Trusted Declassification

High-level policy for a security-typed language

Boniface Hicks Dave King
Patrick McDaniel
Penn State University
{phicks,dhking,mcdaniel}@cse.psu.edu

Michael Hicks
University of Maryland
mwh@cs.umd.edu

Abstract

Security-typed languages promise to be a powerful tool with which provably secure software applications may be developed. 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 more 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 that 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 and provide some of our experience using it to build a secure e-mail client.

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. 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.

Security-typed languages promise to be a valuable tool in making provably secure software applications. In such languages, 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:

$\texttt{String}_{\mathsf{Alice}}$ `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. 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 one bit of information about Alice's password, which is whether or not it is equal to the guess. Assuming that Alice does not delegate to the public, this program would not satisfy noninterference if ??? was public. But then the function is useless as a password checker.

To remedy this problem, practical security-typed languages 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 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. Another way of saying this is that 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 classifier 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.

This paper presents a global security policy system for a security-typed language, which extends existing work by allowing each principal to indicate which declassifiers it trusts. We call our approach *trusted declassification*. 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 [15], a full scale implementation of a security-typed language based on Java.

There has recently been a proliferation of work toward incorporating forms of declassification into security-typed languages [9, 4, 3, 10, 16, 8] as detailed in a recent survey [19]. Placed next to much of this work, what we propose is comparatively simple. Nonetheless, the value of our approach is borne out of practical experience. In particular, we and others [1] have been trying to build applications in Jif. Jif supports *selective declassification* [13], 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, we have used our framework to build an SMTP/POP3-compliant e-mail client called JPmail, and found that it made the process of reasoning about declassification and information flows far easier. Thus, we hope our work takes a step toward making security typed languages more practical.

The structure of the 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. In Section 5, we describe related work. In Section 6, we conclude and give future work.

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)
  {
     bool_{public} valid = Passwd<p>.check(guess,password);
     if (valid) name = newName;
  }
}
```

Figure 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
```

Figure 2. A simple policy

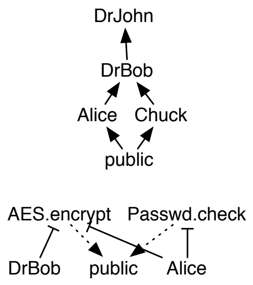

Figure 3. Example acts-for hierarchy and declassifier context.

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 bit at a time) via a password check. With this policy in hand, we can be certain, even without searching the whole program code for declassify statements, that Alice's whole, unencrypted data can *never* flow to Chuck.

In our language, security is enforced statically by the type-checker, by disallowing programs which 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. Note, that our implementation of this in Jif has a more dynamic behavior. We explain this further in Section 4.

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 should not be large, 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.

3. Semantics and properties of FJifP

3.1 Introduction to FJifP

We first describe the FJifP(short for Featherweight Jif with Policy), a security-typed, object-oriented language. FJifP is an extension of Featherweight Java [7] 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 (we will replace these features with our own declassification mechanism about which we can prove some security properties) as well as exceptions[1]. We also omit some labels from Jif which are required for checking a pc-label in order to prevent illegal implicit flows (flows introduced by the control path). Because we do not have state, we are able to capture implicit flows without the use of a pc-label. To additionally simplify the presentation of our system, we omit two mechanisms of FJ: constructors[2] and unrestricted casts. These features were originally included in FJ to ensure every FJ program was also a Java program. In FJifP, it is sufficient to consider upcasts: unrestricted casting can be easily added back to the language.

[1] Covering exceptions in a security-typed language has been covered elsewhere in the literature [17].

[2] The basic constructor which simply assigns input parameters to member variables is, of course, provided.

```
class MedicalRecord<α> ◁ 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>_{pub} updateName(String_{public} guess,
                                    String_{public} newName) {
     if (new Passwd<α>_{α:}().check(guess,password))
         return new MedicalRecord<α>_{public}(this.newName,
                                              this.history,
                                              this.aesKey,
                                              this.password);
     else
         return this;
  }
}
```

Figure 4. Figure 1, rewritten in FJifP

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.

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 $\{\mathsf{Public}:\}$. 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 need the `check` method in Password to allow data to flow from Alice to Public.

There is one other technical detail to note in `updateName`. In order to simplify the semantics of FJifP, we omit including a special security label that keeps track of the current security level of `this`. Therefore, the only legal instances of the `MedicalRecord` class are ones where the two branches of the if statement return an object of the same type, and so `this` must always have the type $\mathtt{MedicalRecord}\langle\alpha\rangle_{pub}$. The inclusion of a security level for `this` would complicate the theory and the challenges this poses are orthogonal to studying trusted declassification.

