtype of $\texttt{C}\langle\overline{p}\rangle\{l_2\}$ if l_2 is at least as restrictive as l_1 and both l_1 and l_2 are no more secure than l.

3.5 Evaluation

Evaluation in FJifP is similar to FJ, with the exception of the `actsfor` expression, where a runtime check of the current security policies is done. The evaluation judgement $\Theta \vdash \texttt{t} \overset{\iota}{\mapsto} \texttt{t}'$ is a small-step reduction: under security context Θ, t makes a single step to t′ with declassifications ι, where ι is either blank or a security label l. When we talk of a series of consecutive evaluations, we write $\Theta \vdash \texttt{t} \overset{\Lambda}{\hookrightarrow} \texttt{v}$, representing multiple evaluation steps containing declassifications to the labels in

Typing

$$\frac{\Gamma(\mathtt{x}) = \mathtt{S}}{\Theta;\Gamma \vdash \mathtt{x} : \mathtt{S}} \text{ (TP-VAR)}$$

$$\frac{\Theta;\Gamma \vdash \mathtt{t}_0 : \mathtt{S} \quad \mathtt{S}_i\ \mathtt{f}_i \in \mathsf{fields}(\mathtt{S})}{\Theta;\Gamma \vdash \mathtt{t}_0.\mathtt{f}_i : \mathtt{S}_i \sqcup \mathsf{lab}(\mathtt{S})} \text{ (TP-FIELD)}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash \mathtt{t}_0 : \mathtt{S}_0 \ \ \mathsf{mtype}(\mathtt{m}, \mathtt{S}_0) = \overline{\mathtt{S}} \to \mathtt{S} \\ \Theta;\Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}'} \ \ \Theta \vdash \overline{\mathtt{S}'} <: \overline{\mathtt{S}}\end{array}}{\Theta;\Gamma \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{S} \sqcup \mathsf{lab}(\mathtt{S}_0)} \text{ (TP-INVK)}$$

$$\frac{\begin{array}{c}\mathsf{fields}(\mathtt{C}\langle\overline{p}\rangle\{l\}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \\ \Theta;\Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}'} \ \ \Theta \vdash \overline{\mathtt{S}'} <: \overline{\mathtt{S}} \\ \Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l\}\ \mathrm{OK}\end{array}}{\Theta;\Gamma \vdash \mathtt{new}\ \mathtt{C}\langle\overline{p}\rangle\{l\}(\overline{\mathtt{t}}) : \mathtt{C}\langle\overline{p}\rangle\{l\}} \text{ (TP-NEW)}$$

$$\frac{\Theta;\Gamma \vdash \mathtt{t}_0 : \mathtt{S}_0 \quad \Theta \vdash \mathtt{S}_0 <: \mathtt{S}}{\Theta;\Gamma \vdash (\mathtt{S})\ \mathtt{t}_0 : \mathtt{S}} \text{ (TP-UPCAST)}$$

$$\frac{\Theta;\Gamma \vdash \mathtt{t} : \mathtt{S} \quad \Theta \vdash p \preceq q}{\Theta;\Gamma \vdash \mathtt{actsfor}(p, q)\ \mathtt{in}\ \mathtt{t} : \mathtt{S}} \text{ (TP-ACTSFOR)}$$

$$\frac{\begin{array}{c}\Theta;\Gamma \vdash \mathtt{t} : \mathtt{C}\langle\overline{p}\rangle\{l_0\} \\ l \in \mathsf{flows}(\Theta, \mathtt{m}, l_0)\end{array}}{\Theta;\Gamma \vdash \mathtt{declassify}^{\mathtt{m}}(\mathtt{t}, l) : \mathtt{C}\langle\overline{p}\rangle\{l\}} \text{ (TP-DECLASSIFY)}$$

Fig. 8. Typing Rules

Λ. The evaluation rules for FJifP are given in Figure 11.

3.6 Type System Properties

With the following lemmas, we prove that FJifP obeys all the usual type system properties.

LEMMA 3.1 WEAKENING. *Suppose* $\Theta;\Gamma \vdash \mathtt{t} : \mathtt{S}$, $\Gamma' \supseteq \Gamma$, *and* $\Theta' \supseteq \Theta$. *Then* $\Theta';\Gamma' \vdash \mathtt{t} : \mathtt{S}$.

PROOF. Proof proceeds by induction on the typing derivation.

Suppose $\Theta;\Gamma \vdash \mathtt{x} : \mathtt{S}$; then by inversion we have $\Gamma(\mathtt{x}) = \mathtt{S}$. Since $\Gamma' \supseteq \Gamma$, $G'(\mathtt{x}) = \mathtt{S}$, and so $\Theta;\Gamma' \vdash \mathtt{x} : \mathtt{S}$ as required.

All other cases follow by straightforward induction on the typing assumptions. □

LEMMA 3.2. *Suppose* $\Theta \vdash \mathtt{S} <: \mathtt{T}$ *and let* $\mathsf{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{S}} \to \mathtt{S}_0$. *Then* $\mathsf{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{S}} \to \mathtt{S}_0$.

PROOF. Induction on the derivation of $\Theta \vdash \mathtt{S} <: \mathtt{T}$. □

Type Well-Formedness

$$\frac{\Theta \vdash \mathtt{C}\langle\overline{\alpha} \lhd \overline{q}\rangle\{\xi \lhd l\}\ \text{OK} \quad \Theta \vdash \overline{p} \preceq \overline{q} \quad \Theta \vdash l_0 \mathtt{<:} l[\overline{p}/\overline{\alpha}]}{\Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l_0\}\ \text{OK}}$$

Class Well-Formedness

$$\frac{\text{for all } \mathtt{m} \in \mathsf{methods}(\mathtt{C}),\ \Theta \vdash \mathtt{m}\ \text{OK IN}\ \mathtt{C}\langle\overline{\alpha} \lhd \overline{q}\rangle\{\xi \lhd l\} \quad \mathsf{parent}(\mathtt{C}) = \mathtt{D}\langle\overline{q}\rangle \quad \Theta \vdash \mathtt{D}\langle\overline{q}\rangle\{l\}\ \text{OK}}{\Theta \vdash \mathtt{C}\langle\overline{\alpha} \lhd \overline{q}\rangle\{\xi \lhd l\}\ \text{OK}}$$

Method Well-Formedness

$$\frac{\begin{array}{c}\mathsf{mbody}(\mathtt{m}, \mathtt{C}\langle\overline{p}\rangle\{l\}) = (\overline{\mathtt{x}}, \mathtt{t}_0) \\ \mathsf{mtype}(\mathtt{m}, \mathtt{C}\langle\overline{p}\rangle\{l\}) = \overline{\mathtt{S}} \to \mathtt{S}_0 \\ \Theta; \overline{\mathtt{x}} : \overline{\mathtt{S}}, \mathtt{this} : \mathtt{C}\langle\overline{p}\rangle\{l\} \vdash \mathtt{t}_0 : \mathtt{T}_0 \\ \Theta \vdash \mathtt{T}_0 \mathtt{<:} \mathtt{S}_0 \\ \mathsf{parent}(\mathtt{C}) = \mathtt{D}\langle\overline{q}\rangle \\ \mathsf{override}(\mathtt{m}, \mathtt{D}\langle\overline{q}\rangle\{l\}, \overline{\mathtt{S}} \to \mathtt{S}_0)\end{array}}{\Theta \vdash \mathtt{m}\ \text{OK IN}\ \mathtt{C}\langle\overline{\alpha} \lhd \overline{q}\rangle\{\xi \lhd l\}}$$

Fig. 9. Well-Formedness Rules

Subtyping Rules

$$\frac{}{\Theta \vdash \mathtt{S} \mathtt{<:} \mathtt{S}}\ (\textsc{s-refl})$$

$$\frac{\Theta \vdash \mathtt{S} \mathtt{<:} \mathtt{S}' \quad \Theta \vdash \mathtt{S}' \mathtt{<:} \mathtt{T}}{\Theta \vdash \mathtt{S} \mathtt{<:} \mathtt{T}}\ (\textsc{s-trans})$$

$$\frac{\Theta \vdash l_1 \sqsubseteq l_2 \quad \Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l_1\}, \mathtt{C}\langle\overline{p}\rangle\{l_2\}\ \text{OK}}{\Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l_1\} \mathtt{<:} \mathtt{C}\langle\overline{p}\rangle\{l_2\}}\ (\textsc{s-label})$$

$$\frac{\mathsf{parent}(\mathtt{C}) = \mathtt{D}\langle\overline{q}\rangle \quad \mathsf{pvars}(\mathtt{C}) = \overline{\alpha} \quad \Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l\}, \mathtt{D}\langle\overline{q}[\overline{p}/\overline{\alpha}]\rangle\{l\}\ \text{OK}}{\Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l\} \mathtt{<:} \mathtt{D}\langle\overline{q}[\overline{p}/\overline{\alpha}]\rangle\{l\}}\ (\textsc{s-class})$$

Fig. 10. Subtyping Rules

LEMMA 3.3 VALUE SUBSTITUTION. *Suppose* $\Theta; \Gamma, \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t} : \mathtt{S}$ *and* $\Theta \vdash \mathtt{v} : \mathtt{S}'_0$, *where* $\Theta \vdash \mathtt{S}'_0 \mathtt{<:} \mathtt{S}_0$. *Then* $\Theta; \Gamma \vdash \mathtt{t}[\mathtt{v}/\mathtt{x}] : \mathtt{S}'$ *for some* $\mathtt{S}'$ *such that* $\Theta \vdash \mathtt{S}' \mathtt{<:} \mathtt{S}$.

