Declassification with Cryptographic Functions in a Security-Typed Language

Boniface Hicks, David King and Patrick McDaniel
Systems and Internet Infrastructure Security Laboratory (SIIS)
Computer Science and Engineering, Pennsylvania State University
Technical Report NAS-0004-2004
{phicks,dhking,mcdaniel}@cse.psu.edu

Abstract

Security-typed languages are powerful tools for provably enforcing noninterference. Real computing systems, however, often intentionally violate noninterference by deliberately releasing (or declassifying) sensitive information. These systems frequently trust cryptographic functions to achieve declassification while still maintaining confidentiality. We introduce the notion of trusted functions that implicitly act as declassifiers within a security-typed language. Proofs of the new language's soundness and its enforcement of a weakened form of noninterference are given. Additionally, we implement trusted functions used for declassification in the Jif language. This represents a step forward in making security-typed languages more practical for use in real systems.

1 Introduction

Data confidentiality is a principal element of secure systems design. However, comprehensive solutions remain surprisingly elusive: massive data compromises have become commonplace, and the resulting costs are in the billions of dollars. Regulatory bodies are responding to this threat, but currently provide only punitive, negative incentives [oHotS03]; they do not require provable security guarantees. Moreover, the effectiveness of the often ad hoc certification processes currently performed on critical systems is questionable [KSRW04]. This is not surprising as the sources of information leakage can be quite subtle (consider the slow leakage of data made possible by timing attacks [BB03]). Simply put, it is infeasible to understand all information flows present in any non-trivial application without automated assistance.

A promising approach to solving the confidentiality problem is to build systems that enforce information flow security. In such systems, data is tagged with a *security level*. The system then enforces the property of *noninterference*: data tagged with a high security level may not flow to low security channels. Mandatory access control systems enforce noninterference dynamically by mediating sensitive operations (e.g., system calls) via an external monitor [Fen73, Fen74]. Monitor-based systems have historically been hampered by performance overhead and their limited ability to track implicit information flows [DD77]. More recently, researchers have sought to use programming language techniques to enforce noninterference statically. In such systems, a type checker is augmented to track information about data security levels, treating illegal information flows as a type error.

Unfortunately, noninterference is too restrictive for many common applications. Consider a password checker that compares a low-security guess with a high security password and releases a single bit about the high security password (i.e., whether the guess is correct). This violates pure nointerference. Likewise, when high-security plaintext is encrypted and sent over a low-security channel, some information, despite how minuscule, is leaked. To handle these and other legitimate, intentional releases of high security data, practical security-typed languages must include a mechanism for *declassification*.

The problem is that existing declassification mechanisms do not distinguish between leaking only small

amounts of information (as in the above examples) and leaking unlimited amounts of information. To address this problem, previous approaches [CM04, LZ05] have sought to provide a powerful, general theoretical framework for characterizing the ways that declassification may release information. These frameworks have not been implemented. Our work approaches the problem from a new angle. We make no attempt to provide such a general theoretical framework; we believe that declassification is most often found in a few, specific forms. In particular, recent evidence suggests that the majority of cases in which declassification is needed involve the use of cryptographic functions [AS05]. Furthermore, by focusing only on this specific area of declassification, we are able to provide not only proofs of correctness and security, but also an implementation in a security-typed language.

In this paper, we extend security-typed languages with *trusted declassification functions*. These functions can declassify data for principals that trust them. This language device is used to formalize the relationship between one-way functions and noninterference. Trusted functions become implicit declassification mechanisms, obviating the need for explicit programmer declassification in these cases. We make the following contributions in this paper:

- The simple security-typed language $\lambda^{\mathcal{F}}_{sec}$ is formulated and illustrated. $\lambda^{\mathcal{F}}_{sec}$ supports a notion of *trusted declassification*, allowing trusted functions to declassify private data.
- A proof of *noninterference modulo trusted functions* for $\lambda^{\mathcal{F}}_{sec}$ is given.
- We implement the language extensions in the Jif compiler [MNZZ01], a security-typed variant of Java.

The rest of the paper is structured as follows. We begin in the following section by illustrating type security, declassification, and our approach via an example. Section 3 gives high-level descriptions of the important theoretical constructs later defined formally in Section 4. Also included in Section 3 is a careful description of the properties of cryptography we utilize, as well as our assumptions about the adversary we model. Section 5 gives theorems and proofs for the important properties of our language, namely soundness and noninterference modulo trusted functions. In Section 6, we describe our extension to the Jif language with a constraint which marks functions as being one-way and trusted for declassification. We discuss general issues associated with our technique in Section 7. Section 8 presents related work. We conclude in Section 9.

2 A Motivating Example

A security-type system extends standard types with security-label annotations. In the following example, we give code that could be run by Alice to send a secret message to Bob over a public channel. Here, a two-point lattice of security labels is used with `secret` being at a higher security level than `public`[1] (written `public` $\sqsubseteq$ `secret`).

```
void send(String{public} address,
          String{public} message);
...
String{secret} msg = "Attack at dawn.";
send("Bob",msg);
```

In this code, `msg` is tagged as secret data by labeling its type with `{secret}`. The prototype given here for the `send` function indicates that `send` requires its inputs to be public. This prototype is fitting, because `send` is meant to be a function which sends a message over a public channel.

If executed as is, this code would violate noninterference, because Alice is attempting to send secret data on a public channel. Consequently, in a security-typed language, the type-checker will flag this information leak, because `msg`'s security level is not as low as the corresponding formal parameter to `send` (`secret` $\not\sqsubseteq$ `public`). Thus this program is disallowed by the compiler.

Intuitively, we should be able to fix this code by using RSA to encrypt the message with Bob's public key. Consider the following:

[1] In Jif, public is denoted as `{}`. Mathematically, public is denoted $\bot$, while the most secret values are denoted as $\top$. Here we use "public" and "secret" for ease of reading.

```
void send(String{public} address,
          String{public} message);
...
String{secret} msg = "Attack at dawn.";
Key{public} bKey = PKI.getPubKey("Bob");
send("Bob",RSAencrypt(msg,bKey));
```

In fact, this may not fix the problem; it still depends on the security annotations in the prototype for `RSAencrypt`. We would like this prototype to be

```
String{public}
RSAencrypt(String{secret} plaintext,
           Key{public} pubKey);
```

This prototype asserts that `RSAencrypt` produces a `public` output given a `secret` input. The only way a function could be annotated in this way is if it either 1) did not involve its secret input `plaintext` in the computation of the output in any way, or 2) if it had a declassification hidden in the function body. The former cannot be true here because `plaintext` is obviously used to produce the output.

This is because, strictly speaking, RSA encryption *does* reveal information about its input. Suppose that `msg` were declared as:

```
String{Alice} msg = "Lie in wait.";
```

In this case, a different ciphertext will be sent to Bob and an observer will be able to tell the difference between the original code and the modified code, even though they cannot determine what either message means. This violates pure noninterference: a low-security observer sees two different low-level outputs for runs on two different high-level inputs.

This contradicts a basic security assumption—in practical systems with computationally bounded adversaries, one-way functions reveal no information about their inputs. To capitalize on this property, we introduce a new option: `RSAencrypt` can be assigned a principal, `RSA`, and marked as a trusted declassification function. This gives us the following secure code:

```
String{public}
RSAencrypt(String{secret} plaintext,
           Key{public} pubKey)
   where declassFor(RSA,secret);

void send(String{public} address,
          String{public} message);
...
String{secret} msg = "Attack at dawn.";
Key{public} bKey = PKI.getPubKey("Bob");
send("Bob",RSAencrypt(msg,bKey));
```

In this code fragment, a new constraint is introduced which associates a security level `RSA` with `RSAencrypt`. If `secret` trusts `RSA` (`secret` $\sqsubseteq$ `RSA`), then this function can be used to declassify `secret` data. The remainder of the paper discusses our analysis and implementation of this approach.

3 Introduction to $\lambda^{\mathcal{F}}_{sec}$

We highlight the common features in security-typed languages and introduce $\lambda^{\mathcal{F}}_{sec}$, a simple language with functions that can be used for declassification. First, we give a brief description of the adversary and the security model.

3.1 Cryptography and Trusted Declassification

It is important to consider which functions are appropriate to trust as declassification mechanisms. We argue only *one-way functions* can be safely used for declassification. A function f is one-way if for essentially all inputs x, computing $f(x)$ given x is easy, but finding some x given $f(x)$ is computationally infeasible [MVO96]. It is precisely this feature that makes many forms of cryptography useful. For example, in using encryption, it is assumed that the adversary can learn little about the plaintext from the ciphertext alone. Our work seeks to exploit this property within a security-typed programming language.

This work assumes a computationally bounded adversary. Because such an adversary cannot recover secret inputs from the outputs of a one-way function, these functions may be trusted to declassify data. Consequently, for ease of exposition, the following sections treat one-way functions as though they expose

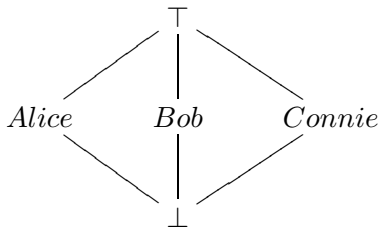

Figure 1: An example security lattice

no information, *which is clearly not the case*. The amount of information a cryptographic algorithm exposes is a feature of the algorithm itself and its security parameters. We briefly discuss the implications of this in Section 7.

3.2 The Security Lattice

In a *security-typed language*, values are assigned a security level. Security levels are taken from a *security lattice* $\mathcal{L}$: a static encoding of access control policy. If $l, l' \in \mathcal{L}$, we write $\ell \sqsubseteq \ell'$ to indicate that ℓ' is at least as secure as ℓ.