3.2 Definitions

A FJifP program consists of a series of defined classes $\mathtt{C}, \mathtt{D}, \ldots$ and terms $\mathtt{t}_1, \mathtt{t}_2, \ldots$ 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 a few possibilities). Classes contain fields `f` and methods `m`. Instantiated classes

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	$\mathtt{CL} ::=$	$\mathtt{class}\ \mathtt{C}\langle\overline{\alpha}\rangle \triangleleft \mathtt{N}\ \{\ \overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\ \}$
Methods	$\mathtt{M} ::=$	$\mathtt{S}\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\ \mathtt{return}(\mathtt{t});\ \}$
Terms	$\mathtt{t} ::=$	$\mathtt{x}$
		$\mid \mathtt{t.f}$
		$\mid \mathtt{t.m}(\overline{\mathtt{t}})$
		$\mid \mathtt{new}\ \mathtt{S}(\mathtt{t})$
		$\mid (\mathtt{S})\ \mathtt{t}$
		$\mid \mathtt{actsfor}(p, q)\ \mathtt{in}\ \mathtt{t}$
Values	$\mathtt{u}, \mathtt{v} ::=$	$\mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}})$
Actsfor Hierarchy	$(p, q) \in \Delta$	
Declass. Policy	$(\mathtt{m}, p, q) \in \Upsilon$	
Security Contexts	$\Theta = (\Delta, \Upsilon)$	

Figure 5. FJifP Language Syntax

are parameterized by principals p and tagged by labels l for security. The language syntax for FJifP is given in Figure 5. As in FJ, the notation $\overline{\mathtt{x}}$ represents a list: so $\overline{\mathtt{x}}$ is a list of variables, parameterized $\mathtt{x}_1, \mathtt{x}_2, \ldots$. The notation $t[\mathtt{v}/\mathtt{x}]$ represents a simultaneous substitution being performed: in this case the value $\mathtt{v}$ is substituted for the free variables $\mathtt{x}$ in the term $\mathtt{t}$.

FJifP classes and terms are typed under a global security context $\Theta = (\Delta, \Upsilon)$. The trust relations between principals are given in the acts-for hierarchy Δ. For example, if Alice trusts Bob to act for her, then we have the pair $(\mathsf{Alice}, \mathsf{Bob}) \in \Delta$. 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 $\mathtt{m}$. $\mathtt{m}$ then acts as an information flow from p's data to q[3]. Define the function $\mathsf{extract}(\Upsilon, m)$ as follows:

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

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

Our security labels follow the decentralized label model (DLM) [14], 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.

[3] It would be simple, but technically more elaborate, to specify a more fine-grained policy that only added these new assumptions while typing certain methods m inside certain classes $\mathtt{C}$.

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

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

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

Figure 6. Security Context Judgements

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 $\mathtt{N}$. 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 $\mathtt{S}$ raised to the security level $\mathsf{lab}(\mathtt{S}) \sqcup l$. The definitions for these are as follows:

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

As in FJ, there is a special class, $\mathtt{Object}$, which has no principal variables, no fields, and no methods. Every other class inherits from this one.

FJifP contains a class table CT which looks up a class's definition. We examine a class's definition:

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

$\mathtt{C}$ is then a class with principal parameters α (the bar indicates a list), which inherits from the class $\mathtt{D}\langle\overline{q}\rangle$ (some of the q_i might be in $\overline{\alpha}$). $\mathtt{C}$ has whatever fields are declared in its parent along with the fields $\overline{\mathtt{S}}\ \overline{\mathtt{f}}$. $\mathtt{C}$ also has the methods declared in $\mathtt{D}\langle\overline{q}\rangle$, along with those in $\overline{\mathtt{M}}$; these might override the implementation of its parent's methods.

We define a few simple functions for future reference, to avoid continual reference to the class table in our inference rules.