PROOF. Induction on the derivation of $\Theta; \Gamma, \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t} : \mathtt{S}$. □

LEMMA 3.4. *Suppose* $\Theta \vdash \mathtt{S}_0$ OK, $\mathsf{mtype}(\mathtt{m}, \mathtt{S}_0) = \overline{\mathtt{T}} \to \mathtt{T}$, *and* $\mathsf{mbody}(\mathtt{m}, \mathtt{S}_0) = (\overline{\mathtt{x}}, \mathtt{t})$. *Then for some* $\mathtt{T}_0$ *such that* $\Theta \vdash \mathtt{S}_0 \mathtt{<:} \mathtt{T}_0$, *there exists* $\mathtt{S}$ *with* $\Theta \vdash \mathtt{S} \mathtt{<:} \mathtt{T}$ *and* $\Theta \cup \mathsf{policyflows}(\Theta, \mathtt{m}); \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{T}_0 \vdash \mathtt{t} : \mathtt{S}$.

PROOF. Induction on the judgement $\mathsf{mbody}(\mathtt{m}, \mathtt{S}_0)$. □

$$\frac{\mathsf{fields}(\mathtt{S}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}}}{\Theta \vdash \mathtt{new\ S}(\overline{\mathtt{v}}).\mathtt{f}_i \mapsto \mathtt{v}_i}\ (\textsc{ev-projnew})$$

$$\frac{\mathsf{mbody}(\mathtt{m}, \mathtt{S}) = (\overline{\mathtt{x}}, \mathtt{t}_0)}{\Theta \vdash \mathtt{new\ S}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}}) \mapsto \mathtt{t}_0[\overline{\mathtt{u}}/\overline{\mathtt{x}}, \mathtt{new\ S}(\overline{\mathtt{v}})/\mathtt{this}]}\ (\textsc{ev-invknew})$$

$$\frac{\Theta \vdash \mathtt{S} <: \mathtt{T}}{\Theta \vdash (\mathtt{T})\ \mathtt{new\ S}(\overline{\mathtt{v}}) \mapsto \mathtt{new\ S}(\overline{\mathtt{v}})}\ (\textsc{ev-castnew})$$

$$\frac{\Theta \vdash p \preceq q}{\Theta \vdash \mathtt{actsfor}(p, q)\ \mathtt{in\ t} \mapsto \mathtt{t}}\ (\textsc{ev-actsfor})$$

$$\frac{}{\Theta \vdash \mathtt{declassify}^{\mathtt{m}}(\mathtt{t}, l) \overset{l}{\mapsto} \mathtt{t}}\ (\textsc{ev-declassify})$$

$$\frac{\Theta \vdash \mathtt{t}_0 \overset{\Lambda}{\mapsto} \mathtt{t}_0'}{\Theta \vdash \mathtt{t}_0.\mathtt{f} \overset{\Lambda}{\mapsto} \mathtt{t}_0'.\mathtt{f}}\ (\textsc{ev-field})$$

$$\frac{\Theta \vdash \mathtt{t}_0 \overset{\Lambda}{\mapsto} \mathtt{t}_0'}{\Theta \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) \overset{\Lambda}{\mapsto} \mathtt{t}_0'.\mathtt{m}(\overline{\mathtt{t}})}\ (\textsc{ev-invk-recv})$$

$$\frac{\Theta \vdash \mathtt{t}_i \overset{\Lambda}{\mapsto} \mathtt{t}_i'}{\Theta \vdash \mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{t}_i, \overline{\mathtt{t}}) \overset{\Lambda}{\mapsto} \mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{t}_i', \overline{\mathtt{t}})}\ (\textsc{ev-invk-arg})$$

$$\frac{\Theta \vdash \mathtt{t}_i \overset{\Lambda}{\mapsto} \mathtt{t}_i'}{\Theta \vdash \mathtt{new\ S}(\overline{\mathtt{v}}, \mathtt{t}_i, \overline{\mathtt{t}}) \overset{\Lambda}{\mapsto} \mathtt{new\ S}(\overline{\mathtt{v}}, \mathtt{t}_i', \overline{\mathtt{t}})}\ (\textsc{ev-new-arg})$$

$$\frac{\Theta \vdash \mathtt{t}_0 \overset{\Lambda}{\mapsto} \mathtt{t}_0'}{\Theta \vdash (\mathtt{T})\ \mathtt{t}_0 \overset{\Lambda}{\mapsto} (\mathtt{T})\ \mathtt{t}_0'}\ (\textsc{ev-cast})$$

Fig. 11. Evaluation Rules

Theorem 3.5 FJifP Type Preservation. *If* $\Theta; \Gamma \vdash \mathtt{t} : \mathtt{S}$ *and* $\Theta \vdash \mathtt{t} \overset{\Lambda}{\mapsto} \mathtt{t}'$, *then there exists* Θ' *such that* $\Theta'; \Gamma \vdash \mathtt{t}' : \mathtt{S}'$ *for some* $\Theta \vdash \mathtt{S}' <: \mathtt{S}$. *Additionally, if* $\Lambda = \emptyset$, *then* $\Theta' = \Theta$.

Proof. Proof by cases based on which evaluation rule is used. □

3.7 Noninterference

To show that FJifP properly enforces security for well-typed programs, we show that the evaluation of a term t does not interfere with data lower than the declassifications that the term performs. To this end, we define a relation $\approx$ on FJifP

terms. The judgement is written $\Theta; \Lambda \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}$: "under security context Θ and allowed flows Λ, the terms $\mathtt{t}_1$ and $\mathtt{t}_2$ are observationally equivalent at security type $\mathtt{S}$ to an observer at security level ζ."

Noninterference only holds for programs that do not release data to level ζ. For example, if a method declassifies data to {Bob:}, then $\Lambda = \{Bob :\}$. If the observer is at $\zeta = \{Public :\}$, no information is leaked.

Two values $\mathtt{v}_1$ and $\mathtt{v}_2$ are equivalent to security level ζ with permitted flows Λ if, from the view of level ζ, doing the same field accesses or method calls evaluates to terms that are also observationally equivalent at ζ. Two terms are equivalent if they evaluate to equivalent values without declassifying information below ζ. We do not consider termination leaks.

Where our security theorem differs from traditional logical relations-style definitions of noninterference is that we consider equivalence of values only under methods that permit flows in Λ. This allows for a finer-grained definition of security: two values may not be in general observationally equivalent, but they may be used by the program in a safe way. For example, two classes may have different implementations of a method `release` that releases data to public, but this method may not be used by a program.

To capture what declassifications a program term might perform, we define the function $\mathsf{rtflows}$, which contains all of the possible declassifications that a term might perform during evaluation. As this is necessarily runtime information, it cannot be exactly captured by a static analysis. The main result of this section requires only a conservative approximation of $\mathsf{rtflows}$ for language security.