There is a "write-up, read-down" relationship on data in the security lattice. Anyone at level ℓ can read data that is stored at a level $\ell' \sqsubseteq \ell$. Similarly, any data stored at level ℓ can always be made more secure to a level ℓ', where $\ell \sqsubseteq \ell'$. This is the $\star$-property [LB73].

An example security lattice is given in Figure 1. The highest security level is $\top$; none of Alice, Bob, or Connie can view data stored at $\top$ security. The lowest security level in the lattice is $\bot$; anyone can read $\bot$ data. As there is no $\sqsubseteq$ relationship between $Alice$ and Bob, Alice cannot read Bob's data nor vice versa.

A security lattice $\mathcal{L}$ is a *join semilattice*: for any two levels $\ell, \ell' \in \mathcal{L}$, there is a security level $\ell \sqcup \ell' \in \mathcal{L}$, where $\ell \sqcup \ell'$ is the least upper bound of ℓ and ℓ'. The least upper bound of two security levels is the lowest security level that is at least as secure as both ℓ and ℓ'. Any two security levels have a lower bound ($\bot$), but they do not necessarily have a greatest lower bound. The $\sqsubseteq$ relation is transitive: if ℓ is at least as secure as ℓ' ($\ell' \sqsubseteq \ell$) and ℓ'' is at least as secure as ℓ' ($\ell'' \sqsubseteq \ell'$), then ℓ is at least as secure at ℓ'' ($\ell'' \sqsubseteq \ell$).

3.3 The Language $\lambda_{sec}^{\mathcal{F}}$

λ_{sec} is a simple security-typed language; it has expressions for function abstraction, application, conditionals, and primitives operating on integer and boolean values. It is a purely functional language: lambda functions are first-class values and there is no notion of program state.

$\lambda_{sec}^{\mathcal{F}}$ is λ_{sec} extended with *transformation functions*. Transformation functions are a generalization of primitives such as addition or multiplication: they transform one value into another. Each transformation function F is associated with a security level ℓ_F, the upper bound on the data that F is able to declassify. To our knowledge, this is the first time that named functions are associated with a level in the security policy lattice.

Transformation functions can model important operations such as addition, equality testing, public-key encryption and decryption, hashing, and so on. It have would be simple to add a new encryption type constructor instead; a value of type $t\ enc$ would then be an encrypted value of type t. However, this restricts the use of encrypted values. For example, an arithmetic function on integers has type $\mathsf{int} \times \mathsf{int} \rightarrow \mathsf{int}$: it could not also operate on encrypted values. It is often necessary to perform operations on encrypted values, such as sending them over public channels, writing them out to files, or exploiting homomorphic properties of the encryption. We therefore choose a more general approach.

Some of these functions can be used for declassification of high-security data, e.g. encryption. The addition function reveals too much information about its inputs to be a safe declassifier, while the result of a public-key decryption should be kept secret. Equality testing is a transformational function that can be either trusted or not, depending on whether a user wishes to allow potentially dangerous information release [Vol00].

Let ℓ_F be the security level associated with F. If Alice trusts a transformational function F, the relation $Alice \sqsubseteq \ell_F$ holds in $\mathcal{L}$. Figure 2 shows an example security policy, extending the simple lattice given earlier. Alice allows her data to be declassified after it passes through either an RSA or a DES encryption function. Bob insists that his data can only be declassified after public key operations, while Connie

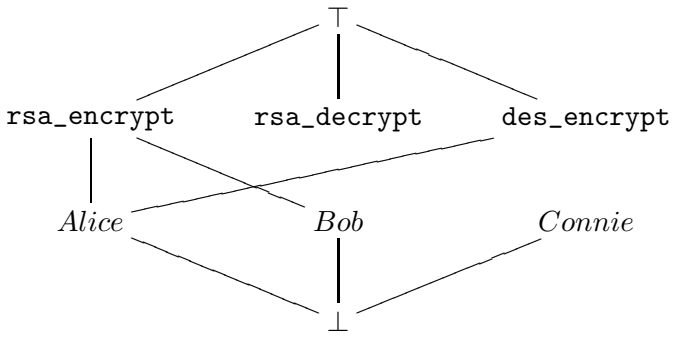

Figure 2: An example security lattice with trusted and untrusted functions

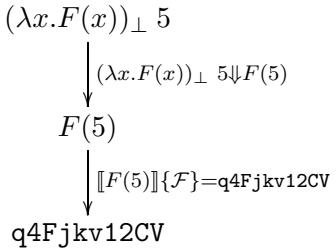

Figure 3: An example two-step evaluation in $\lambda^{\mathcal{F}}_{sec}$

does not allow her information to be declassified at all. Wisely, none of the principals allow their data to be declassified after being passed through a decryption function.[2]

The above notion of trust becomes even more useful when dealing with applications such as password checking. Traditional noninterference does not allow information release with an equality test. In the language $\lambda^{\mathcal{F}}_{sec}$, users are able to specify with fine granularity which methods of information release are allowed.

3.4 Evaluation in $\lambda^{\mathcal{F}}_{sec}$

Let e be the expression `rsa_encrypt`(x), where x is a variable owned by a principal that trusts RSA encryption to declassify her data. After substituting a value for x, the end result of computing e is some ciphertext. Even though two different ciphertexts look random to an observer, different values yield different resulting ciphertexts; this violates pure noninterference.

For this reason, evaluating an expression in $\lambda^{\mathcal{F}}_{sec}$ is composed of two distinct steps. The *operational semantics* provide a method of reducing the portions of the program which do not rely on transformation functions, while the *reduction semantics* evaluate transformation functions. Our noninterference result then applies to the language $\lambda^{\mathcal{F}}_{sec}$ after the operational semantics are applied but before the reduction semantics are invoked.

Figure 3 provides an example evaluation. Suppose F is an encryption function outputting the ciphertext `q4Fjkv12CV` for the input value 5. During the execution of the program $P \equiv (\lambda x.F(x))_\perp\ 5$, the operational semantics first apply to the parts of the program that do not rely on transformation functions: in this case, $(\lambda x.F(x))_\perp\ 5 \Downarrow F(5)$. After everything else has been evaluated, the reduction semantics are applied, filling in the semantic meaning of the transformation functions. The reduction semantics then yield $[\![F(5)]\!]\{\mathcal{F}\} =$ `q4Fjkv12CV`, and so the execution of the program P ultimately results in the string `q4Fjkv12CV`.

The operational semantics of $\lambda^{\mathcal{F}}_{sec}$ first evaluates an expression to an intermediate state where values are either basic values (integers, booleans), lambda functions, or the result of the application of a transformation function. Transformation functions as they occur in $\lambda^{\mathcal{F}}_{sec}$ are designated by *transformation function symbols*: the symbol F, used in the operational semantics, corresponds to the semantic function $\mathcal{F}$, used in the reduction semantics.

4 Formal Definition of $\lambda^{\mathcal{F}}_{sec}$

The syntax of $\lambda^{\mathcal{F}}_{sec}$ is given in Figure 4. $\lambda^{\mathcal{F}}_{sec}$ is not a strict extension of λ_{sec}: we remove the primitive operation $\oplus$ from the language and introduce in its place an expression for applying transformation functions. $\lambda^{\mathcal{F}}_{sec}$ also introduces product expressions, $\langle e_1, e_2 \rangle$, and the usual expressions for dealing with them, **fst**(e) and **snd**(e). Binary primitives used in λ_{sec} are then added to $\lambda^{\mathcal{F}}_{sec}$ as transformation functions. For example, the

[2]Though we can have transformation functions for both encryption and decryption in $\lambda^{\mathcal{F}}_{sec}$, our focus in this work is on functions trusted to serve as declassification mechanisms. Declassifying the result of decrypted ciphertext would reveal the original message; for our purposes then, decryption is not an interesting operation.

Security Levels	$\ell \in \mathcal{L}$	
Security Types	$s ::=$	t_ℓ
Base Types	$\iota ::=$	$\mathsf{int} \mid \mathsf{bool}$
Types	$t ::=$	$\iota \mid s_1 \times s_2 \mid s_1 \rightarrow s_2$
Base Values	$b ::=$	$0 \mid 1 \mid \cdots \mid \mathsf{true} \mid \mathsf{false}$
Values	$v ::=$	$b_\ell \mid x \mid (\lambda x.e)_\ell \mid$ $\langle v_1, v_2 \rangle \mid F(v)$
Expressions	$e ::=$	$v \mid e_1\ e_2 \mid \langle e_1, e_2 \rangle \mid$ $\mathbf{fst}(e) \mid \mathbf{snd}(e) \mid F(e)$

Figure 4: Syntax for $\lambda^{\mathcal{F}}_{sec}$

λ_{sec} expression $2 + 3$ corresponds to the $\lambda^{\mathcal{F}}_{sec}$ expression $+(\langle 2, 3 \rangle)$, where $+$ is a transformation function symbol representing the semantic addition function on integers.

Transformation function symbols are denoted by F. The type of a transformation function symbol is given by $type(F) = ((t')_{\ell'} \rightarrow (\iota)_{\ell'})_{\ell_F}$. The type ℓ_F is the security level of the transformation function symbol F, and is the upper bound on information that F is trusted to declassify. Transformation functions always take values at one security level ℓ' to another value at the same security level (but not necessarily the same type). Declassification then results from the context that the transformation function is used in and is not reflected by the function's type. A transformation function can then be used in two different locations, serving as a declassification method in the first and not in the second; this is more general than will be later required by our implementation.