- $\mathsf{parent}(\mathtt{C}) = \mathtt{D}\langle\overline{q}\rangle$: 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 $\mathtt{m}$ are declared as follows: $\mathtt{S}_0\ \mathtt{m}(\overline{\mathtt{S}}\ \overline{\mathtt{x}})$. Then the method $\mathtt{m}$ takes arguments $\overline{\mathtt{x}}$ of security type $\overline{\mathtt{S}}$ and returns a value of the security type $\mathtt{S}_0$. We now give important auxiliary definitions for field lookup, method lookup, method type lookup, method overriding, and others. We first give these definitions for parameterized

Subtyping Rules

$$\frac{}{\Theta \vdash \mathtt{S} \mathrel{<:} \mathtt{S}} \quad (\text{S-RELF})$$

$$\frac{\Theta \vdash \mathtt{S} \mathrel{<:} \mathtt{S}' \quad \Theta \vdash \mathtt{S}' \mathrel{<:} \mathtt{T}}{\Theta \vdash \mathtt{S} \mathrel{<:} \mathtt{T}} \quad (\text{S-TRANS})$$

$$\frac{\Theta \vdash l_1 \sqsubseteq l_2 \quad \mathsf{parent}(\mathtt{C}) = \mathtt{D}\langle\overline{q}\rangle \quad \mathsf{pvars}(\mathtt{C}) = \overline{\alpha}}{\Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l_1\} \mathrel{<:} \mathtt{D}\langle\overline{q}[\overline{p}/\overline{\alpha}]\rangle\{l_2\}} \quad (\text{S-CLASS})$$

Figure 8. Subtyping Rules

classes, then later overload their definition for security types in our inference rules; they are defined in Figure 7 and closely follow the analogous functions from FJ.

3.3 Subtyping

In FJ, a class C is a subtype of another class D if either 1) D is C, 2) C inherits from D, or 3) 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 need to define exactly what it means for a security type $\mathtt{C}\langle\overline{p}\rangle\{l\}$ to be a subtype of $\mathtt{D}\langle\overline{q}\rangle\{l\}$. The combination of two observations forms our subtyping rules, given in Figure 8. If we have $\mathrm{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\alpha\rangle \lhd \mathtt{D}\langle\alpha\rangle\ \{\ \cdots\}$, then $\mathtt{C}\langle\mathsf{Alice}\rangle\{l\}$ is a subtype of $\mathtt{D}\langle\mathsf{Alice}\rangle\{l\}$ for all l. Following Jif, even when $\Theta \vdash \mathsf{Alice} \preceq \mathsf{Bob}$, we do not have $\mathtt{C}\langle\mathsf{Alice}\rangle\{l\}$ as a subtype of $\mathtt{C}\langle\mathsf{Bob}\rangle\{l\}$.

As we can always safely raise the security level of a class, $\mathtt{C}\langle\overline{p}\rangle\{l_1\}$ is a subtype of $\mathtt{C}\langle\overline{p}\rangle\{l_2\}$ if l_2 is at least as restrictive as l_1. Subtyping for security classes then needs to be done under a security context Θ.

3.4 Typing Rules

We are prepared to present our typing rules for terms. Let Γ be an environment mapping variables to security types. There are three important judgements here. The first is term typing, written $\Theta; \Gamma \vdash \mathtt{t} : \mathtt{S}$; under security context Θ and environment Γ, the term t has type S. The second and third involve checking that classes and methods are well-formed. The judgement $\Theta \vdash \mathtt{S}$ OK specifies that a security-tagged and parameterized class C is well-formed under security context Θ; we can view Θ as the constraints that need to be satisfied in order to use C. The judgement $\Theta \vdash \mathtt{m}$ OK IN S says that the method m is well-formed within a security-tagged and parameterized class S under a security context Θ. Inference rules for term typing, class checking, and method checking are given in Figure 9.

Unfortunately, we must individually check that a class is well-formed at each instantiation of a security type. For example, suppose C has an integer in field f and the class D has a method m that takes an integer at $\{\mathsf{Alice} :\}$ security level. If a method in C calls D.m(this.f), then this call is alternatively legal or illegal depending on the current security level that C has been instantiated to. This difficulty could be circumvented by adding a special "this" security level, bound locally within each class. We do not include such a feature for reasons mentioned above and thus willing to accept this checking behavior.

3.5 Evaluation

Evaluation in FJifP is done in a similar to FJ, with one exception. To evaluate some terms, we need security information. The evaluation judgement is thus $\Theta \vdash \mathtt{t} \mapsto \mathtt{t}'$; under security context Θ, t makes a single step to $\mathtt{t}'$. When we talk of a complete evaluation from a term to a value, we write $\Theta \vdash \mathtt{t} \mapsto^* \mathtt{v}$, representing multiple evaluation steps. The evaluation rules for FJifP are given in Figure 10.

Note that there are two method invocation rules, (EV-INVKNEW) and (EV-INVKNEW-DEC). If $\Theta \vdash \mathtt{t} \mapsto^* \mathtt{v}$ without using the (EV-INVKNEW-DEC) rule, then noninterference still holds and an observer cannot gain any additional information from the term's evaluation. Otherwise, it is possible that some data has been leaked.