Definition 3.6 Runtime Flows. *Define the function* $\mathsf{rtflows}(\Theta, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{t})$*:*

$$\mathsf{rtflows}(\Theta, \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{t}) = \bigcup \{\Lambda \mid \Theta \vdash \mathtt{t}[\overline{\mathtt{u}}/\overline{\mathtt{x}}] \overset{\Lambda}{\hookrightarrow} \mathtt{v} \text{ for some } \overline{\mathtt{u}} \text{ such that } \Theta \vdash \overline{\mathtt{u}} : \overline{\mathtt{T}}\}$$

Suppose for method $\mathtt{m}$, $\mathsf{mbody}(\mathtt{m}, \mathtt{S}) = (\overline{\mathtt{x}}, \mathtt{t})$ *and* $\mathsf{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}} \to \mathtt{T}$. *We overload* $\mathsf{rtflows}$ *on methods* $\mathtt{m}$ *in type* $\mathtt{S}$.

$$\mathsf{rtflows}(\Theta, \mathtt{m}, \mathtt{S}) = \mathsf{rtflows}(\Theta, \overline{\mathtt{x}} : \overline{\mathtt{T}}; \mathtt{this} : \mathsf{container}(\mathtt{m}, \mathtt{S}), \mathtt{t})$$

Definition 3.7 Observational Equivalence. *Under security context* Θ*, two terms* $\mathtt{t}_1$ *and* $\mathtt{t}_2$ *are observationally equivalent at security type* S *to security level* ζ *in the presence of declassifications* Λ*, written* $\Theta; \Lambda \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}$*, if:*

—*There exists* $\Theta' \supseteq \Theta$ *such that* $\Theta' \vdash \mathtt{t}_1 : \mathtt{S}$ *and* $\Theta' \vdash \mathtt{t}_2 : \mathtt{S}$

—*Suppose* $\mathtt{t}_1 \equiv \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}})$ *and* $\mathtt{t}_2 \equiv \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}})$*. Then if either* $\Theta \vdash \mathsf{lab}(\mathtt{S}) \sqsubseteq \zeta$ *or* $\Theta \vdash l_i \sqsubseteq \zeta$ *for* $l_i \in \Lambda$*:*

(*1*) *For all* $\mathtt{T}_i\ \mathtt{f}_i \in \mathsf{fields}(\mathtt{S})$, $\Theta \vdash \mathtt{v}_i \approx_\zeta \mathtt{w}_i : \mathtt{T}_i$.

(*2*) *For all* $\mathtt{m} \in \mathsf{methods}(\mathtt{S})$ *with* $\mathsf{rtflows}(\Theta, \mathtt{m}) \subseteq \Lambda$*, and* $\mathsf{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}} \to \mathtt{T}$*, then for all* $\overline{\mathtt{u}}$ *such that* $\Theta \vdash \overline{\mathtt{u}} : \overline{\mathtt{T}}$, $\Theta; \Lambda \vdash \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}}) \approx_\zeta \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}}).\mathtt{m}(\overline{\mathtt{u}}) : \mathtt{T}$.

—*Otherwise, if* $\Theta \vdash \mathtt{t}_1 \hookrightarrow \mathtt{v}_1$ *and* $\Theta \vdash \mathtt{t}_2 \hookrightarrow \mathtt{v}_2$*, then* $\Theta; \Lambda \vdash \mathtt{v}_1 \approx_\zeta \mathtt{v}_2 : \mathtt{S}$.

We present two subtyping results. The first states that if observational equivalence holds at a type $\mathtt{S}'$, then it holds at a supertype $\mathtt{S}$ of $\mathtt{S}'$. The second states that if l' is one of the allowed flows in observational equivalence, then values observationally equivalent a a label l are also observationally equivalent at l'.

LEMMA 3.8 SECURITY SUBTYPING 1. *Suppose* $\Theta;\Lambda \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}'$ *and* $\Theta \vdash \mathtt{S}' <: \mathtt{S}$. *Then* $\Theta;\Lambda \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}$.

PROOF. Proof is by cases as to whether or not $\mathtt{t}_1$ and $\mathtt{t}_2$ are both values.

First suppose $\mathtt{t}_1 \equiv \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}})$ and $\mathtt{t}_2 \equiv \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}})$. Note that $\mathsf{fields}(\mathtt{S}') \subseteq \mathsf{fields}(\mathtt{S})$, so if $\Theta \vdash \mathsf{lab}(\mathtt{S}) \sqsubseteq \zeta$, then for all $\mathtt{T}_i\ \mathtt{f}_i \in \mathsf{fields}(\mathtt{S})$, $\Theta \vdash \mathtt{v}_i \approx_\zeta \mathtt{w}_i : \mathtt{T}_i$ as required.

For all $\mathtt{m} \in \mathsf{methods}(S)$ such that $\mathsf{rtflows}(\Theta, \mathtt{m}) \subseteq \Lambda$, we have

$$\mathsf{mtype}(\mathtt{m}, \mathtt{S}') = \mathsf{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}} \to \mathtt{T}$$

From the assumption $\Theta;\Lambda \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}'$, we have for all $\overline{\mathtt{u}}$ such that $\Theta \vdash \overline{u} : \overline{\mathtt{T}}$,

$$\Theta;\Lambda \vdash \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}}) \approx_\zeta \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}}).\mathtt{m}(\overline{\mathtt{u}}) : \mathtt{T}$$

This is the required result.

Otherwise $\mathtt{t}_1$, $\mathtt{t}_2$ or both are not values; so $\Theta \vdash \mathtt{t}_1 \hookrightarrow \mathtt{v}_1$ and $\Theta \vdash \mathtt{t}_2 \hookrightarrow \mathtt{v}_2$ with $\Theta \vdash \mathtt{v}_1 \approx_\zeta \mathtt{v}_2 : \mathtt{S}'$. By the above, $\Theta \vdash \mathtt{v}_1 \approx_\zeta \mathtt{v}_2 : \mathtt{S}$ and so $\Theta \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}$ as required. □

LEMMA 3.9 SECURITY SUBTYPING 2. *Suppose* $\Theta;\Lambda \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{C}\langle\overline{p}\rangle\{l\}$ *and* $l' \in \Lambda$. *Then* $\Theta;\Lambda \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{C}\langle\overline{p}\rangle\{l'\}$.

PROOF. Proof by cases on the terms $\mathtt{t}_1, \mathtt{t}_2$. Theorem 3.5 guarantees that there exists $\Theta' \supseteq \Theta$ such that $\Theta' \vdash \mathtt{t}_i : \mathtt{S}$ for $i \in \{1, 2\}$.

Suppose $\mathtt{t}_1 \equiv \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}})$ and $\mathtt{t}_2 \equiv \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}})$.

Suppose $\Theta \vdash l' \sqsubseteq \zeta$. As $l' \in \Lambda$, $\mathsf{fields}(\mathtt{C}\langle\overline{p}\rangle\{l'\}) = \mathsf{fields}(\mathtt{C}\langle\overline{p}\rangle\{l\})$, and $\mathsf{methods}(\mathtt{C})\overline{p}l' = \mathsf{methods}(\mathtt{C}\langle\overline{p}\rangle\{l\})$, conditions 1 and 2 in the definition of $\approx_\zeta$ hold.

The reasoning for the case where $\mathtt{t}_1$ and $\mathtt{t}_2$ are not both values proceeds identically to this case in the proof of 3.8.

□

We now show that if $\mathtt{t}$ is typeable to $\mathtt{S}$ under security context Θ and context $\mathtt{x} : \mathtt{S}_0$ with allowed flows Λ, then $\mathtt{x}$ can be replaced in $\mathtt{t}$ by any values $\mathtt{v}_1, \mathtt{v}_2$ of type $\mathtt{S}_0$ that are observationally equivalent to an attacker of level ζ under Θ and Λ. The terms $\mathtt{t}[v_1/\mathtt{x}]$ and $\mathtt{t}[v_2/\mathtt{x}]$ are then observationally equivalent to an attacker at level ζ.

THEOREM 3.10 SECURITY. *Suppose* $\Theta;\mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t} : \mathtt{S}$, $\mathsf{rtflows}(\Theta, \mathtt{x} : \mathtt{S}_0, \mathtt{t}) \subseteq \Lambda$, *and* $\Theta;\Lambda \vdash \mathtt{v}_1 \approx_\zeta \mathtt{v}_2 : \mathtt{S}_0$. *Then* $\Theta;\Lambda \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \mathtt{t}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}$.

PROOF. Proof proceeds by induction on the typing derivation.