We assume that the return type of the transformation function is a base type such as int or bool. We do this to ensure that we can always reduce the result of a transformation function. If we allow expressions such as $\mathbf{fst}(F(e))$, then values in our language no longer can be identified solely by their syntactic form. We restrict the return type to be a base type; this limitation is reasonable as transformation functions are meant to be a generalization of the notion of a primitive.

The function $lvl(s)$ is defined on labeled types as $lvl(t_\ell) = \ell$. The lattice operation $\sqsubseteq$ is also extended to security types: $s \sqsubseteq \ell$ if $lvl(s) \sqsubseteq \ell$. Write s as t_ℓ, then the security type $s \sqcup \ell'$ is defined to be $t_{\ell \sqcup \ell'}$, while the security type $s \sqcap \bot$ is $t_\bot$. Intuitively, $s \sqcup \ell'$ is the security type s raised to be at the least security level above both $lvl(s)$ and ℓ while $s \sqcap \bot$ is the security type s fully exposed to the public.

The reduction semantics are given in Figure 5. The transformation function symbol F is given meaning by the semantic transformation function $\mathcal{F}$. Each semantic transformation function $\mathcal{F}$ is a partial function on the set of values; if $type(F) = (t'_{\ell'} \rightarrow t_{\ell'})_{\ell_F}$, then $\mathcal{F}$ is a function from values of type t' to values of type t. $\mathcal{O}$ is a set of semantic transformation functions $\mathcal{F}$. The reduction rule $[\![v']\!]\mathcal{O} = v$ gives meaning to transformation function symbols in the value v'.

The typing rules for $\lambda^{\mathcal{F}}_{sec}$ are the same as the typing rules for λ_{sec} for their common syntactic constructions (if statements, function abstraction and applications, and so on). Contexts Γ are maps from variables to security types: in an expression e, the context Γ gives meaning to free variables in e. The typing judgement then is $\Gamma \vdash e : s$, read as "under context Γ, expression e has security type s." In Section 5 we will see that this implies that e evaluates under the operational semantics to a value v and $\vdash v : s$.

The two new important typing rules are (TP-TRANS1) and (TP-TRANS2). (TP-TRANS1) states that if a transformation function is not trusted to declassify data at the $lvl(s_1)$ security level, then its application can only raise the security level of the result. On the other hand, if it is trusted to declassify data at that level, (TP-TRANS2) allows the result to be declassified to the $\bot$.[3]

In both cases, the type of the argument need only be a subtype of the argument that the transformation function expects. This adds a form of label polymorphism to these transformation functions. For example, there may be one encryption function F with $type(F) = (\mathsf{int}_\top \rightarrow \mathsf{int}_\top)_{\ell_F}$. Let v be a value with $\vdash v : \mathsf{int}_{Alice}$. If Alice trusts F for declassification, then $F(v)$ can be given the security type $\mathsf{int}_\bot$; otherwise $F(v)$ has the security type $\mathsf{int}_\top$. F can also be applied to a value of type int_{Bob} or an int_{Connie} with the same declassification behavior.

The operational rule (EV-TRANS) evaluates the inside of the application of a transformation function.

[3] It may be useful for some functions to only declassify data to a certain security level for such applications as secret splitting. We do not consider this possibility, though we believe such an extension would not be difficult.

$$\frac{}{[\![b]\!]\mathcal{O} = b}\ (\text{RED-BVAL})$$

$$\frac{}{[\![(\lambda x.e)_\ell]\!]\mathcal{O} = (\lambda x.e)_\ell}\ (\text{RED-LAM})$$

$$\frac{[\![v_1]\!]\mathcal{O} = v_1' \quad [\![v_2]\!]\mathcal{O} = v_2'}{[\![\langle v_1, v_2\rangle]\!]\mathcal{O} = \langle v_1', v_2'\rangle}\ (\text{RED-PROD})$$

$$\frac{\mathcal{F} \in \mathcal{O} \quad [\![v]\!]\mathcal{O} = v_0 \quad \mathcal{F}(v_0) = v'}{[\![F(v)]\!]\mathcal{O} = v'}\ (\text{RED-ORAC})$$

Figure 5: Reduction Semantics

The important typing rules and operational semantics are given in Figure 6.

The security lattice induces subtyping judgements on security types and normal types: $s \preceq s'$ and $t \preceq t'$. Write s as t_ℓ and s' as $t'_{\ell'}$: if $s \preceq s'$, then $t \preceq t'$ and $\ell \sqsubseteq \ell'$. As usual, function types are contravariant and product types are covariant [Pie02]. If $s \preceq s'$ and the type derivation $\Gamma \vdash e : s$ holds, then by the (TP-SUB) rule we can construct the derivation $\Gamma \vdash e : s'$.

Extending the theory of $\lambda^{\mathcal{F}}_{sec}$ to include state, an operator for unbounded recursion, concurrency, and message-passing would not be difficult; security-typed languages that incorporate these already exist. However, some language features add new methods of exposing information. By focusing on a small core language, we are able to concentrate on the important issues raised by placing transformation functions in the security lattice.

The typing rules, operational semantics, and subtyping judgement for the full language $\lambda^{\mathcal{F}}_{sec}$ are given in Appendix A.

5 Properties of $\lambda^{\mathcal{F}}_{sec}$

We now demonstrate that a number of standard language properties hold in $\lambda^{\mathcal{F}}_{sec}$. We first show that the language is *sound*, i.e. a typing judgement implies safe evaluation. We then show that *noninterference modulo trusted functions* holds.

5.1 Properties of the Observer

The formal security semantics presume that an observer is at security level ζ. An expression e is then *noninterfering modulo trusted functions* if any two inputs to e which are indistinguishable below ζ result in computations that are also indistinguishable below ζ. Our observers occupy a position in the security lattice. For example, an omniscient adversary might occupy the level $\top$ in the lattice and thus be able observe all changes in data, no matter at what security level. Any program using RSA encryption would then violate noninterference to such an adversary.

$$\frac{\Gamma \vdash e : s_1 \quad type(F) = (s' \to s)_{\ell_F} \quad s_1 \sqsubseteq s' \quad s_1 \not\sqsubseteq \ell_F}{\Gamma \vdash F(e) : s \sqcup lvl(s_1)}\ (\text{TP-TRANS1})$$

$$\frac{\Gamma \vdash e : s_1 \quad type(F) = (s' \to s)_{\ell_F} \quad s_1 \sqsubseteq s' \quad s_1 \sqsubseteq \ell_F}{\Gamma \vdash F(e) : s \sqcap \bot}\ (\text{TP-TRANS2})$$

$$\frac{\Gamma \vdash e : s \quad s \preceq s'}{\Gamma \vdash e : s'}\ (\text{TP-SUB})$$

$$\frac{e \Downarrow v}{F(e) \Downarrow F(v)}\ (\text{EV-TRANS})$$

Figure 6: Important Typing Rules and Operational Semantics

As we consider our adversary to be computationally bounded, they occupy a security level ζ not above the levels of one-way functions such as RSA or DES encryption. Our adversary may be above the security levels of transformation functions such as addition, multiplication, equality testing. Noninterference modulo trusted functions holds for a program as long as none of the transformation functions used for declassification is below the observer's security level ζ. Note that if no transformation functions are used for declassification, then pure noninterference holds.

5.2 Language Properties

Our language $\lambda^{\mathcal{F}}_{sec}$ has a few important properties. For example, typing rules are invariant under variable substitutions and expressions that are given a security type evaluate to values of the same type. These standard results show that the typing rules correctly correspond with the operational semantics.

A substitution γ is a finite map from variables to values. A substitution is said to satisfy a type envi-

ronment ($\gamma \models \Gamma$) if γ is a type-preserving substitution, meaning that γ assigns each variable a value of the security type required by Γ. The notation $\gamma(e)$ is short-hand for a simultaneous, capture-avoiding substitution.

Definition 5.1. *For γ a substitution and Γ a type environment, $\gamma \models \Gamma$ if $\mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma)$ and for all $x \in \mathrm{dom}(\Gamma), \vdash \gamma(x) : \Gamma(x)$.*

The following results are easily proved by induction.

Lemma 5.2 (Value Substitution). *If $\Gamma \vdash e : s$ and $\gamma \models \Gamma$ then $\vdash \gamma(e) : s$.*

Proof. By induction on the shape of the typing derivation. □

The following theorem states that if an expression is typable and its computation results in a value, then the expression and the value have the same security type.

Theorem 5.3 (Type Preservation). *If $\vdash e : s$ and $e \Downarrow v$, then $\vdash v : s$.*

Proof. By induction on the shape of the typing derivation, using Lemma 5.2 for the application case. □

Theorem 5.4 states that "well-typed programs do not go wrong". If we type an expression to a value, then it will not get stuck during its evaluation at a type error. As $\lambda^{\mathcal{F}}_{sec}$ has no expression that would allow unbounded computation, if an expression is typed under the empty context, it will evaluate to a value v.

Theorem 5.4 (Operational Soundness). *If $\vdash e : s$, then there exists a v such that $e \Downarrow v$.*

Proof. Again, by induction on the shape of the typing derivation. Expressions always terminate as the language has no unbounded computation, and expressions never get stuck since transformation functions always return a value that has a basic type. □

As an immediate corollary, the reduction semantics interact with the operational semantics to give a full soundness result. If e is properly typed and the transformation functions are well-behaved, then it has a complete two-step evaluation as discussed above.