3.6 Type System Properties

With the following lemmas, we prove that FJifP is sound. The proofs are provided in the full version of this paper [5].

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. Induction on the typing derivation. □

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

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

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* $\Theta \vdash \mathtt{S}_0' \mathrel{<:} \mathtt{S}_0$*. Then* $\Theta; \Gamma \vdash \mathtt{t}[\mathtt{v}/\mathtt{x}] : \mathtt{S}'$ *for some* $\mathtt{S}'$ *such that* $\Theta \vdash \mathtt{S}' \mathrel{<:} \mathtt{S}$*.*

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

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

Theorem 3.5 (FJifP Type Preservation)**.** *If* $\Theta; \Gamma \vdash \mathtt{t} : \mathtt{S}$ *and* $\Theta \vdash \mathtt{t} \mapsto \mathtt{t}'$*, then there exists* Θ' *such that* $\Theta'; \Gamma \vdash \mathtt{t}' : \mathtt{S}'$ *for some* $\Theta \vdash \mathtt{S}' \mathrel{<:} \mathtt{S}$*.*

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

3.7 Noninterference

In order to show that the desired security policies hold, we define a bisimulation relation $\approx$ on FJifP terms. The judgement is written $\Theta \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}$: "under security context Θ, the terms $\mathtt{t}_1$ and $\mathtt{t}_2$ are observationally equivalent at security type S to an observer at security label ζ".

Essentially, two values are equivalent at security type S to the security label ζ if any equivalent operation that is performed on those values looks the same. If S is at a security level above the observer, then, assuming both values are typable to subtypes of S, any two values "look" the same. Otherwise, any action that the values can take, notably field access and method invocation, must also "look" the same. Two terms are equivalent if they both eventually evaluate to equivalent values. For our definition of noninterference, we do not address termination leaks: one term might finish evaluation while the other diverges.

The above reasoning is formalized in Definition 3.6.

Definition 3.6 (Observational Equivalence)**.** *Under security context* Θ*, two terms* $\mathtt{t}_1, \mathtt{t}_2$ *are observationally equivalent at security type S at security label* ζ*, written* $\Theta \vdash \mathtt{t}_1 \approx_\zeta \mathtt{t}_2 : \mathtt{S}$*, if:*

- $\Theta \vdash \mathtt{t}_1 : \mathtt{S}_1$ *and* $\Theta \vdash \mathtt{t}_2 : \mathtt{S}_2$ *and both* $\Theta \vdash \mathtt{S}_1 \mathrel{<:} \mathtt{S}$ *and* $\Theta \vdash \mathtt{S}_2 \mathrel{<:} \mathtt{S}$*.*
- *Suppose* $\mathtt{t}_1 \equiv \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}})$*,* $\mathtt{t}_2 \equiv \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}})$*. Then:*
  1. *If* $\Theta \vdash \mathsf{lab}(\mathtt{S}) \sqsubseteq \zeta$*, then for all* $\mathtt{T}_i\ \mathtt{f}_i \in \mathsf{fields}(\mathtt{S})$*,* $\Theta \vdash \mathtt{v}_i \approx_\zeta \mathtt{w}_i : \mathtt{T}_i$*.*

Field Lookup

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

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

Method Typing

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

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

Method Body Lookup

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

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

Declared Methods

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

Method Overriding

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

Overloaded Functions for Security Types

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

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

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

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

Figure 7. Auxiliary Definitions

Typing

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

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

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

$$\frac{\begin{array}{c}\mathsf{fields}(\mathtt{S}_0) = \overline{\mathtt{S}}\,\overline{\mathtt{f}} \quad \Theta; \Gamma \vdash \overline{\mathtt{t}} : \overline{\mathtt{S}'} \quad \Theta \vdash \overline{\mathtt{S}'} <: \overline{\mathtt{S}} \\ \Theta \vdash \mathtt{S}_0\ \text{OK}\end{array}}{\Theta; \Gamma \vdash \mathtt{new}\ \mathtt{S}_0(\overline{\mathtt{t}}) : \mathtt{S}_0} \quad (\text{TP-NEW})$$

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

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

Class Checking

$$\frac{\text{for all}\ \mathtt{m} \in \mathsf{methods}(\mathtt{C}\langle\overline{p}\rangle\{l\}),\ \Theta \vdash \mathtt{m}\ \text{OK IN}\ \mathtt{C}\langle\overline{p}\rangle\{l\}}{\Theta \vdash \mathtt{C}\langle\overline{p}\rangle\{l\}\ \text{OK}}$$