Suppose $\Theta;\mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t}.\mathtt{f}_i : \mathtt{U}$. Then by inversion the last rule used was (TP-FIELD) and so $\Theta;\mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t} : \mathtt{S}$, $\mathtt{T}_i\ \mathtt{f}_i \in \mathsf{fields}(\mathtt{S})$, and $\mathtt{U} \equiv \mathtt{T}_i \sqcup \mathsf{lab}(\mathtt{S})$. By applying induction to the typing derivation, we have

$$\Theta \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \mathtt{t}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}$$

Therefore, if we have evaluations

$$\Theta \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \hookrightarrow \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}})$$
$$\Theta \vdash \mathtt{t}[\mathtt{v}_2/\mathtt{x}] \hookrightarrow \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}})$$

We must have $\Theta \vdash \texttt{new}\ \mathtt{S}_1(\overline{\mathtt{v}}) \approx_\zeta \texttt{new}\ \mathtt{S}_2(\overline{\mathtt{w}}) : \mathtt{S}$. We therefore have

$$\Theta \vdash \mathtt{t}_0[\mathtt{v}_1/\mathtt{x}].\mathtt{f}_i \hookrightarrow \mathtt{v}_i \;\; \Theta \vdash \mathtt{t}_0[\mathtt{v}_2/\mathtt{x}].\mathtt{f}_i \hookrightarrow \mathtt{w}_i$$

If $\Theta \vdash \mathsf{lab}(\mathtt{T}_i \sqcup \mathsf{lab}(\mathtt{S})) \sqsubseteq \zeta$, then $\Theta \vdash \mathsf{lab}(\mathtt{S}) \sqsubseteq \zeta$, and so $\Theta \vdash \mathtt{v}_i \approx_\zeta \mathtt{w}_i : \mathtt{T}$ by condition 1 in the definition of observationally equivalent values. By Lemma 3.8, $\Theta \vdash \mathtt{v}_i \approx_\zeta \mathtt{w}_i : \mathtt{T} \sqcup \mathsf{lab}(\mathtt{S}) = \mathtt{U}$. This is the required result.

Suppose the last typing rule used was (TP-INVK). By inversion we have the typing

$$\frac{\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t}_0 : \mathtt{T} \;\; \mathsf{mtype}(\mathtt{m}, \mathtt{T}) = \overline{\mathtt{S}} \to \mathtt{S} \quad \Theta; \mathtt{x} : \mathtt{S}_0 \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}'} \;\; \Theta \vdash \overline{\mathtt{S}'} <: \overline{\mathtt{S}}}{\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) : \mathtt{S} \sqcup \mathsf{lab}(\mathtt{T})}$$

Applying the induction hypothesis, we have

$$\Theta \vdash \mathtt{t}_0[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \mathtt{t}_0[\mathtt{v}_2/\mathtt{x}] : \mathtt{T}_0$$
$$\Theta \vdash \overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}'_0$$

We must show observational equivalence of the method calls by considering the result of evaluating the terms $\mathtt{t}_0[\mathtt{v}_1/\mathtt{x}].\mathtt{m}(\overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}])$ and $\mathtt{t}_0[\mathtt{v}_2/\mathtt{x}].\mathtt{m}(\overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}])$. Suppose we have the evaluations:

$$\Theta \vdash \mathtt{t}_0[\mathtt{v}_1/\mathtt{x}].\mathtt{m}(\overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}]) \overset{\Lambda_1}{\hookrightarrow} \mathtt{v}'_1$$
$$\Theta \vdash \mathtt{t}_0[\mathtt{v}_2/\mathtt{x}].\mathtt{m}(\overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}]) \overset{\Lambda_2}{\hookrightarrow} \mathtt{v}'_2$$

We know $\Lambda_1 \cup \Lambda_2 \subseteq \mathsf{rtflows}(\Theta, \mathtt{m}) \subseteq \mathsf{rtflows}(\Theta, \mathtt{x} : \mathtt{S}_0, \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}})) \subseteq \Lambda$. Therefore

$$\Theta; \Lambda \vdash \mathtt{v}'_1 \approx_\zeta \mathtt{v}'_2 : \mathtt{S}$$

by condition 2 in the definition of observational equivalence. This is the required result.

Suppose the last typing rule used was (TP-DECLASSIFY). By inversion

$$\frac{\Theta; \Gamma \vdash \mathtt{t} : \mathtt{C}\langle\overline{p}\rangle\{l_0\} \quad l \in \mathsf{flows}(\Theta, \mathtt{m}, l_0)}{\Theta; \Gamma \vdash \texttt{declassify}^{\mathtt{m}}(\mathtt{t}, \mathtt{C}\langle\overline{p}\rangle\{l\}) : \mathtt{C}\langle\overline{p}\rangle\{l\}}$$

By induction, $\Theta; \Lambda \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \mathtt{t}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}_0$. Therefore, if there are evaluations

$$\Theta \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \hookrightarrow \texttt{new}\ \mathtt{S}_1(\overline{\mathtt{u}})$$
$$\Theta \vdash \mathtt{t}[\mathtt{v}_2/\mathtt{x}] \hookrightarrow \texttt{new}\ \mathtt{S}_2(\overline{\mathtt{w}})$$

then

$$\Theta; \Lambda \vdash \texttt{new}\ \mathtt{S}_1(\overline{\mathtt{u}}) \approx_\zeta \texttt{new}\ \mathtt{S}_2(\overline{\mathtt{w}}) : \mathtt{C}\langle\overline{p}\rangle\{l_0\}$$

By the definition of $\mathsf{rtflows}$, we know that $l \in \mathsf{flows}(\mathtt{S}, \mathtt{x} : \mathtt{S}_0, \texttt{declassify}^{\mathtt{m}}(\mathtt{t}, l))$, and so $l \in \Lambda$. By Lemma 3.9, we have

$$\Theta; \Lambda \vdash \texttt{new}\ \mathtt{S}_1(\overline{\mathtt{u}}) \approx_\zeta \texttt{new}\ \mathtt{S}_2(\overline{\mathtt{w}}) : \mathtt{C}\langle\overline{p}\rangle\{l\}$$

Therefore,

$$\Theta; \Lambda \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \mathtt{t}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}$$

This is the required result.

Suppose $\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{x} : \mathtt{S}$. Then as $\Theta \vdash \mathtt{v}_1 \approx_\zeta \mathtt{v}_2 : \mathtt{S}_0$, $\Theta \vdash \mathtt{x}[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \mathtt{x}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}_0$ as required. Suppose $\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{y} : \mathtt{S}$. Then as $\mathtt{y}[\mathtt{v}_1/\mathtt{x}] = \mathtt{y}[\mathtt{v}_1/\mathtt{x}] = \mathtt{y}$, the result is trivially true. (An expression is always observationally equivalent to itself.) Suppose $\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}) : \mathtt{S}$. By inversion:

$$\frac{\mathsf{fields}(\mathtt{S}) = \overline{\mathtt{S}}\ \overline{\mathtt{f}} \qquad \Theta; \mathtt{x} : \mathtt{S}_0 \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}'}\ \ \Theta \vdash \overline{\mathtt{S}'} <: \overline{\mathtt{S}}}{\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}) : \mathtt{S}}$$

By induction and Lemma 3.8,

$$\Theta; \Lambda \vdash \overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}] : \overline{\mathtt{S}}$$

Let there be evaluations

$$\Theta \vdash \overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}] \hookrightarrow \overline{\mathtt{v}}$$
$$\Theta \vdash \overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}] \hookrightarrow \overline{\mathtt{w}}$$

By the definition of $\approx$, $\Theta \vdash \overline{\mathtt{v}} \approx_\zeta \overline{\mathtt{w}} : \mathtt{S}$. To show $\Theta; \Lambda \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}) \approx_\zeta \mathtt{new}\ \mathtt{S}(\overline{\mathtt{w}}) : \mathtt{S}$, we need to show that all field acceses and methods on these values lead to observationally equivalent evaluations.

For the first case, we consider the fields of $\mathtt{S}$. For all $\mathtt{T}_i\ \mathtt{f}_i \in \mathsf{fields}(\mathtt{S})$,

$$\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}).\mathtt{f}_i \hookrightarrow \mathtt{v}_i$$
$$\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{w}}).\mathtt{f}_i \hookrightarrow \mathtt{w}_i$$

From induction, we have $\Theta; \Lambda \vdash \mathtt{v}_i \approx_\zeta \mathtt{w}_i : \mathtt{T}_i$.