Corollary 5.5 (Full Soundness). *If $\vdash e : s$ and for every transformation function symbol F occurring in e, $\mathcal{F} \in \mathcal{O}$ and $\mathcal{F}$ is well-defined on the correct subset of values as specified by $type(F)$, then there exists a v' and a v such that $e \Downarrow v'$ and $[\![v']\!]\mathcal{O} = v$.*

5.3 Security Properties

In this section, we give the formal definition of what it means for a program to be noninterfering. We then show that if a program has been given a security-type judgement, then all evaluations of it are indistinguishable modulo the result of these transformation functions.

We define what it means for two values to be observationally equivalent to an attacker. This is given by the judgement $v_1 \approx_\zeta v_2 : s$, read "$v_1$ is observationally equivalent to v_2 at security type s to an observer at security level ζ." We extend the definition of $\approx_\zeta$ from λ_{sec} for values of the form $F(v)$ (the full definition of $\approx_\zeta$ for all values can be found in Appendix A). The intuition is that if an observer at level ζ can read both of $F(v_1)$ and $F(v_2)$ and $\ell_F \sqsubseteq \zeta$, then the observer can see v_1 and v_2. If F is a one-way function and ζ is a computationally bounded observer, this corresponds to the idea that no information is released by F.

Definition 5.6 (Value Observational Equivalence). *$F(v_1) \approx_\zeta F(v_2) : t_\ell$ if the following hold:*

- *$\vdash F(v_1) : t_\ell$ and $\vdash F(v_2) : t_\ell$*
- *$type(F) = ((t')_{\ell'} \to (t)_{\ell'})_{\ell_F}$*
- *$\ell \sqsubseteq \zeta$ and $\ell_F \sqsubseteq \zeta$ implies $v_1 \approx_\zeta v_2 : (t')_{\ell_F \sqcup \ell}$*

Subtyping interacts with observational equivalence in the expected way. If two values are equivalent at a security type, then they are also equivalent at higher security types.

Lemma 5.7 (Subtyping). *If $v_1 \approx_\zeta v_2 : s$ and $s \preceq s'$, then $v_1 \approx_\zeta v_2 : s'$.*

Proof. Proof by induction on the structure of the value. Suppose $F(v_1) \approx_\zeta F(v_2) : s$. We show $F(v_1) \approx_\zeta F(v_2) : s'$.

Since $F(v_1) \approx_\zeta F(v_2) : s$, we have the judgements

$$\vdash F(v_1) : s \quad \text{and} \quad \vdash F(v_2) : s$$

By the rule (TP-SUB), we then have the typings

$$\vdash F(v_1) : s' \quad \text{and} \quad \vdash F(v_2) : s'$$

Since $s \preceq s'$, $lvl(s) \sqsubseteq lvl(s')$. If $lvl(s) \not\sqsubseteq \zeta$, then $lvl(s') \not\sqsubseteq \zeta$ and we are finished. Otherwise assume $\ell_F \sqsubseteq \zeta$. Then

$$v_1 \approx_\zeta v_2 : (t')_{\ell_F \sqcup lvl(s)}$$

We then have

$$(t')_{\ell_F \sqcup lvl(s)} \preceq (t')_{\ell_F \sqcup lvl(s')}$$

The required result then follows by induction. □

The $\approx_\zeta$ relation is extended for expressions. Two expressions are observationally equivalent if they evaluate to observationally equivalent values.

Definition 5.8 (Expression Observational Equivalence). *Two expressions e_1, e_2 are $e_1 \approx_\zeta e_2 : s$ if*

- $\vdash e_1 : s$ *and* $\vdash e_2 : s$
- $e_1 \Downarrow v_1$ *and* $e_2 \Downarrow v_2$
- $v_1 \approx_\zeta v_2 : s$.

Two substitutions are "related" above a security type ζ in a context Γ if both γ_1 and γ_2 define the variables in the domain of Γ and their definitions look the same to an observer sitting at level ζ. Formally:

Definition 5.9 (Related Substitutions). *Two substitutions γ_1 and γ_2 are related above a security level ζ in a context Γ, written $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$, if:*

- $\gamma_1 \models \Gamma$ *and* $\gamma_2 \models \Gamma$
- *for all* $x \in \mathrm{dom}(\Gamma)$, $\gamma_1(x) \approx_\zeta \gamma_2(x) : \Gamma(x)$

Two related substitutions $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ then are the result of switching data at a level above ζ. An attacker should not be able to tell the difference between applying γ_1 or γ_2 to a "secure" expression e.

Theorem 5.10 (Noninterfering Substitutions). *If $\Gamma \vdash e : s$ and $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$, then $\gamma_1(e) \approx_\zeta \gamma_2(e) : s$.*

Proof. Proof proceeds by induction on the derivation of $\Gamma \vdash e : s$. The two key cases are when the judgement has been made by either of the transformation function rules.

$$\frac{\begin{array}{c}\Gamma \vdash e : s_1 \\ type(F) = (s' \to s)_{\ell_F} \\ s_1 \sqsubseteq s' \quad s_1 \not\sqsubseteq \ell_F\end{array}}{\Gamma \vdash F(e) : s \sqcup lvl(s_1)} \ (\text{TP-TRANS1})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : s_1 \\ type(F) = (s' \to s)_{\ell_F} \\ s_1 \sqsubseteq s' \quad s_1 \sqsubseteq \ell_F\end{array}}{\Gamma \vdash F(e) : s \sqcap \bot} \ (\text{TP-TRANS2})$$

In either case, apply induction to the derivation $\Gamma \vdash e : s_1$ to receive

$$\gamma_1(e) \approx_\zeta \gamma_2(e) : s_1$$

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

$$\gamma_1(e) \Downarrow v_1 \quad \gamma_2(e) \Downarrow v_2$$

and the equivalence

$$v_1 \approx_\zeta v_2 : s_1$$

There are now three important security labels to keep track of.

- ℓ_1 is the security level of the initial data, before the transformation has been applied. Here $\ell_1 = lvl(s_1)$.
- ℓ_F is the security level of the function.
- ℓ is the security level of the final data.

Write $s_1 \equiv (t_1)_{\ell_1}$ and $s' \equiv (t')_{\ell'}$; since $s_1 \sqsubseteq s'$, $t_1 \sqsubseteq t'$ and $\ell_1 \sqsubseteq \ell'$. With this rewriting, we have

$$v_1 \approx_\zeta v_2 : (t_1)_{\ell_1}$$

For noninterference to hold, we need to show

$$F(v_1) \approx_\zeta F(v_2) : t_\ell$$

For the desired result to hold, we must show that $\ell \sqsubseteq \zeta$ and $\ell_F \sqsubseteq \zeta$ implies $v_1 \approx_\zeta v_2 : (t')_{\ell \sqcup \ell_F}$. We must then show $\ell_1 \sqsubseteq \ell \sqcup \ell_F$: then as $t_1 \sqsubseteq t'$, Lemma 5.7 gives required result.

If the rule (TP-TRANS1) holds, we have $\ell = lvl(s \sqcup lvl(s_1)) = lvl(s \sqcup \ell_1)$ and $\ell_1 \sqsubseteq lvl(s \sqcup \ell_1) \sqcup \ell_F$; thus the result follows.

If instead the rule (TP-TRANS2) holds, we know $\ell_1 \sqsubseteq \ell_F$, and thus $\ell_1 \sqsubseteq \ell \sqcup \ell_F$ and again the result follows.

We then have

$$F(v_1) \approx_\zeta F(v_2) : t_\ell$$

and so

$$\gamma_1(F(e)) \approx_\zeta \gamma_2(F(e)) : t_\ell$$

This is the required result. □

The preceding theorem implies that noninterference modulo trusted functions holds in $\lambda^{\mathcal{F}}_{sec}$. As $\lambda^{\mathcal{F}}_{sec}$ is a functional language, inputs to a program are given by substitutions assigning values to free variables. If the inputs for an expression e are indistinguishable above ζ, this result guarantees that the computations resulting from the inputs are also indistinguishable above ζ.

6 Implementation

A central motivation of this work was to make security-typed languages more practical. We have implemented our approach as an extension to Jif. Our extension consists of an annotation on functions which associates each function with a principal and allows the function to declassify data for all other principals that trust that principal.

We chose Jif because it is a full-scale security-typed language implementation based on Java. The only other existing full-featured security-typed language is Flow Caml, an extension of the functional language Objective Caml. Some features of Jif, such as the extensible Polyglot framework with which it is built, made it a more attractive target for our implementation. Furthermore, since Jif already has a general mechanism for declassification, we could leverage this to provide our more specialized mechanism. Finally, Jif's ability to use the Java class libraries allowed easier integration of cryptography.

6.1 The Jif programming language

Jif is an object-oriented, strongly-typed language capturing nearly[4] a superset of Java. In Jif, the programmer must label types with security annotations. The compiler uses these annotations during type-checking to ensure noninterference.

The Decentralized Label Model (DLM) Types in Jif are annotated with security labels based on the DLM [ML00]. Similar to work in mandatory access control that tags data with complete access control lists, the DLM allows for the virtual tagging of data with owners-readers lists. Each label consists of a set of policies of the form `{o:`$\mathtt{r}_1$`,`$\mathtt{r}_2$`,...,`$\mathtt{r}_n$`}`, where `o` and $\mathtt{r}_i$ are principals with `o` being the owner of the policy and the $\mathtt{r}_i$ being authorized readers of the policy. Furthermore, a label can consist of multiple policies (allowing for multiple owners of a piece of data). As an example, `int{Alice:} i;` declares an `int` owned and readable only by `Alice` (the owner is always implicitly included in the reader list). The statement `String{Bob:Charlie,Dana} str;` declares a `String` which is owned by `Bob` but also readable by `Charlie` and `Dana`. Data may also be annotated with multiple policies as in `int{Alice:;Bob:} j;`. The policy on `j` indicates that it is owned and readable by both `Alice` and `Bob`. In Jif, when a variable is used in a security label, it refers to its own label. Thus, using `i` and `str` as defined above, `float{i;str} f;` declares a `float` that is owned by both `Alice` and `Bob` and which can be read by `Charlie` and `Dana`.