Method Checking

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

Figure 9. Typing Rules

$$\frac{\mathsf{fields}(\mathtt{S}) = \overline{\mathtt{S}}\,\overline{\mathtt{f}}}{\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}).\mathtt{f}_i \mapsto \mathtt{v}_i} \quad (\text{EV-PROKJNEW})$$

$$\frac{\mathsf{mbody}(\mathtt{m}, \mathtt{S}) = (\overline{\mathtt{x}}, \mathtt{t}_0) \quad \mathsf{extract}(\Upsilon, \mathtt{m}) = \emptyset}{\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}}) \mapsto \mathtt{t}_0[\overline{\mathtt{u}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}})/\mathtt{this}]} \quad (\text{EV-INVKNEW})$$

$$\frac{\mathsf{mbody}(\mathtt{m}, \mathtt{S}) = (\overline{\mathtt{x}}, \mathtt{t}_0) \quad \mathsf{extract}(\Upsilon, \mathtt{m}) \neq \emptyset}{\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}}) \mapsto \mathtt{t}_0[\overline{\mathtt{u}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}})/\mathtt{this}]} \quad (\text{EV-INVKNEW-DEC})$$

$$\frac{\Theta \vdash \mathtt{S} <: \mathtt{T}}{\Theta \vdash (\mathtt{t})\ \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}) \mapsto \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}})} \quad (\text{EV-CASTNEW})$$

$$\frac{\Theta \vdash p \preceq q}{\Theta \vdash \mathtt{actsfor}(p, q)\ \mathtt{in}\ \mathtt{t} \mapsto \mathtt{t}} \quad (\text{EV-ACTSFOR})$$

$$\frac{\Theta \vdash \mathtt{t}_0 \mapsto \mathtt{t}_0'}{\Theta \vdash \mathtt{t}_0.\mathtt{f} \mapsto \mathtt{t}_0'.\mathtt{f}} \quad (\text{EV-FIELD})$$

$$\frac{\Theta \vdash \mathtt{t}_0 \mapsto \mathtt{t}_0'}{\Theta \vdash \mathtt{t}_0.\mathtt{m}(\overline{\mathtt{t}}) \mapsto \mathtt{t}_0'.\mathtt{m}(\overline{\mathtt{t}})} \quad (\text{EV-INVK-RECV})$$

$$\frac{\Theta \vdash \mathtt{t}_i \mapsto \mathtt{t}_i'}{\Theta \vdash \mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{t}_i, \overline{\mathtt{t}}) \mapsto \mathtt{v}_0.\mathtt{m}(\overline{\mathtt{v}}, \mathtt{t}_i', \overline{\mathtt{t}})} \quad (\text{EV-INVK-ARG})$$

$$\frac{\Theta \vdash \mathtt{t}_i \mapsto \mathtt{t}_i'}{\Theta \vdash \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}, \mathtt{t}_i, \overline{\mathtt{t}}) \mapsto \mathtt{new}\ \mathtt{S}(\overline{\mathtt{v}}, \mathtt{t}_i', \overline{\mathtt{t}})} \quad (\text{EV-NEW-ARG})$$

$$\frac{\Theta \vdash \mathtt{t}_0 \mapsto \mathtt{t}_0'}{\Theta \vdash (\mathtt{t})\ \mathtt{t}_0 \mapsto (\mathtt{t})\ \mathtt{t}_0'} \quad (\text{EV-CAST})$$

Figure 10. Evaluation Rules

2. *If* $\Theta \vdash \mathsf{lab}(\mathtt{S}) \sqsubseteq \zeta$*, then for all* $\mathtt{m} \in \mathsf{methods}(\mathtt{S})$ *with* $\mathsf{mtype}(\mathtt{m}, \mathtt{S}) = \overline{\mathtt{T}} \to \mathtt{T}$ *and for all* $\overline{\mathtt{u}}, \overline{\mathtt{u}'}$ *such that* $\Theta \vdash \overline{\mathtt{u}} \approx_\zeta \overline{\mathtt{u}'} : \overline{\mathtt{t}}$*, then* $\Theta \vdash \mathtt{new}\ \mathtt{S}_1(\overline{\mathtt{v}}).\mathtt{m}(\overline{\mathtt{u}}) \approx_\zeta \mathtt{new}\ \mathtt{S}_2(\overline{\mathtt{w}}).\mathtt{m}(\overline{\mathtt{u}'}) : \mathtt{T}$.