For the first case, we consider the methods of $\mathtt{S}$ that have the required information flows. For all $\mathtt{m} \in \mathsf{methods}(\mathtt{S})$ such that $\mathsf{rtflows}(\Theta, \mathtt{m}) \subseteq \Lambda$ with $\mathsf{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}} \to \mathtt{T}$, $\mathsf{mbody}(\mathtt{m}, \mathtt{S}) = (\overline{\mathtt{y}}, \mathtt{t})$, and $\overline{\mathtt{u}}, \overline{\mathtt{u}'}$, we have

$$\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}]).\mathtt{m}(\overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}]) \mapsto \mathtt{t}[\mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}])/\mathtt{this}, \overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}]/\overline{\mathtt{y}}$$
$$\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}]).\mathtt{m}(\overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}]) \mapsto \mathtt{t}[\mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}])/\mathtt{this}, \overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}]/\overline{\mathtt{y}}$$

By Theorem 3.5, as the above evaluation uses the rule (TP-INVK) and a reverse application of Lemma 3.3,

$$\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t}[\mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}})/\mathtt{this}, \overline{\mathtt{t}}/\overline{\mathtt{y}} : \mathtt{T}$$

As

$$\begin{aligned} \mathsf{rtflows}(\Theta, \mathtt{x} : \mathtt{S}_0, \mathtt{t}[\mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}})/\mathtt{this}], \overline{\mathtt{t}}/\overline{\mathtt{y}}) &\subseteq \mathsf{rtflows}(\Theta, \mathtt{x} : \mathtt{S}_0, \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}).\mathtt{m}(\overline{\mathtt{t}})) \\ &\subseteq \mathsf{rtflows}(\Theta, \mathtt{x} : \mathtt{S}_0, \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}})) \\ &\subseteq \Lambda \end{aligned}$$

by induction we have

$$\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}).\mathtt{m}(\overline{\mathtt{t}})[\mathtt{v}_1/\mathtt{x}] \approx_\zeta \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}).\mathtt{m}(\overline{\mathtt{t}})[\mathtt{v}_2/\mathtt{x}] : \mathtt{T}$$

By the definition of $\approx_\zeta$, we thus have

$$\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}[\mathtt{v}_1/\mathtt{x}]) \approx_\zeta \mathtt{new}\ \mathtt{S}(\overline{\mathtt{t}}[\mathtt{v}_2/\mathtt{x}]) : \mathtt{T}$$

This is the required result. The proof for the actsfor and cast cases follow from straightforward application of induction. □

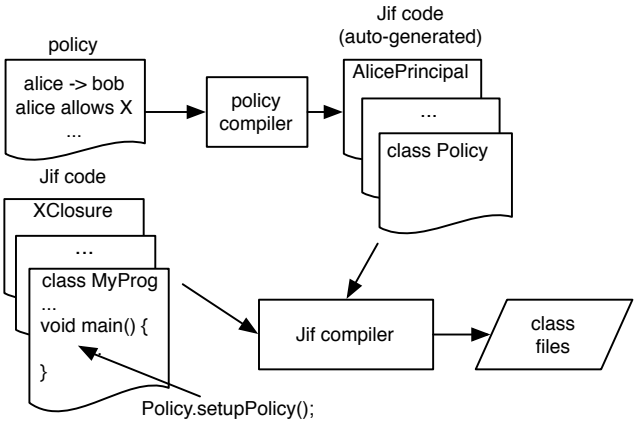

Fig. 12. Integrating an external policy into Jif.

4. POLICY FRAMEWORK IMPLEMENTATION

We implemented our trusted declassifiers in Jif [Myers et al. 2001]. In this section, we first describe how we compile an external policy into Jif code and access it from a Jif program. Then we comment on our approach, relating it to FJifP.

4.1 Compiling policy into Jif

We have developed a simple policy language for introducing principals and describing the delegations and declassifiers allowed by each principal. We built a small translator to compile policies into Jif code. Its use is illustrated in Figure 12. The translator automatically generates principal class definitions as well as a `Policy` class. The `Policy` class instantiates these principals and establishes the delegations described in the policy. In order to use our system, a programmer must provide a policy file (such as in Figure 2), an application and the declassifiers mentioned in the policy file. This policy is applied to the application by adding a single line to the starting point of the application. Finally, the automatically generated files must be compiled (other than the one line inserted into the main application file, all other files in the application do not need to be changed and thus do not need to be re-compiled).

Our policy language currently consists of only two kinds of statements (and described earlier in Section 2): –>-rules corresponding to delegations and `allow`-rules, establishing trust in declassifiers. The syntax is shown in Figure 13. There is a special `allow` rule, `allow None`. Since a principal must be used in a rule in order to be added to the system, a principal, `p`, which trusts no declassifiers and has no delegations should be added with the special policy, `p` **allows** `None`. The policy compiler takes policies and produces Jif code. To understand the Jif code, a brief explanation of Jif `Principal`s and `Closure`s is necessary.

The Jif `Principal` class has methods for adding delegations called `addDelegate` and for checking authorizations called `isAuthorized`. Our policy compiler leverages this interface by automatically generating Principal subclasses which override the

principal	p ::=	`alice` | `bob` | ...
declassifier	D ::=	`method1` | `method2` | ...
delegation	Del ::=	p −> p
trust stmt	$Allow$ ::=	p **allows** $D(p)$ | p **allows** `None`
policy stmts	$Stmt$ ::=	$(Del \mid Allow)*$

Fig. 13. Policy language syntax.

authorization method in order to authorize only the declassifiers mentioned in `allow` statements in the given policy file. To establish the delegations given by −>-rules, code is automatically generated for the `Policy.setupPolicy` method. This method instantiates each principal and uses the principal's `addDelegate` method to perform the delegations given in the policy file. This gives the desirable result that, after writing the policy in a simple syntax, the programmer merely has to invoke the `Policy.setupPolicy` method at the beginning of an application in order to put the policy into effect.

We implement declassifiers using Jif's `Closure` class. The Jif `Closure` class provides a way of packaging up a function with some arguments and then treating it as a first-class value. More importantly, it is parameterized by a principal, whose authorization it needs in order to execute. This authorization is sought from the principal's `isAuthorized` method when it is invoked. By requiring that all declassifications take place in `Closure`s, we can be sure that all declassifications will consult the policy before they execute.

Consider the example policy in Figure 2. Compiling this policy generates classes for `AlicePrincipal`, `DrBobPrincipal`, `DrJohnPrincipal` and `ChuckPrincipal`, as well the `Policy` class with a `setupPolicy` method that instantiates each class and performs the indicated delegations. The principals are automatically generated such that the `isAuthorized` method gives authorization to the `Closure`s named in the `allow` statements in the policy file. The `AlicePrincipal` class, for example, allows for the `Passwd.check`(**public**) and `AES.encrypt`(**public**) closures to operate on data labeled with a policy owned by Alice. The declassifier in the `allow` statements is parameterized by a principal which indicates the lowest possible security level to which the method may declassify.

4.1.0.1 *Adding a declassifier.* An important benefit of our system is that adding a declassifier is simple. Consider a declassifier for triple DES encryption. In our system, this would require the programmer to provide a `Closure` to call the encryption function and do the declassification, as shown in Figure 14. In order to use this `Closure` to encrypt and declassify some plaintext, the principal who owns the plaintext must authorize `TripleDESClosure`. This authorization must be established through the policy file with a command such as:

`Alice` **allows** `crypto.TripleDESClosure`(**public**)

This command is automatically translated into a line of Jif code in the automatically generated `AlicePrincipal` class.Once this has been done, the programmer simply needs to use the declassifier by first instantiating the closure class with the

```
public class TripleDESClosure[principal P,label L]
implements Closure[P,{P:}] {
   byte{P:}[]{P:} plaintext;
   Key{P:} key;
   ...
   public Object{this} invoke{P:}() where caller(P) {
      return declassify( TripleDES[{P:}].encrypt( key, plaintext ), {this} );
}}
```

Fig. 14. A closure for declassifying the cipher text generated by triple DES encryption. The standard constructor is defined, but not displayed.

particular arguments that are to be used. Then Jif's built-in authorize method must be called with the principal and the declassifier closure as arguments:

```
principalUtil.authorize(...)
```

This built-in method calls the principal's `isAuthorized` method and if it authorizes the `Closure`, it allows the `Closure` to be executed.

4.2 Relating the implementation to FJifP

In FJifP, typing and evaluation take place in the presence of a security context Θ, which contains an acts-for hierarchy, Δ and a declassification policy, Υ. The implementation of the acts-for hierarchy is straight-forward; all delegation statements indicated by $->$-rules in the policy file are automatically generated in the `Policy.setupPolicy` method. We implement Υ by first defining all the principals that may be used in the program. Recall that Υ contains triples $(\mathtt{m}, p, q)$. These correspond to `allow` statements in the policy written `p` **allows** `m(q)`. Such `allow`-statements correspond to lines of Jif code in the particular `Principal` class definitions, such that exactly the methods in Υ relating to a particular principal are explicitly allowed by that principal's `isAuthorized` method. For example, if $p = \mathsf{Alice}$ then for all triples $(\mathtt{m}, \mathsf{Alice}, q)$, the `isAuthorized` method for the `AlicePrincipal` class explicitly allows closures `m` with return type, q. In our example, this would be `TripleDES.encrypt`(**public**) and `Passwd.check`(**public**).