Selective declassification Jif implements *selective declassification*. Principals in Jif are defined external to the program. Each one has a delegation set containing all the principals it trusts. This forms a runtime principal hierarchy. Each process maintains an authority set which contains principals from the runtime principal hierarchy. A process is only authorized to declassify policies that are owned by principals in its authority set. Our language extension takes advantage of this as described below.

[4] Jif does not provide support for inner classes or threads, because of the ways they complicate information flow analysis. Jif is described most completely by Myers [Mye99b] and a helpful, practical overview, along with expository examples, is given by Askarov and Sabelfeld [AS05].

Class parameterization Another feature in Jif which we utilize is class parameterization. A Jif class can be parameterized by a principal or security label. This means that a class may be defined once and then be instantiated at various security levels. For example, we might want a `Vector` class which contains `secret` data and another `Vector` class that contains `public` data. Without having to implement the `Vector` class multiple times, it could be parameterized with a label and then instantiated at different levels. In Jif, such a class could be defined as

```
public class Vector[principal P]
{
   Object{P:}[]{P:} elements;
}
```

Note that the member array `elements` has two labels. One is the label of the `Objects` stored in the array. The other is the label of the array itself.

Since `Vector` has been parameterized by `P`, `P` can now be used throughout the body of the class to denote a principal. This principal will be instantiated when an object of type `Vector` is declared, as in the following code, where `Alice` and `Bob` are two principals.

```
Vector[Alice] vector1;
Vector[Bob] vector2;
```

Handling exceptions One thing which makes Jif particularly challenging for programming is handling the information leaks that occur through function termination, exceptions and side-effects. For example, an encryption method that throws an `InvalidKeyException` releases information about the key (which is secret data) both by throwing the exception (indicating the key is invalid) and by not throwing the exception, i.e. by terminating normally (indicating the key is not invalid). For this reason, it can be advantageous to catch exceptions and handle them locally in order to bound information leakage they might cause. This is the approach we have taken for our encryption function, allowing simpler integration of encryption for programmers using this class.

6.2 A language extension for cryptographic declassification

The key insight in our formal system is in assigning a principal to each function and allowing cryptographic functions to be trusted for declassification by placing them higher in the security lattice. In our implementation, we only need to mark those functions which will be trusted for declassification. All other functions are presumed to be unsafe for declassification and thus implicitly labeled as $\bot$. Thus, our implementation serves to extend Jif with an additional optional constraint which marks functions as being trusted for declassification. This constraint assigns a principal to a function and places it in the security lattice by relating it to some other existing principal. Corresponding to the formal system, the constraint also requires the compiler to make certain security label checks. Finally, the constraint relaxes noninterference, allowing the return value of the function (which should be encrypted data) to be public.

It was already possible to use cryptography in Jif prior to our extension. Because Jif builds on Java, Jif programs can use the existing Java class libraries. The only stipulation is that the programmer must provide security-annotated prototypes (called *Jif signatures*) for each class, interface and method that has been used. Methods may be marked as `native` to signal Jif that Java code will be provided elsewhere for the body of this method. The signatures for these Java methods must be written in a way that truly reflects the information leaked by the method, because the signature cannot be automatically checked by Jif. Ideally, classes would be re-implemented in Jif to take advantage of Jif's automatic type-checking.

The primary Java class for cryptography is the `javax.crypto.Cipher` class provided in the Java Cryptography Extension (JCE). To create a new DES cipher in CBC mode, the `Cipher.getInstance` method is called with the argument `"DES/CBC/PKCS5Padding"`. The cipher's `init` method must then be called to configure it for either encryption or decryption. Finally, an encryption or decryption may be performed using the method `doFinal`. The proper Jif signature for this method is given in Figure 7.

This signature indicates that `doFinal` takes a `byte` array which is at some security level, designated as

```
public native byte{this;input}[]{this;input} doFinal(byte{input}[] input)
        throws IllegalBlockSizeException, BadPaddingException;
```

Figure 7: Jif signature for the `Cipher.doFinal` method.

`input`. Furthermore, the elements of the array are also at the security level `input`. The label `this` refers to the label of the `Cipher` instance whose `doFinal` method has been called. Thus, the encrypted output array is labeled `{this;input}` because it contains data that reveals information about both the `Cipher` instance used to encrypt it and also the `input` array which was encrypted.

In addition to producing ciphertext, an encryption in CBC mode also generates an initialization vector which is needed for subsequent decryption. Our Jif `Ciphertext` class merely serves to package these two outputs into a data structure as a single output from the encryption function.

The syntax of the function constraint we have introduced is `declassFor`$(Principal, Principal)$. The constraint `declassFor(CryptoPrin,DataPrin)` indicates that the `CryptoPrin` is trusted to declassify data owned by `DataPrin`. An example is given in Figure 8. Although the method `encrypt` takes as input a key and data owned by `Alice` (i.e., annotated with the label `{Alice:}`), the `Ciphertext` output is completely public (given as `{}`). This corresponds with the accepted notion that an encryption function can safely publicize the ciphertext of secret data. It indicates that `Alice` trusts this function to declassify her data because she believes it will reveal no information about it.

The stipulations on the use of `declassFor` are the ones given in the formal semantics in Section 4. Namely, all inputs to the method must be owned by the principal `DataPrin` (or principals that trust `DataPrin`) and `DataPrin` must trust the principal `CryptoPrin`. We have extended the Jif compiler to check these conditions automatically and then to declassify the return value of the method. Since Jif's selective declassification requires a process to have sufficient authority to declassify, the `declassFor` constraint also extends the authority set of the calling process to include the `CryptoPrin` principal.

The encryption function given above is limited in that it can only be used for `Alice`. For real applications, we can take advantage of Jif's class parameterization and build a DES class which is parameterized based on the data-owning principal as shown in Figure 9.

This allows the `encrypt` method to be invoked for any desired principal as in the following code which encrypts and declassifies Bob's data using Bob's key.

```
Key{Bob:} key = DES[Bob].getKey();
String{Bob:} data;
Ciphertext{} ciphertext =
   DES[Bob].encrypt(key,data);
```

The full code for the DES class is given in Appendix B.

A full implementation of the language and associated documentation is available online at:

http://siis.cse.psu.edu/tools/infoflow/jif/jifcrypto.html

6.3 Implementation challenges

Jif is built using the Polyglot extensible compiler frontend framework for Java [NCM03]. This framework is powerful and robust, allowing arbitrary extensions and deletions to the syntax of Java. Abstract syntax tree (AST) extensions and deletions can be given along with arbitrary additional passes over these ASTs.

An important challenge in implementing our extension was in determining how to handle the two principals (`CryptoPrin` and `DataPrin`) involved in the `declassFor` constraint. This required manipulating the static and dynamic principal hierarchies in Jif (which correspond to the security lattice in our formal system). In addition to the runtime principal hierarchy described earlier, Jif also has a static principal hierarchy which can be formed within a program through inserting dynamic checks. Since the body of these dynamic checks will only be executed if the check succeeds at runtime, the body can be statically type-checked using the assumption that the delegation (or "acts-for") relationship holds. In this way, a static principal hierarchy is formed.

The following code demonstrates how our DES class would be used. In this example, a check

```
static public Ciphertext{} encrypt(Key{Alice:} key,
                                   String{Alice:} s)
       where declassFor(DESprin,Alice);
```

Figure 8: An encryption method for Alice's data.

```
public class DES [principal P] authority(DESprin)
{
  static public Ciphertext{} encrypt(Key{P:} key, String{P:} s)
         where declassFor(DESprin,P);
  ...
}
```

Figure 9: A parameterized encryption method.

will be made at runtime to ensure that `Alice` trusts `DESprin` before `Alice`'s data may be encrypted and declassified. If Alice does not trust `DESprin`, a `SecurityException` is thrown.

```
actsFor(DESprin,Alice) {
  Ciphertext{} ciphertext =
     DES[Alice].encrypt(key,data);
}
else throw new SecurityException();
```

The `declassFor` constraint ensures at compile time that such an `actsFor` check has been made and that the `CryptoPrin` is already established to be higher than the `DataPrin` in the static principal hierarchy. This has the advantage of forcing the programmer to be explicit about which cryptographic methods are being trusted for declassification in a given program.

An alternative we explored was to infer the `DataPrin` by examining the owners of the labels of the input arguments. Because of the limitations of dynamic labels in the current version of Jif, however, we were unable to pursue this further. Currently, dynamic labels can only be inspected by using a `switch label` statement. Furthermore, the label would have to be passed in explicitly as a dynamic label, not merely inferred from the existing arguments, as an artifact of how default labels are treated in Jif.

7 Discussion

7.1 A Practical Relaxation of Noninterference

As is true with any program using declassification, this work depends on a relaxation of noninterference. Our relaxation is minimal, however, as we only allow as much information as is leaked by trusted functions. With the notable exception of one-time pads, all one-way functions considered in this paper release some information about the inputs [Sha49]. The amount of information released is bounded by the algorithms' security parameters and is assumed to be vanishingly small. Ours is a general mechanism and hence not bound to any particular algorithm. The use of one-way functions makes the application output inherently subject to cryptanalysis. This vulnerability is orthogonal to the present work. The lesson is that due caution must be used in selecting and using cryptographic functions.