- *Otherwise, for all* $\mathtt{v}_1$ *and* $\mathtt{v}_2$ *such that without using the evaluation rule* (EV-INVKNEW-DEC)*, both* $\Theta \vdash \mathtt{t}_1 \mapsto^* \mathtt{v}_1$ *and* $\Theta \vdash \mathtt{t}_2 \mapsto^* \mathtt{v}_2$ *then* $\Theta \vdash \mathtt{v}_1 \approx_\zeta \mathtt{v}_2 : \mathtt{S}$.

Value equivalence only makes sense without declassifier methods. Classes that use declassifiers usually cannot be shown to be observationally equivalent to one another, as it would require typing the bodies of their methods under a reduced security context. This is intuitively what we want: if a class has a method that can be used as a declassifier, it may not be noninterfering. On the other hand, by constructing the system in this way, we can be certain that the *only* points of noninterference are the points allowed explicitly in the security context. Thus, all information leaked is goverened by the declassification policy.

We now state the main security theorem. Suppose we have a program that is well-typed that relies on a free variable x. If we substitute in two observationally equivalent values to the term, then the evaluations of the program are also observationally equivalent. This captures the essence of noninterference: if we make a change in the program the observer cannot determine, then he cannot distinguish between the results of the two different evaluations.

Theorem 3.7 (Security)**.** *Suppose* $\Theta; \mathtt{x} : \mathtt{S}_0 \vdash \mathtt{t} : \mathtt{S}$ *and* $\Theta \vdash \mathtt{v}_1 \approx^p_\zeta \mathtt{v}_2 : \mathtt{S}_0$*. Then* $\Theta \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \approx^p_\zeta \mathtt{t}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}$.

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

Note that if the term t uses any declassifiers, then $\Theta \vdash \mathtt{t}[\mathtt{v}_1/\mathtt{x}] \approx^p_\zeta \mathtt{t}[\mathtt{v}_2/\mathtt{x}] : \mathtt{S}$ holds vacuously, since $\mathtt{t}[\mathtt{v}_1/\mathtt{x}]$ and $\mathtt{t}[\mathtt{v}_2/\mathtt{x}]$ cannot finish evaluation without using the evaluation rule (EV-INVKNEW-DEC). Suppose a term t finishes evaluation under a security context Θ; then any informational leakage that occured must have been done through declassification methods. The sections of the program which do not involve declassification are subject to the above theorem and so they remain observationally equivalent as they evaluate to values. Those values are then used in the larger program by methods that involve declassification; after information has been safely released, observational equivalence no longer holds. In short, all information leakage can be justified by the declassification policy, Υ.

4. Implementation

We implemented our trusted declassifiers in Jif 2.0 [15]. 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 written in this language into Jif code. 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). This is illustrated in Figure 11.

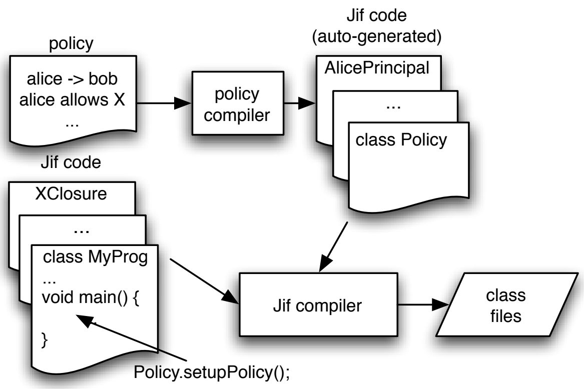

Figure 11. Integrating an external policy into Jif.

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* ::=	$(Del \mid Allow)*$

Figure 12. Policy language syntax.

Our policy language currently consists of only two kinds of statements, `->`-rules corresponding to delegations and `allow`-rules, establishing trust in declassifiers. The syntax is shown in Figure 12. 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 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` 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 building `Closure` subclasses for each declassifier, we can be sure that all declassifications will consult the policy before executing.

Consider the example policy in Figure 2. Compiling this policy generates classes for `AlicePrincipal`, `DrBobPrincipal`, `DrJohnPrincipal` and `ChuckPrincipal`, as well the `Policy`

```
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(AES[{P:}].encrypt(
                                  key,plaintext),{this});
   }
}
```

Figure 13. A closure for declassifying the cipher text generated by triple DES encryption. The standard constructor is defined, but not displayed.

class with a `setupPolicy` method that instantiates each class and performs the indicated delegations. The principals are automatically generated such that the `isAuthorized` method give authorization to the `Closures` 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.