In order to faithfully implement FJifP, and achieve *noninterference modulo trusted methods*, we must place some restrictions on Jif's principals and declassification mechanism:

(1) We require that no declassification may take place other than through `Closures`. This is because all declassifications should first consult the declassification context, which is distributed throughout the `Principal` classes in our implementation. Since `Closures` require an authorization before they may be executed, they will always consult the principal whose data they are trying to declassify, to make sure that the newly introduced flow is allowed by policy.

(2) We require that no new principals are introduced other than the ones introduced

in the `Policy.setupPolicy` method which is automatically generated from the policy file.

(3) We require that no delegations are established or revoked, other than the ones introduced in the `Policy.setupPolicy` method which is automatically generated from the policy file.

These restrictions present only minor limitations to the language. The declassification restriction does not limit the expressive power of Jif at all, since it would be possible to wrap every declassify statement in a `Closure` and add the appropriate **allows** statements to the policy. The restrictions on the principal hierarchy could be somewhat more serious. By requiring that all principals and delegations are established at the outset of the program, this would disallow dynamic updates to the security policy. Currently, however, the mechanism for dynamic updating in Jif is arguably unsafe [Swamy et al. 2006], and needs revision. Additionally, the static, global nature of the acts-for hierarchy is less critical for our approach and it is easy to imagine this restriction could be adapted to work with safe and secure dynamic updates.

One difference between FJifP and our Jif implementation is in the enforcement of the security policy. Jif is currently configured to do all delegations and policy authorizations using a runtime mechanism. Although we use this runtime mechanism, the Jif compiler could be modified to check the policy at compile-time. Our restrictions force delegations and declassifications to be static, global entities. Thus, the policy must be established at the outset of the program and the policy checks could be integrated into the type-checker, which would give static enforcement, as we have in FJifP.

5. JPMAIL: A PRACTICAL APPLICATION

We evaluate the practicality of our approach by using trusted declassification to implement several significant applications, including an email client, called JPmail, a chat client, chat server, and a logrotate UNIX tool. We focus on JPmail here. JPmail uses several declassifiers, including a variety of symmetric and asymmetric encryption declassifiers for sending sensitive data to an insecure mail server, a password hashing method, as well as other filter declassifiers which filter e-mails for certain recipients. Principals can choose which encryption and filter declassifiers they trust, merely by changing a few lines in the policy file. Likewise, email recipient groups may be changed by changing a few lines in the policy file.

5.1 JPmail Overview

An email system is particularly useful for the study of application development in security-typed languages. This is not only because email is ubiquitous, but also because it has been a frequent avenue for security leaks. Moreover, email has a wide variety of security policies that it might need to enforce, from military multi-level security to organizational hierarchies. Finally, email policy is naturally distributed, with unique principals interacting across potentially distant clients. We seek to support policies that involve these diverse and dynamic principals.

Illustrated in Figure 15, the *JPmail system* (*JP = Jif/Policy*) consists of three main components: JPmail clients, the Internet and public mail servers. Written

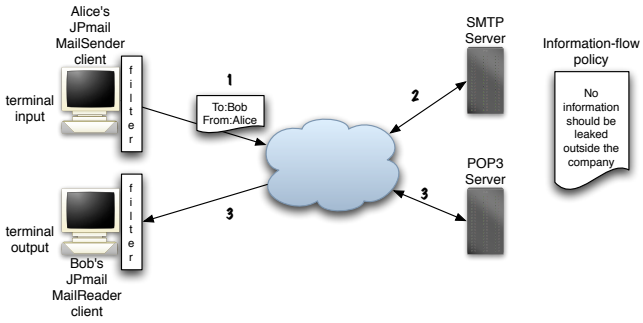

Fig. 15. Sending email

in Jif, the *JPmail client* (or just JPmail throughout) is a functional email client implementing a subset of the MIME protocol. The JPmail client software consists of three software components: a POP3-based mail reader, an SMTP-based mail sender and a policy store. The client provably enforces security policy from end to end (sender to recipient). Policy is defined with respect to a principal hierarchy. Each environment defines principal hierarchies representative of their organizational rights structure.

5.2 Security policy

The high-level security policy we defined for JPmail is: *The body of an email should be visible only to the authorized senders and receivers.* We make two clarifications about this policy. Firstly, in this work, we are only concerned with privacy (confidentiality). Secondly, our email client is not inherently limited to sending email only to authorized receivers. The way JPmail handles unauthorized recipients depends on the user-defined policy.

We make the following assumptions. The JPmail-local file systems are trusted to store information securely, based on the access control list on a given file (thus if a file is readable only by the user, it is considered safe from leakage). Internet communication is generally untrustworthy, and is deemed as *public* channels throughout. The SMTP and POP3 servers are not written in Jif, and do not enforce any security policy save that which is provided by their implementation and administration. For the purposes of this work, we assume nothing about the servers' ability to prevent leakage of user data: i.e., any information sent to them is deemed *public*.

Consider some dangers in email. 1) In the case of a malicious insider, email was used to leak classified documents [Ross and Esposito 2005]. 2) In another case, a programmer mistake led to a privacy violation for pharmaceutical clients [Federal Trade Commission 2002]. 3) An email application also handles passwords for logging into remote servers and could leak a password by sending it to the server as plaintext (a protocol that some servers use, in fact). 4) An email client that uses PGP could accidentally or maliciously leak keys.

5.3 JPmail Security Evaluation

We evaluate the enforcement of our security policy: "*The body of an email should be visible only to the authorized senders and receivers.*" by examining small sections of the JPmail code, policy and declassifiers. We examine the label on emails and cross-check it with the policy file to see what information flows are possible with that label. The body of the email is initially labeled with the sender's principal. The email is then passed through a declassifier, `DeclassMsgBody` which relabels the body to {`rcpt`:} (derived from the To: field in the email header) so long as `rcpt` is one of the allowed recipient principals, given in the policy file. From this point, the `rcpt` policy governs all information flows. Namely, before this email can be placed on the public socket and sent to the SMTP server, it must pass through another declassifier based on `rcpt`'s policy. If `rcpt` allows for AES and RSA encryption, then the email is encrypted using a one-time randomly generated key (which cannot be leaked, because it is labeled `rcpt`) and that key is then encrypted with the recipient's public key. After the declassifications, both email body and key are considered public and can be sent to the SMTP server. Without the declassifications, the program would not type-check.

We can repeat this evaluation for other sensitive data such as keys and passwords. For these items the analysis is even simpler, because they are not dynamically labeled like emails (which depend on user input). The password is given a label when a `MailSenderCrypto` object is created. Checking the policy file, we can see that Strings cannot be declassified except through an MD5 filter. Creating an MD5 hash is necessary for authentication with the mail server. This was made clear when we tried to send the password as plaintext over the mail server socket when establishing a connection. This insecure practice was automatically disallowed by Jif with JPolicy.

There are other caveats to security that must not be overlooked. Firstly, the security properties of a program are dependent on the correctness of the Jif compiler (and our policy compiler). Secondly, the security properties may also be dependent on supporting infrastructures. This includes the correctness of encryption libraries and the strength of used cryptographic algorithms, the protection on keystores and correctness of public-key cryptographic libraries as well as the security enforced by the local file system. Moreover, for the system to be secure, the enforced policy must be consistent across all clients.

One advantage of Jif is that it forces the programmer to think in terms of information flows and to consider security concerns from the outset. Interestingly, there is a strong consensus in the software engineering community that performing these kinds of security analysis at design time is essential to the security of the resulting system [Devanbu and Stubblebine 2000]. On the other hand, security by design in advance can cause problems if it is too rigid and tied to particular operating system constructs (i.e. insufficiently portable). This was the long-term experience of MULTICS [Karger and Schell 2002]. We developed our policy tools in the hopes of balancing the best of both worlds—security in advance by design along with expressive policy, flexibility and portability.

Implementing policy in our model was significantly easier than managing all the complex structures provided by Jif for principals, delegations and declassifications.

The ability to implement a policy by merely giving a series of delegation and `allow`-statements made the policy easier to construct and easier to manage. Furthermore, we found that it is quite beneficial to be able to understand all possible flows by merely examining the policy file.

Finally, we observed that the policy tool effectively decoupled policy from the programs that they govern. This allowed us to modify policy easily in order to accommodate different security models. By instrumenting the code during development with different options for each filter, we could implement distinct security models without altering the code. Furthermore, by gathering the policy into a single file, it was easier to do a security analysis and gauge what information flows could take place for a given principal, in contrast to leaving declassify-statements in the code.