7.2 Language-Based Security Tools

The recent years have been marked by the emergence of tools for enforcing static guarantees of language-based information security. Programming language techniques have laid the theoretical foundations for the implementation of security-typed languages, but new languages such as Jif and FlowCaml [MNZZ01, Sim03] have made it possible for programmers to write secure code in a robust, full-featured environment. These tools and research such as ours serve to

further support *high-assurance* requirement. Security-typed languages provide a new approach to providing provable security guarantees for critical applications, e.g., medical devices, electrical grid control.

Two main challenges face the designers of these language tools. The first involves increasing language semantics with new features, such as run-time principals, revocation, and more coarse-grained policy controls. The second involves making these tools easier to deploy in business and government applications. Our work meets this second challenge by tapping into a well-understood programming mechanism and relating it to information flow control. Encryption and hashing are fundamental tools for building secure applications and are naturally used for publishing secure data in a way that maintains confidentiality. In the realm of information flow control, this is called declassification. Thus, with this language extension, we have tied together a standard security technique with a less commonly understood mechanism for information release in a way that is safe and intuitive.

7.3 Future Work

The language $\lambda^{\mathcal{F}}_{sec}$ admits a simple noninterference proof precisely because the semantics of a transformation function are not involved with our language semantics. Consider a symmetric key encryption function such as DES: it takes an integer as a key argument, a string argument to encrypt, then returns a string argument as the encrypted result. This function should have the type

$$((\mathsf{int}_\ell \times \mathsf{String}_\ell)_\ell \to \mathsf{String}_\ell)_{\ell_{\mathrm{DES}}}$$

If DES is able to declassify data at security level l, the result should be public. However, if the symmetric key was originally viewable to the public, an observer will be able to see changes in which value has been encrypted. It may be illuminating to investigate some notion of modeling values and their security levels as trap-doors into certain transformation functions.

Another direction is in continuing this approach to making simple, practical, safe mechanisms for declassification. Other work [CM04, LZ05, AS05, SS05] has identified common paradigms for safe declassification. Although encryption is the most common, others exist. One of particular importance are seals which protect data until something occurs and then releases data. An example is an auction where all bids are confidential until a certain time when the winning bid is released. Analogous to our work, there may be other specialized, practical safe mechanisms for declassification with which a security-typed language could be extended.

Such mechanisms can best be recognized by making the effort to build more applications with security-typed languages. This would help in considering the needs of systems designers when building real programs. Further case studies towards this end are needed [Zda04, AS05].

8 Related Work

The concept of information flow control is well established. After the first formulation for the Orange Book lattice (Top Secret, Secret, etc.) by Bell and La Padula [LB73] and the subsequent definition of noninterference [GM82], Smith, Volpano, and Irvine first recast the question of information flow into a static type judgement for a simple imperative language [VSI96]. The notion of information flow has been extended to languages with many other features, such as programs with multiple threads of execution [SV98], functional languages and their extensions [Zda02] and distributed systems [MS03]. Sabelfeld and Myers [SM03a] provide a comprehensive survey of the field.

Two robust security-typed languages have been implemented that statically enforce noninterference. JFlow [Mye99a] and its successor Jif [MNZZ01] introduce such features as a decentralized label model, label polymorphism, and run-time principals in an extension to the Java language. Flowcaml [Sim03] [PS02] implements a security-typed version of the Caml language, that satisfies noninterference.

The property of noninterference tends to be overly restrictive in practice. Most real programs require some notion of interference and intentionally leak some information to an observer. A simple password checking program reveals one bit of information to an observer: whether or not the guessed password was correct. An encrypted message leaks a ciphertext which is dependent on the input. This introduces *declassification*: the process of decreasing the security

annotation on data in some controlled manner. Declassification is a powerful mechanism, but it violates pure noninterference. It is therefore important to retain some security guarantees when using declassification. Sabelfeld and Sands recently provide a survey of this field [SS05].

Intransitive noninterference [RG99] is a relaxing of noninterference where information may flow between different security levels by a some policy. If information can flow from level ℓ to level ℓ', we write $\ell \rightsquigarrow \ell'$. The policy may be intransitive: $\ell \rightsquigarrow \ell_0$ and $\ell_0 \rightsquigarrow \ell'$ do not imply $\ell \rightsquigarrow \ell'$. We do not consider declassification with this granularity in this paper.

Myers and Liskov [ML00] formulated a *selective* form of declassification which requires that a process have the proper *authority* in order to declassify data. Each process is declared to have authority to act for a (possibly empty) set of principals. That process can legitimately declassify data owned by those principals. This is the notion of declassification that is built into Jif and thus also the one extended by our implementation.

Myers and Zdancewic proposed a *robustness* property [ZM01] for declassification, for which a type system [Zda03] and enforcement mechanism [MSZ04] were subsequently developed. This work enforces the security property that an attacker cannot exploit declassification to gain more information than is intentionally released. This is accomplished by tracking integrity and ensuring that declassification only takes place in code trusted by the owner of the declassified data.

Two other approaches have attempted to formalize notions of declassification so that specific forms can be expressed in concise terms. Chong and Myers [CM04] introduce the concept of *noninterference until conditions*. This property guarantees that noninterference is maintained until certain conditions occur (e.g. a seal is lifted, all bids have been cast in an auction, or password guesses are checked by certified code). Li and Zdancewic [LZ05] propose a mathematical model called *relaxed noninterference* which factors a program into high and low inputs and specifies exactly what high information is released by using composable, extensional functions. These two works are most relevant to our result. They seek to provide a general mathematical framework for characterizing the ways that declassification may release information. We make no such attempt to provide this; rather we focus on a very specific form of declassification—encryption. Since the goal of our work is making security-typed languages more practical rather than more powerful, we address the form of declassification which is most standard in secure programs. By limiting the scope of our work, we are able to provide not only a theoretical soundness result, but also an implementation in a security-typed language.

Also related but orthogonal are quantitative approaches [Lau03, Low02, DPHW02, Vol00, SM03b] and *relative secrecy* [VS00] which attempt to determine or bound how much information is released using probabilistic methods. We do not attempt to determine how much information is being released by these trusted cryptographic functions; we trust that they release sufficiently little so as to be impervious to cryptanalysis. This work could be complementary to ours by providing further guarantees and give formal reasons for trusting certain functions to declassify secret data.

Several approaches towards showing security through behavior equivalence have been done outside of a security-typed framework. Abadi and Gordon [AG98] consider the spi-calculus [AG99] with shared-key symmetric cryptography, and prove a bisimulation result. Sumii and Pierce [SP01] focus on extending the λ-calculus with cryptographic primitives. Both show the behavioral equivalence of observationally equivalent systems and assume only one implicitly trusted cryptographic function; they do not consider more general declassification within a security policy lattice.

9 Conclusion

In real systems, one-way functions are used to declassify sensitive data because they are trusted to release no information. Our work introduces this property into a security-typed language. We annotate functions with security levels. One-way functions may be annotated with a high security level; this indicates they are *trusted* and permits them to serve as a safe mechanism for declassification. In this paper, we have given a formal definition of a security-typed language with this extension, and proved that it enforces *noninterfer-*

ence modulo trusted functions. Additionally, we have implemented this extension in the Jif language. This extension represents another step forward in making provable noninterference policies accessible for practical applications.

Acknowledgements

A very special thanks to Michael Hicks for his repeated excellent and challenging reviews of the paper. Thanks also to Tal Malkin for his help in aiding our precision with the cryptography terminology. We would also like to extend thanks to Stephen Chong for his technical help with Jif and helpful reviewing of the paper. Finally, thanks to John Hannan, Stephen Tse, Aslan Askarov, Andrei Sabelfeld, William Enck and Patrick Traynor for their helpful reviews and comments.

References

[AG98] Martín Abadi and Andrew D. Gordon. A bisimulation method for cryptographic protocols. *Lecture Notes in Computer Science*, 1381:12–26, 1998.

[AG99] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[AS05] Aslan Askarov and Andrei Sabelfeld. Secure implementation of cryptographic protocols: A case study of mutual distrust. Technical Report 2005-13, Chalmers University of Technology and Göteborg University, April 2005. `http://www.cs.chalmers.se/~aaskarov/jifpoker/as05.pdf`.

[BB03] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proceedings of the 12th Usenix Security Symposium*, August 2003.

[CM04] Stephen Chong and Andrew C. Myers. Security policies for downgrading. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, Oct 2004.

[DD77] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[DPHW02] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Approximate noninterference. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 1–17. IEEE Copmuter Society Press, 2002.

[Fen73] J. S. Fenton. *Information Protection Systems*. PhD thesis, University of Cambridge, Cambridge, England, 1973.

[Fen74] J. S. Fenton. Memoryless subsystems. *Computing J.*, 17(2):143–147, May 1974.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[KSRW04] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–. IEEE Computer Society, 2004.

[Lau03] Peeter Laud. Handling encryption in analyses for secure information flow. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming, ESOP 2003*, Programming Languages and Systems, pages 159–173, Warsaw, Poland, April 2003.

[LB73] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

[Low02] Gavin Lowe. Quantifying information flow. In *Proc. of the IEEE Computer Security Foundations Workshop*, pages 18–31. IEEE Computer Society Press, 2002.

[LZ05] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the ACM Conference on Principles in Programming Languages (POPL05)*, January 2005. to appear.