Adding a declassifier One of the selling points 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 13. In order to use this closure to encrypt and declassify some plaintext, the principal who owns the plaintext must authorize `AESClosure`. 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 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, 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 which 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 $p = \text{Alice}$ then for all triples $(\mathtt{m}, \text{Alice}, q)$, the `isAuthorized` method for the `AlicePrincipal` class explicitly allows closures `m` with return type, q. In our example, this would be `AES.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. Although we have not built our restrictions into the Jif compiler, it should be straight-forward.
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.

We believe 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. In particular, 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 [20], 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.

4.3 A significant example

Since a key motivation for our approach is the hope of gaining practical experience with security-typed programming, we have used trusted declassification to implement a significant application, an e-mail client we call JPmail[4] (JP = Jif/policy). JPmail consists of an SMTP-compliant mail sender and a POP3-compliant mail reader. In our prototype implementation, we use a policy with dozens of principals, including a few groups (implemented as principals which delegate to the members of the group). It uses several declassifiers, including a variety of symmetric and asymmetric encryption declassifiers for sending sensitive data to an insecure mail server, 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, groups may be changed by merely changing a few lines in the policy file.

We offer an anecdote here from our experience to support the effectiveness of our model. When adding the policy to JPmail, we forgot that we had introduced a temporary work-around. When encrypting the body of an e-mail, we use skip encryption. We encrypt the body of the e-mail with a symmetric encryption method and then include the symmetric key in the body after encrypting it with the public key of the principal. Prior to integrating assymetric cryp-

[4] This application is still in development, but a preliminary version of the code can be found at `http://fatherboniface.org/jpmail`.

tography, so that we could encrypt the key with the principal's public key, we had introduced a hack to simply declassify shared keys before sending them (without first encrypting the key). We placed this declassification in a closure, as required by our model. When later developing our policy, we did not think to allow it in the policy file, because we clearly did not want to permit a declassifier which declassified shared keys. Consequently, our program, obedient to the policy, refused to send any keys over e-mail! This led us to track down the deprecated closure which was correctly maintaining the security we had established in our policy file. Lifting our policy to a global viewpoint was beneficial to understanding the security enforced in our application.

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 a great benefit to be able to understand all possible flows, including the fact that no symmetric keys were allowed to be declassified for any principal, by merely examining the policy file.

5. Related Work

This work falls into a long line of research on using security-typed languages to enforce information flow control and noninterference. This research development is detailed in a survey by Sabelfeld and Myers [18]. Various languages have been extended with security types for statically validating noninterference, but only one other system exists using an object calculus [2]. FJifP differs from the object calculus of Banerjee and Naumann in several ways. FJifP does not include notions of state or permissions. On the other hand, it is closer to Jif, because security types are built directly into the language as opposed to being an inferred annotation. The differences illustrate the difference in our motives for designing the language. Namely, we are primarily concerned with showing noninterference in a simple object-oriented language with declassification. Myers, et al. describe rules for the decentralized label model as implemented in Jif [14], but do not prove security properties about their system. To our knowledge, we provide the first proofs of noninterference and its relation to declassification in an object-based language.

Our work is most closely connected with Jif's selective declassification [12] which is the only declassification mechanism currently implemented in a full-scale language. We restrict this mechanism in order to lift out authorization into an external, global policy. In this way, we are able to prove the security property of *noninterference modulo trusted methods*.

Much work has recently been done on declassification, as described in a recent survey [19]. In this survey, the authors loosely divide declassification schemata into four categories: who, what, where and when. Our model does not fit nicely into any of these categories. It corresponds mostly to "where" declassification may occur (in explicitly identified declassifiers). "Who", "when" and "what" is declassified may be gleaned from analyzing the policy and the declassifiers themselves. Our system could naturally be strengthened by quantifying exactly what may be leaked by declassifiers. For example, our system facilitates knowing that Alice's data can only be leaked by a password check by merely examining the external policy. Analyzing this declassifier, it could be determined that only one bit of information is leaked per call. A further analysis could ensure that it is not called a sufficient number of times to leak more than a certain amount of information.

Broberg and Sands recently introduced the notion of flow locks [3] for describing temporal policies. This work is similar to ours in that spots of declassification are limited to explicitly identified regions. We can imagine placing appropriate flow locks around our declassifiers. To handle an acts-for hierarchy, flow locks must be places around the whole program. While this mechanism is very general, it is also very localized. Our policies are more global and more flexible. For example, we can more readily ensure, given only our global policy, that Alice's data satisfies noninterference.

Chong and Myers introduce a mechanism for downgrading until conditions [4]. This model allows downgrading only in the presence of externally verified conditions. It is similar to ours in that we both check an externally verified condition. They open new flows, which are not subsequently closed, while our mechanism limits declassification to the bodies of declassifiers. Furthermore, they provide some possibilities for conditions, but they provide no external policy or actual implementation.