6. RELATED WORK

This work falls into a long line of research on using security-typed languages to enforce information flow control and variations of noninterference [Sabelfeld and Myers 2003]. Our work is substantively distinguished from related work in two ways: our model of declassification, and the way that permitted declassifications are expressed in the security policy and implemented in our policy framework.

FJifP permits declassifications to occur only within trusted methods, where the added trust granted those methods (in terms of information flows that contrary to the acts-for hierarchy Δ) is part of a global policy Υ. Following the Sabelfeld and Sands' declassification nomenclature [Sabelfeld and Sands 2008], our declassification mechanism is identified by *where* declassifications occur—within expressions **declassify** e that occur in trusted methods $\mathtt{m}$ mentioned in Υ—and *who* allows the declassifications, as indicated by the principal p where $(\mathtt{m}, p, q) \in \Upsilon$. A determination of *what* may be declassified requires (manual) analysis of the actual declassifications. For example, we may know that Alice allows her data to be leaked by a password check. By manually analyzing this declassifier, we could determine that only one bit of information is leaked per call. We would have to understand how the program uses the declassifier to ensure it is not called too often.

FJifP's **declassify** e construct resembles the language construct proposed for selective declassification [Myers 1999], robust declassification [Myers et al. 2006], and delimited release [Sabelfeld and Myers 2004], among others. One difference is that these prior mechanisms may downgrade e to an arbitrary label (under the right conditions, see below), while in FJifP the new label is restricted by the portion of declassification policy Υ that applies to the trusted method $\mathtt{m}$ in which the **declassify** appears. This restriction is similar to that imposed by *flow policies* F in the construct `flow` F `in` S proposed by Matos and Boudol [2005]. This construct specifies that the execution of statement S should allow additional flows according to F, which consists of statements of the form $A \preceq B$. Follow-on work [Boudol and Kolundzija 2007] constrains when this construct can be used according to a dynamic access control policy, similar to an approach taken by Tse and Zdancewic [2004]. We could implement our trusted declassification policy Υ using such a construct by, for each $(\mathtt{m}, p, q) \in \Upsilon$, changing the body S of $\mathtt{m}$ to be `flow` $(p \preceq q)$ `in` S. The actual occurrences of declassification within S, using this approach, are implicit,

while for trusted declassifiers they are explicit, aiding manual analysis.

Robust declassification considers the integrity of a program's control flow in deciding whether a declassification should be allowed. For example, data owned by Alice can only be robustly downgraded if the integrity of the control flow at the declassification point is *trusted* by Alice. At present, FJifP lacks this notion of robustness—it only considers the static policy Υ with respect to a declassifying method `m`, and not the program context in which `m` could be called. This notion of robustness could be incorporated into FJifP by changing how **declassify** is type checked; we could allow the programmer to augment Υ to specify intransitive integrity constraints similar to the confidentiality constraints now expressed.

Designating trusted methods as part of the global security policy reveals all potential information flows as part of that policy, simplifying policy analysis (e.g., as reachability in a graph like that shown in Figure 3). This is in contrast to the declassification mechanisms mentioned above, which require analyzing the program in conjunction with the global policy. Work by Smith and Thober [2007] proposes separately declaring information flow policies at the API level for input/output classes, and for methods that would perform declassification or endorsement (for integrity). The latter are similar in spirit to our trusted methods, but are implemented as simply in-lining selective declassification construct **declassify** around the returned value, rather than relaxing the policy within the body of the method. This approach is in the spirit of an in-lined reference monitor (IRM) [Erlingsson 2004], in which a reference monitor is compiled into a program to enforce a separately-declared policy.

If we also consider security labels as part of the policy, then two other recently-proposed systems also express all possible information flows as part of the policy: the *declassification policies* of Chong and Myers [2004] and the *downgrading policies* of Li and Zdancewic [2005a][2005b]. In both of these systems, the way that data may be declassified is expressed directly in that data's label.

Declassification policies p are expressed using security labels of the form $l \longrightarrow^{c} p$, where l is the initial security label of the data, but the label may be changed to label p when condition c is true. The initial label l may be another declassification label, or an atomic label drawn from a traditional security lattice. As an example, the label $S \longrightarrow^{\texttt{curryear}\geq 2010} P$ could express that a document currently labeled as *secret* (S) should be relabeled to be *public* (P) after the year 2010. Another example would be to label a document s with $S \longrightarrow^{\texttt{isAnonymized}(s)} P$ to state that s is considered *secret* until some function `isAnonymized` indicates it is safe to treat as public. The transition is witnessed by a selective declassification construct and is only permitted if the type checker can prove c is satisfied at the point of the declassification. Chong and Myers prove a property called *noninterference until* $c_1, ...c_k$, which is similar to our noninterference modulo trusted methods property, but refers to conditions rather than trusted methods. Declassification policies have been implemented in a recent extension to Jif 3.0 [Clarkson et al. 2008]. (We note that the *flow locks* of Broberg and Sands [2006] can specify similar temporal policies but require the programmer to explicitly indicate in the program text at which points release conditions have been met, confounding a separate policy analysis.)

Li and Zdancewic's downgrading policies are also expressed as labels, but they

indicate *what* data can be leaked. In particular, a label is a function whose input parameter represents secret data, and whose result can be considered public. In effect, the label of some data indicates precisely by what computation it may be leaked, and the enforcement mechanism can ensure that any program actions on data will not leak information in violation of its label's policy. For example, the identity function $\lambda x.x$ represents the label `public`, since as "input" it takes a secret value x, and then returns that value unchanged, making it publicly-visible. At the opposite extreme are labels described via constant functions; e.g., $\lambda x.c$ (where c is some constant integer) indicates that no matter what secret argument is passed in, the same constant value c is returned. The middle ground would be a label such as $\lambda p.(x = p)$ where p is a (secret) password and $=$ represents the equality operation (not assignment). This says that information about password p can be made public by comparing to the guess x; in other words, a public user will learn, each time the password checker is invoked, whether a public value x is or is not p. Since this leaks a small amount of information about p, it expresses a kind of downgrading. Li and Zdancewic prove that their system enjoys a property they call *relaxed noninterference*, which essentially states that the only information leaked is that according to the policy (lattice and labels) [Sabelfeld and Myers 2004]. They have also adapted their approach to integrity policies [Li and Zdancewic 2005b].

A disadvantage of both of these approaches is that a security analyst must review the code of an application, and in particular the labels of each variable (which could themselves refer to program variables), to understand the impact of declassification operations on the overall policy goals. The insight behind trusted declassifiers is that declassification operations can be mentioned in a policy that is separate from programs that use them if (1) declassification operations have a *name*, and (2) if the programmer has defined abstract conditions under which the named declassification operations may be used. On the other hand, global release policies may not always be appropriate—it may be more natural to specify declassification policies for particular data, i.e., as labels. Moreover, both of these systems say something about *what* is released (in Li and Zdancewic's system) and *when* it is released (in Chong and Myers' system), whereas trusted declassifiers say nothing about these concerns. A natural direction for further research would be to explore combining trusted declassifiers with elements of these approaches.

Trusted declassifiers and the above systems require label annotations within programs themselves. For our work, only the allowed declassification contexts and acts-for hierarchies are separate from the program. Sun et al. [2004] and Smith and Thober [2007] use whole-program inference and API-based specifications to avoid labeling in the program text entirely. *Labeling policies* similarly allow for a separate specification of how to label data read/written to/from external channels [Hicks et al. 2007]. This work could be combined with ours for fully separate security policies.

As discussed previously, the security property presented by Matos and Boudol [2005] allows adding local information flows through the expression $\mathsf{flow}\ F\ \mathsf{in}\ M$. The authors then define a non-disclosure policy, similar to noninterference with certain flows allowed, and then prove that a successful program typing under a flow context G indicates that the program satisfies the non-disclosure policy $\mathcal{ND}(G)$.

As the flow context G is expanded during a static typing to provide temporary flows in the case of a flow F in M expression, this provides a fine-grained result as to what conditions information can leak under; the only circumstances under which information can leak beyond G are these special flow allowance statements. The proof of our Theorem 3.10 also implies a result of fine-grained information leakage, as our notion of Λ contains all of the declassification flows possible in a program at runtime. An easy lemma shows that if $\Lambda' \leq \Lambda$, then observational equivalence at Λ implies observational equivalence at Λ'. This implies that those portions of the computation that perform declassification to Λ' are noninterfering below Λ'.