[ML00] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.

[MNZZ01] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic. Jif: Java information flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[MS03] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *J. Computer Security*, 11(4):615–676, 2003.

[MSZ04] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 172–186, Pacific Grove, California, June 2004.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[Mye99a] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

[Mye99b] Andrew C. Myers. Mostly-static decentralized information flow control. Technical Report MIT/LCS/TR-783, Massachussetts Institute of Technology, University of Cambridge, January 1999. Ph.D. thesis.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. 12th International Conference on Compiler Construction*, number 2622 in Lecture Notes in Computer Science, pages 138–152, Warsaw, Poland, April 2003.

[oHotS03] Department of Health and Human Services Office of the Secretary. Health insurance reform: Security standards; final rule. Federal Register, February 2003. `http://www.cms.hhs.gov/hipaa/hipaa2/regulations/security/03-3877.pdf`. Consider statute 164.312(e)(1) especially.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[PS02] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

[RG99] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *CSFW*, pages 228–238, 1999.

[Sha49] C. E. Shannon. Communication theory of secrecy systems. *Bell Sys. Tech. J.*, 28(4):656–715, 1949.

[Sim03] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003. ©INRIA.

[SM03a] Andrei Sabelfeld and Andrew Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003. This is a survey article on

language-based techniques for the specification and enforcement of confidentiality properties.

[SM03b] Andrei Sabelfeld and Andrew Myers. A model for delimited information release. In *Proceedings of the 2003 International Symposium on Software Security (ISSS'03)*, Tokyo, Japan, November 2003.

[SP01] E. Sumii and B. Pierce. Logical relations for encryption. In *Proc. IEEE Computer Security Foundations Workshop*, pages 256–269, June 2001.

[SS05] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop*, Aix-en-Provence, France, June 2005.

[SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 355–364, January 1998.

[Vol00] Dennis M. Volpano. Secure introduction of one-way functions. In *CSFW*, pages 246–254, 2000.

[VS00] Dennis M. Volpano and Geoffrey Smith. Verifying secrets and relative secrecy. In *POPL*, pages 268–276, 2000.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. Computer Security*, 4(3):167–187, 1996.

[Zda02] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, July 2002.

[Zda03] Steve Zdancewic. A type system for robust declassification. In *Mathematical Foundations of Programming Semantics (MFPS XIX)*, Montreal, Canada, March 2003.

[Zda04] Steve Zdancewic. Challenges in information-flow security. In *Proceedings of the First International Workshop on Programming Language Interference and Dependence (PLID)*, Verona, Italy, August 2004.

[ZM01] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.

Appendix A

$$\frac{}{t \preceq t}\ (\text{ST-REFL})$$

$$\frac{t_1 \preceq t_2 \quad t_2 \preceq t_3}{t_1 \preceq t_3}\ (\text{ST-TRANS})$$

$$\frac{s_1' \preceq s_1 \quad s_2 \preceq s_2'}{s_1 \to s_2 \preceq s_1' \to s_2'}\ (\text{ST-FUN})$$

$$\frac{s_1 \preceq s_1' \quad s_2 \preceq s_2'}{s_1 \times s_2 \preceq s_1' \times s_2'}\ (\text{ST-PROD})$$

$$\frac{t \preceq t' \quad \ell \sqsubseteq \ell'}{t_\ell \preceq t'_{\ell'}}\ (\text{ST-SLAB})$$

Figure 12: Subtyping Rules

The full typing rules, subtyping judgement, and operational semantics, and subtyping are given in Figure 10, Figure 12, and Figure 11 respectively. The full definition of the $\approx_\zeta$ relation is given in Figure 13. The map $\pi : b \to \iota$ assigns basic values to their basic types, i.e. $\pi(3) = \mathsf{int}$, $\pi(\mathsf{true}) = \mathsf{bool}$.

Theorem A.1 (Substitution). *If* $\Gamma \vdash e : s$ *and* $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$, *then* $\gamma_1(e) \approx_\zeta \gamma_2(e) : s$.

Proof. Proof by induction on the size of the typing derivation.

Case (TP-PROD):

We have

$$\Gamma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)_\ell$$

where $\ell = lvl(s_1) \sqcup lvl(s_2)$. We have the derivations

$$\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_2 : s_2$$

Applying induction to these smaller derivations gives us

$$\gamma_1(e_1) \approx_\zeta \gamma_2(e_1) : s_1 \quad \gamma_1(e_2) \approx_\zeta \gamma_2(e_2) : s_2$$

and thus we have the evaluations

$$\gamma_1(e_1) \Downarrow v_1 \quad \gamma_2(e_1) \Downarrow v_2$$
$$\gamma_1(e_2) \Downarrow w_1 \quad \gamma_2(e_2) \Downarrow w_2$$

and the equivalences

$$v_1 \approx_\zeta v_2 : s_1 \quad w_1 \approx_\zeta w_2 : s_2$$

Suppose $\ell \sqsubseteq \zeta$: then $lvl(s_1) \sqsubseteq \zeta$ and $lvl(s_2) \sqsubseteq \zeta$, and by the above value equivalences we have

$$\langle v_1, w_1 \rangle \approx_\zeta \langle v_2, w_2 \rangle : (s_1 \times s_2)_\ell$$

and thus the expression equivalence

$$\langle \gamma_1(e_1), \gamma_1(e_2) \rangle \approx_\zeta \langle \gamma_2(e_1), \gamma_2(e_2) \rangle : (s_1 \times s_2)_\ell$$

This is the required result.

Case (TP-FST):

Suppose we have $\Gamma \vdash \mathbf{fst}(e) : s_1 \sqcup \ell$ using the (TP-FST) rule. Then we have the derivation

$$\Gamma \vdash e : (s_1 \times s_2)_\ell$$

Apply induction to receive the equivalence

$$\gamma_1(e) \approx_\zeta \gamma_2(e) : (s_1 \times s_2)_\ell$$

We thus have the evaluations

$$\gamma_1(e) \Downarrow v \quad \gamma_2(e) \Downarrow w$$

with

$$v \approx_\zeta w : (s_1 \times s_2)_\ell$$

Thus we have

$$v \equiv \langle v_1, w_1 \rangle \quad w \equiv \langle v_2, w_2 \rangle$$

If $\ell \not\sqsubseteq \zeta$, then $lvl(s_1) \sqcup \ell \not\sqsubseteq \zeta$ and so the values are related at the proper security level. Otherwise $\ell \sqsubseteq \zeta$, and the following equivalences hold.

$$v_1 \approx_\zeta v_2 : s_1 \quad w_1 \approx_\zeta w_2 : s_2$$

By Lemma 5.7,

$$v_1 \approx_\zeta v_2 : lvl(s_1) \sqcup \ell$$

We thus have

$$\gamma_1(\mathbf{fst}(e)) \approx_\zeta \gamma_2(\mathbf{fst}(e)) : lvl(s_1) \sqcup \ell$$

This is the required result.

Case (TP-SND):

The proof of this case is symmetric to the proof for (TP-FST).

Case (TP-IFTHEN):

Suppose $\Gamma \vdash$ **if** e_1 **then** e_2 **else** $e_3 : s$: by inversion we have the derivation

$$\Gamma \vdash e_1 : \mathsf{bool}_\ell$$

and the derivations

$$\Gamma \vdash e_2 : s \quad \Gamma \vdash e_3 : s$$

Apply induction to the first derivation to receive

$$\gamma_1(e_1) \approx_\zeta \gamma_2(e_1) : \mathsf{bool}_\ell$$

We then have the evaluations

$$\gamma_1(e_1) \Downarrow v_1 \quad \gamma_2(e_1) \Downarrow v_2$$

and the equivalence

$$v_1 \approx_\zeta v_2 : \mathsf{bool}_\ell$$

If $\ell \not\sqsubseteq \zeta$, then the if-then expressions are related by definition. Otherwise as v_1 and v_2 are equivalent at a basic type and $\ell \sqsubseteq \zeta$, then $v_1 = v_2$ and so the same evaluation rule applies to evaluating the entire if-then expression. Assume without loss of generality that the (EV-IF1) rule applies: then apply induction to the typing derivation of e_2 to receive

$$\gamma_1(e_2) \approx_\zeta \gamma_2(e_2) : s$$

We thus have

$$\gamma_1(e_2) \Downarrow v_1' \quad \gamma_2(e_2) \Downarrow v_2'$$

and the equivalence

$$v_1' \approx_\zeta v_2' : s$$

By Lemma 5.7,

$$v_1' \approx_\zeta v_2' : s \sqcup \ell$$

As this is the result of the evaluation of the entire if statement, we have:

$$\gamma_1(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) \approx_\zeta \gamma_2(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) : s \sqcup \ell$$

This is the required result.

Case (TP-LAM):

We have $\Gamma \vdash (\lambda x.e)_l : (s_1 \to s)_\ell$.

Suppose $v_1 \approx_\zeta v_2 : s_1$: we must show $e[v_1/x] \approx_\zeta e[v_2/x] : s$. If $\ell \not\sqsubseteq \zeta$ this holds as the expressions type correctly: thus, assume $\ell \sqsubseteq \zeta$. As $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$, we have

$$\Gamma[x : s_1] \vdash \gamma_1\{x \mapsto v_1\} \approx_\zeta \gamma_2\{x \mapsto v_2\}$$

By induction on the typing derivation

$$\Gamma[x : s_1] \vdash e : s$$

we then have the equivalence

$$\gamma_1\{x \mapsto v_1\}(e) \approx_\zeta \gamma_2\{x \mapsto v_2\}(e) : s \sqcup \ell$$

because $x \notin \mathrm{dom}(\Gamma)$ we can rewrite this as

$$\gamma_1(e[v_1/x]) \approx_\zeta \gamma_2(e[v_2/x]) : s \sqcup \ell$$

and thus

$$\gamma_1((\lambda x.e)_\ell) \approx_\zeta \gamma_2((\lambda x.e)_\ell) : s \sqcup \ell$$

This is the required result.