Ana Matos and Boudol's non-disclosure policies [11, 10] are also related to our approach. They have locally induced, transitive policies. Their system makes an important contribution in handling concurrency, but they do not have an implementation; we accepted the limitations of Jif (no concurrency) in order to facilitate an implementation. They also do not allow declassifications to be expressed as a global policy.

Another well-studied declassification mechanism related to ours is robust declassification [16] which is currently in the process of being implemented in Jif. The key to this mechanism is in the use of integrity. It uses integrity to ensure that low integrity flows do not influence high confidentiality data that will later be declassified. Integrity is not currently implemented for Jif, although it is actively being developed. Once robust declassification is implemented in Jif, it will work well with our model, making declassifications safe from active attackers. It is orthogonal to the issues we discuss in allowing principals to choose declassifiers which they trust and implementing this in an external, global policy. It will also lead naturally to additional policy statements which may place robustness constraints on the input parameters for declassifiers.

Tse and Zdancewic implement a decentralized, certificate-based mechanism for declassification [21]. They describe their system in a lambda-calculus with subtyping and modals in order to make the addition of features more modular. Like us, they use subtyping to describe declassifications. They prove a noninterference theorem which says that so long as no declassifications are visible to the observer, noninterference is maintained. Furthermore, they are able to justify all declassifications based on externally validated certificates. They provide a prototype implementation for a functional language, but it is not as robust or practical as Jif. Ideally, a merging of our approach and theirs could yield a very useful architecture for building distributed applications. As research on dynamic updates to the security policy is more developed and our model is relaxed to allow later delegations and declassifications, certificates could be a good way to encode our policy statements and allow runtime updates to the policy.

6. Conclusion and Future Work

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 in Jif 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 restrictions we place on Jif are that all delegations must appear at the beginning of a program, that declassify statements can only be placed in Jif's `Closure`'s, and that only our automatically generated principals may be used in programs. We demonstrated the practicality of our approach by using it in a prototype Email client and we found it easy to use in practice.

Previously, determining the security of a Jif application would require combing over all the code to find all the `declassify` statements, as done by Askarov et al. in their analysis of mental poker [1]. This is already a great improvement over other ad hoc security certification techniques, because it narrows down the escape hatches to a small number of `declassify` statements. Our approach takes this a step further, however. For our system, the security analysis only requires inpspection of the policy and the code of the declassifiers.

One area of future work is in relaxing some of the restrictions we have imposed on Jif. For example, delegations and allowances for declassification could appear later in the program, even being established at runtime, and runtime checks could be used to consult a principal's policy. This would actually be a more natural implementation of our mechanism in Jif, but some security theorems should be proved for this. This is nontrivial, since it opens the door for dynamic updating of policies, which is still an open area of research [6, 20].

Another avenue of future work lies in expanding the policy model. It is currently very simple, but could be more expressive. For example, constraints could be added to indicate negative information flows. Policy analyses could also be used to determine whether separation of duties is maintained between two principals. When integrity is added to Jif, it could be expanded with robustness constraints.

We plan to continue using our declassification mechanism to gain practical experience. It is a general problem in language-based security that there is too little experience with security-typed programming to help guide such research as designing the best form of declassification. We hope that our implementation of this mechanism in Jif will help to promote more practical experience with declassifiers which will better inform future research.

References

[1] A. Askarov and A. Sabelfeld. 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, Milan, Italy, September 2005. Springer-Verlag.

[2] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005. Special issue on Language-based Security.

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

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

[5] B. Hicks, D. King, P. McDaniel, and M. Hicks. Trusted declassification: High-level policy for a security-typed language. Technical Report NAS-TR-0033-2006, Networking and Security Research Center, Department of Computer Science, Pennsylvania State University, March 2006.

[6] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proceedings of the Foundations of Computer Security Workshop (FCS '05)*, March 2005.

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

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

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

[10] A. A. Matos and G. Boudol. On declassification and the non-disclosure policy. Submitted for possible publication, October 2005.

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

[12] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '99)*, pages 228–241, January 1999.

[13] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, pages 186–197, May 1998.

[14] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

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

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

[17] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '02)*, pages 319–330, January 2002.

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

[19] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop*, Aix-en-Provence, France, June 2005.

[20] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic. Managing policy updates in security-typed languages, February 2006. Submitted for publication.

[21] S. Tse and S. Zdancewic. A design for a security-typed language with certificate-based declassification. In *Proc. of the 10th European Symposium on Programming*, Lecture Notes in Computer Science, 2005.