Askarov and Sabelfeld [2007] give a system that weakens noninterference with a policy for *gradual release*: at each step during the run of a program, the only information that is available to an observer (defined as *knowledge*) is that which has been already released by declassify statements. They then show that a standard type system (with minor modifications) over an imperative language with while-loops satisfies gradual release. As our system does not contain side effects, no information is released during a computation besides the final value that a term evaluates to. If we extended the language of FJifP to include side effects, the only alteration that we would need to make to the type system so that well-typed terms satisfied gradual release would be to disallow declassifications to levels L that occur with a program counter of label L' with $L \leq L'$ (for example, within either branch of a conditional of secrecy L').

7. CONCLUSION

In this paper, we have presented a security-typed, object-oriented language, FJifP, which incorporates declassification and delegation as authorized by an external, global policy. We have shown that this language satisfies a modified form of noninterference, *noninterference modulo trusted methods*, meaning that all violations of noninterference can be justified by the policy. Consequently, noninterference is maintained for principals which allow no declassifications (i.e. have no trusted declassifiers) in the policy (and no one who can act for them makes any declassifications). We implemented our policy and trusted declassification for Jif in the JPolicy framework by using a restricted form of Jif's selective declassification: we provide a policy compiler to compile simple policy files into Jif code and we restrict the use of Jif's **declassify** in a way that does not limit the expressive power of the language.

The JPolicy framework presents an important step forward in making STLs more practical. For real applications, exceptions must be made to the strong security property of noninterference for handling such operations as authentication, data encryption, anonymization and filtration. The key to our approach is in moderating the use of these exceptional functions, called declassifiers, through an external policy. In this way it is still possible to understand the end-to-end information flow properties of an application without having to inspect the code. Rather, it is only necessary to inspect the external policy to understand the security properties of an application running under that policy. This enables to apply the accepted systems principle of separating security policy from mechanism [Hansen 1970] to STLs.

Introducing a policy specification for a Jif program that introduces principals,

establishes delegations and also integrates declassification is a significant advance in improving the flexibility of policy in Jif. Such policies, separated from program code, can be understood, modified and analyzed apart from the programs in which they are used. This facilitates high-level reasoning about program policy not previously possible, opening the door for policy reconciliation analyses such as one recently defined for Jif and Security-Enhanced Linux policies [Hicks et al. 2007] and used for developing secure systems [Hicks et al. 2007b; 2007a].

To demonstrate the practicality of our approach, we have successfully applied our policy framework to some significant applications. We describe our experience with the Jif application, JPmail, and conclude the value of our approach. Building the JPmail application and understanding its security properties were greatly improved through the introduction of JPolicy, our policy framework incorporating trusted declassification.

REFERENCES

Askarov, A. and Sabelfeld, A. 2005. Secure implementation of cryptographic protocols: A case study of mutual distrust. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS '05)*. LNCS. Springer-Verlag, Milan, Italy.

Askarov, A. and Sabelfeld, A. 2007. Gradual release: Unifying declassification, encryption and key release policies. In *IEEE Symposium on Security and Privacy*. 207–221.

Bangeman, E. 2006. Complaint filed with FTC over AOL data exposure. `http://arstechnica.com/news.ars/post/20060815-7504.html`.

Boudol, G. and Kolundzija, M. 2007. Access Control and Declassification. In *Computer Network Security*. CCIS, vol. 1. Springer-Verlag, 85–98.

Broberg, N. and Sands, D. 2006. Flow locks: Towards a core calculus for dynamic flow policies. In *Proceedings of ESOP'06, European Symposium on Programming*. LNCS. Springer-Verlag, Vienna, Austria.

Chen, H. and Malacaria, P. 2007. Quantitative analysis of leakage for multi-threaded programs. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*. ACM Press, New York, NY, USA, 31–40.

Chong, S. and Myers, A. C. 2004. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM.

Chong, S., Vikram, K., and Myers., A. C. 2007. Sif: Enforcing confidentiality and integrity in web applications. In *Proceedings of 16th USENIX Security Symposium*.

Clark, D., Hunt, S., and Malacaria, P. 2007. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security 15,* 3, 321–371.

Clarkson, M. R., Chong, S., and Myers, A. C. 2008. Civitas: Toward a secure voting system. In *Proc. IEEE Symp. on Security and Privacy*. 354–367.

Devanbu, P. T. and Stubblebine, S. 2000. Software engineering for security: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. ACM Press, New York, NY, USA, 227–239.

Erlingsson, U. 2004. The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Ithaca, NY, USA. Adviser-Fred B. Schneider.

Federal Trade Commission. 2002. Eli Lilly settles FTC charges concerning security breach. Available at `http://www.ftc.gov/opa/2002/01/elililly.htm`.

Hansen, P. B. 1970. The nucleus of a multiprogramming system. *Commun. ACM 13,* 4, 238–241.

Hicks, B. 2007. Secure systems development using security-typed languages. Ph.D. thesis, Penn State.

Hicks, B., Misiak, T., and McDaniel, P. 2007. Channels: Runtime system infrastructure for security-typed languages. In *Proceedings of the Computer Security Applications Conference (ACSAC)*. 443–452.

Hicks, B., Rueda, S., Jaeger, T., and McDaniel, P. 2007a. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*. Number NAS-TR-0061-2007. Santa Clara, CA, USA.

Hicks, B., Rueda, S., Jaeger, T., and McDaniel, P. 2007b. Integrating selinux with security-typed languages. In *Proceedings of the 3rd SELinux Symposium*. Baltimore, MD, USA.

Hicks, B., Rueda, S., St. Clair, L., Jaeger, T., and McDaniel, P. 2007. A logical specification and analysis for SELinux MLS policy. In *Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT)*. Antipolis, France.

Igarashi, A., Pierce, B., and Wadler, P. 1999. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, L. Meissner, Ed. Vol. 34(10). N. Y., 132–146.

Karger, P. A. and Schell, R. R. 2002. Thirty years later: lessons from the multics security evaluation. *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, 119–126.

Li, P. and Zdancewic, S. 2005a. Downgrading policies and relaxed noninterference. In *Proc. 32nd ACM Symp. on Principles of Programming Languages (POPL)*.

Li, P. and Zdancewic, S. 2005b. Unifying confidentiality and integrity in downgrading policies. In *Proc. of Foundations of Computer Security Workshop*.

Lowe, G. 2002. Quantifying information flow. 18–31.

Mantel, H. and Sands, D. 2004. Controlled declassification based on intransitive noninterference. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Vol. 3302. Springer-Verlag, Taipei, Taiwan, 129–145.

Matos, A. and Boudol, G. 2005. On declassification and the non-disclosure policy. In *Proceedings of the Computer Security Foundations Workshop (CSFW'05)*.

Myers, A. C. 1999. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachussetts Institute of Technology, University of Cambridge. January. Ph.D. thesis.

Myers, A. C., Sabelfeld, A., and Zdancewic, S. 2006. Enforcing robust declassification. To appear in *Journal of Computer Security*.

Myers, A. C., Zheng, L., Zdancewic, S., Chong, S., and Nystrom, N. 2001. Jif: Java + information flow. Software release. Located at http://www.cs.cornell.edu/jif.

Ross, B. and Esposito, R. 2005. Espionage case breaches the white house. ABC news report. `http://abcnews.go.com/WNT/story?id=1187030`.

Sabelfeld, A. and Myers, A. C. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21,* 1 (January), 5–19.

Sabelfeld, A. and Myers, A. C. 2004. A model of delimited information release. *Lecture Notes in Computer Science 3222*, 174–191.

Sabelfeld, A. and Sands, D. 2008. Dimensions and principles of declassification. *Journal of Computer Security*. To appear.

Smith, S. F. and Thober, M. 2007. Improving usability of information flow security in java. In *PLAS '07: Proceedings of the 2007 workshop on Programming languages and analysis for security*. ACM, New York, NY, USA, 11–20.

Sun, Q., Banerjee, A., and Naumann, D. A. 2004. Modular and constraint-based information flow inference for an object-oriented language. In *Static Analysis, 11th International Symposium, SAS 2004*. 84–99.

Swamy, N., Hicks, M., Tse, S., and Zdancewic, S. 2006. Managing policy updates in security-typed languages. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*. IEEE Computer Society, Washington, DC, USA, 202–216.

Tse, S. and Zdancewic, S. 2004. Run-time Principals in Information-flow Type Systems. In *IEEE 2004 Symposium on Security and Privacy*. IEEE Computer Society Press.