Case (TP-APP):

We have $\Gamma \vdash e_1\ e_2 : s \sqcup \ell$ and so we have the deductions

$$\Gamma \vdash e_1 : (s_1 \to s)_\ell \quad \Gamma \vdash e_2 : s_1$$

Apply induction to both of these smaller derivations the receive the equivalences

$$\gamma_1(e_1) \approx_\zeta \gamma_2(e_1) : (s_1 \to s)_\ell \quad \gamma_1(e_2) \approx_\zeta \gamma_2(e_2) : s_1$$

We thus have the evaluations

$$\gamma_1(e_1) \Downarrow (\lambda x_1.e_1')_\ell \quad \gamma_2(e_1) \Downarrow (\lambda x_2.e_2')_\ell$$
$$\gamma_1(e_2) \Downarrow v_1 \quad \gamma_1(e_2) \Downarrow v_2$$

and the equivalences

$$(\lambda x_1.e_1')_\ell \approx_\zeta (\lambda x_2.e_2')_\ell : (s_1 \to s)_\ell \quad v_1 \approx_\zeta v_2 : s_1$$

By the definition of $\approx_\zeta$ for function types, we have

$$e_1'[v_1/x_1] \approx_\zeta e_2'[v_2/x_2] : s \sqcup \ell$$

and thus

$$e_1'[v_1/x_1] \Downarrow v_1' \quad e_2'[v_2/x_2] \Downarrow v_2'$$

and so

$$\gamma_1(e_1\ e_2) \approx_\zeta \gamma_2(e_1\ e_2) : s \sqcup \ell$$

This is the required result.

Case (TP-SUB):

This case follows directly from Lemma 5.7. □

Appendix B

To demonstrate our Jif extension, we have developed a class which performs DES encryption and decryption. The `encrypt` method in the `DES` object is marked with a `declassFor` constraint to indicate that it is trusted to encrypt the data of a given principal. The code is given in Figure 14.

$$\frac{\pi(b_\ell) = \iota_\ell}{\Gamma \vdash b : \iota}\ (\textsc{tp-bval}) \quad \frac{\Gamma(x) = s}{\Gamma \vdash x : s}\ (\textsc{tp-var})$$

$$\frac{\Gamma \vdash e_1 : s_1 \quad \Gamma \vdash e_1 : s_2 \quad \ell = lvl(s_1) \sqcup lvl(s_2)}{\Gamma \vdash \langle e_1, e_2 \rangle : (s_1 \times s_2)_\ell}\ (\textsc{tp-prod})$$

$$\frac{\Gamma \vdash e : (s_1 \times s_2)_\ell}{\Gamma \vdash \mathbf{fst}(e) : s_1 \sqcup \ell}\ (\textsc{tp-fst}) \quad \frac{\Gamma \vdash e : (s_1 \times s_2)_\ell}{\Gamma \vdash \mathbf{snd}(e) : s_2 \sqcup \ell}\ (\textsc{tp-snd})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{bool}_\ell \\ \Gamma \vdash e_2 : s \quad \Gamma \vdash e_3 : s\end{array}}{\Gamma \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : s \sqcup \ell}\ (\textsc{tp-ifthen})$$

$$\frac{\Gamma[x : s_1] \vdash e : s \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash (\lambda x.e)_\ell : (s_1 \to s)_\ell}\ (\textsc{tp-lam})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (s_1 \to s)_\ell \\ \Gamma \vdash e_2 : s_1\end{array}}{\Gamma \vdash e_1\ e_2 : s \sqcup \ell}\ (\textsc{tp-app})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : s_1 \\ type(F) = (s' \to s)_{\ell_F} \\ s_1 \sqsubseteq s' \quad s_1 \not\sqsubseteq \ell_F\end{array}}{\Gamma \vdash F(e) : s \sqcup lvl(s_1)}\ (\textsc{tp-trans1}) \quad \frac{\begin{array}{c}\Gamma \vdash e : s_1 \\ type(F) = (s' \to s)_{\ell_F} \\ s_1 \sqsubseteq s' \quad s_1 \sqsubseteq \ell_F\end{array}}{\Gamma \vdash F(e) : s \sqcap \bot}\ (\textsc{tp-trans2})$$

$$\frac{\Gamma \vdash e : s \quad s \preceq s'}{\Gamma \vdash e : s'}\ (\textsc{tp-sub})$$

Figure 10: Typing Rules for $\lambda^{\mathcal{F}}_{sec}$

$$\frac{\begin{array}{c}e_1 \Downarrow (\lambda x.e)_\ell \\ e_2 \Downarrow v' \quad e[v'/x] \Downarrow v\end{array}}{e_1\ e_2 \Downarrow v}\ (\textsc{ev-app})$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\langle e_1, e_2 \rangle \Downarrow \langle v_1, v_2 \rangle}\ (\textsc{ev-prod})$$

$$\frac{e \Downarrow \langle v, v_1 \rangle}{\mathbf{fst}(e) \Downarrow v}\ (\textsc{ev-fst}) \quad \frac{e \Downarrow \langle v_1, v \rangle}{\mathbf{snd}(e) \Downarrow v}\ (\textsc{ev-snd})$$

$$\frac{e_1 \Downarrow \text{true} \quad e_2 \Downarrow v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v}\ (\textsc{ev-if1})$$

$$\frac{e_1 \Downarrow \text{false} \quad e_3 \Downarrow v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Downarrow v}\ (\textsc{ev-if2})$$

$$\frac{e \Downarrow v}{F(e) \Downarrow F(v)}\ (\textsc{ev-orac})$$

Figure 11: Evaluation Rules for $\lambda^{\mathcal{F}}_{sec}$

$$
\begin{array}{rcl}
v_1 \approx_\zeta v_2 : 1_\ell & \equiv_{def} & \text{for } i \in \{1,2\}, \vdash v_i : 1 \text{ and } \ell \sqsubseteq \zeta \text{ implies } v_1 = v_2 \\
(\lambda x_1.e_1)_\ell \approx_\zeta (\lambda x_2.e_2)_\ell : (s_1 \to s_2)_\ell & \equiv_{def} & \text{for } i \in \{1,2\}, \vdash (\lambda x_i.e_i)_\ell : (s_1 \to s_2)_\ell \text{ and} \\
 & & \ell \sqsubseteq \zeta \text{ implies } (\forall v_1' \approx_\zeta v_2' : s_1\ (e_1[v_1'/x_1] \approx_\zeta e_2[v_2'/x_2] : s_2 \sqcup \ell) \\
\langle v_1, w_1 \rangle \approx_\zeta \langle v_2, w_2 \rangle : (s_1 \times s_2)_\ell & \equiv_{def} & \text{for } i \in \{1,2\}, \vdash \langle v_i, w_i \rangle : (s_1 \times s_2)_\ell \text{ and} \\
 & & \ell \sqsubseteq \zeta \text{ implies } v_1 \approx_\zeta v_2 : s_1 \text{ and } w_1 \approx_\zeta w_2 : s_2 \\
F(v_1) \approx_\zeta F(v_2) : t_\ell & \equiv_{def} & \text{for } i \in \{1,2\}, \vdash F(v_i) : t_\ell \text{ and } type(F) = ((t_1)_{\ell_0} \to (t)_{\ell_0})_{\ell_F} \\
 & & \text{and } \ell \sqsubseteq \zeta \ \&\ \ell_F \sqsubseteq \zeta \text{ implies } v_1 \approx_\zeta v_2 : (t_1)_{\ell_F \sqcup \ell}
\end{array}
$$

Figure 13: Definition of the $\approx_\zeta$ Relation

```
public class DES [principal P] authority(DESprin)
{
  static public Key{P:} getNewKey()
         throws NoSuchAlgorithmException, NullPointerException
  {
    return KeyGenerator.getInstance("DES").generateKey();
  }

  static public Ciphertext{} encrypt(Key{P:} key, String{P:} s)
         where declassFor(DESprin,P)
  {
    Ciphertext{P:} ciphertext = null;

    try {
      Cipher{P:} desCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");

      desCipher.init(Cipher.ENCRYPT_MODE,key);

      final byte{P:}[]{P:} input = s.getBytes();
      final byte{P:}[]{P:} encrypted = desCipher.doFinal(input);

      ciphertext = new Ciphertext(new String(encrypted),
                                  new String(desCipher.getIV()));
    }
    catch (Exception e) {}

    return ciphertext;
  }

  static public String{P:} decrypt(Key{P:} key, Ciphertext{P:} ciph)
         throws (InvalidKeyException{P:}, IllegalBlockSizeException,
                 BadPaddingException,
                 NoSuchPaddingException, InvalidAlgorithmParameterException,
                 NoSuchAlgorithmException, NullPointerException)
  {
     Cipher{P:} desCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
     desCipher.init(Cipher.DECRYPT_MODE,
                    key,
                    new IvParameterSpec(ciph.iv.getBytes()));

     byte{P:}[]{P:} encrypted = ciph.encText.getBytes();
     String{P:} output = new String(desCipher.doFinal(encrypted));

     return output;
  }
}
```

Figure 14: Jif code for a DES class in which `encrypt` is trusted to declassify data.