

Faceted Execution of Policy-Agnostic Programs

Extended Version

Technical Report #UCSC-SOE-12-18

Thomas H. Austin¹, Jean Yang², Cormac Flangan¹, and Armando Solar-Lezama²

¹ UC Santa Cruz

² MIT CSAIL

Abstract. It is important for applications to protect sensitive data. Even for simple confidentiality and integrity policies, it is often difficult for programmers to reason about how the policies should interact and how to enforce policies across the program. A promising approach is *policy-agnostic programming*, a model that allows the programmer to implement policies separately from core functionality. Yang *et al.* describe Jeeves [1], a programming language that supports information flow policies describing how to reveal sensitive values in different output channels. Jeeves uses symbolic evaluation and constraint-solving to produce outputs adhering to the policies. This strategy provides strong confidentiality guarantees but limits expressiveness and implementation feasibility.

We describe a new language Jeeves* that provides the same guarantees while exploiting the structure of sensitive values to yield greater expressiveness and to facilitate reasoning about runtime behavior. We present a semantics based on Austin *et al.*'s *faceted execution* [2] describing a model for propagating multiple views of sensitive values through a program. We provide a proof of termination-insensitive non-interference and describe how the semantics facilitate reasoning about program behavior.

1 Introduction

It is increasingly important for applications to protect user privacy. Even for simple confidentiality and integrity policies, it is often difficult for programmers to reason about how the policies should interact and how to enforce policies across the program.

Policy-agnostic programming has the goal of allowing the programmer to implement core functionality separately from privacy policies. The programmer specifies policies as declarative rules and relies on the system to produce outputs adhering to the policies. Yang *et al.* describe Jeeves [1], a language that supports confidentiality policies describing how to reveal views of sensitive values based on the output channel. Sensitive values are pairs $\langle \ell ? v_H : v_L \rangle$, where v_H is the high-confidentiality value, v_L is the low-confidentiality value, and guard ℓ is a *level variable*. Jeeves uses symbolic evaluation and constraint-solving to produce outputs adhering to the policies. This strategy provides strong confidentiality

guarantees, but at the cost of expressiveness and implementation feasibility. For instance, Jeeves restricts recursion under symbolic conditionals and requires the cumulative constraint environment to persist.

In this paper, we present a new language Jeeves* that exploits the structure of sensitive values in order to increase expressiveness, facilitate reasoning about runtime behavior, and automatically enforce confidentiality policies. We base the Jeeves* evaluation strategy on Austin *et al.*'s *faceted execution* [2], which manipulates explicit representations of sensitive values. With this strategy, level variables are the only symbolic variables, allowing Jeeves* to lift restrictions on the flow of sensitive values. To further improve ease of reasoning, Jeeves* allows policies to only constrain level variables to **low**. This guarantees that the constraint environment is always consistent, a property that allows for policy garbage-collection.

We present a semantics based on Austin *et al.*'s faceted evaluation [2] describing how to propagate views of sensitive values. We describe λ^{jeeves} , which extends the faceted semantics with a declarative policy language. We prove termination-insensitive non-interference and policy compliance for λ^{jeeves} and describe how the semantics allows for reasoning about properties such as termination, consistency, and policy independence.

2 Jeeves* and Faceted Evaluation

We introduce how to use Jeeves* for confidentiality and then describe Jeeves* idioms for implementing a more complex health record database example.

In this section, we present Jeeves* using an ML-like concrete syntax, shown in Figure 1. Jeeves* extends the λ -calculus with references, facets ($\langle \ell ? Exp_H : Exp_L \rangle$), a **level** construct for introducing level variables that guard access to facets, and a **restrict** construct for introducing policies on level variables. Jeeves* statements include let-bound expressions and the effectful **print** statement.

2.1 Jeeves* for Confidentiality

Jeeves* allows the programmer to introduce a variable `name` that can be either "Alice" or "Anonymous" depending on the output channel:

```
let name: string = level a in <a ? "Alice" : "Anonymous" >
```

This code introduces a level variable `a` that determines whether the private (high-confidentiality) value "Alice" or the public (low-confidentiality) "Anonymous" should be revealed. Level variables take on the values { **low**, **high** }.

A simple policy on a sensitive value `name` is that the user must be the user `alice` to have high-confidentiality status:

```
let name: string = level a in  
  restrict a:  $\lambda(c: \text{User}).(c == \text{alice})$  in <a ? "Alice" : "Anonymous">
```

x		variables
ℓ		level variables
p, r		primitives, records
k	::= $p \mid r$	principals
$Level$::= low \mid high	levels
τ	::= int \mid bool \mid string \mid record $\overline{x} : \vec{\tau}$	types
	$\mid \tau_2 \rightarrow \tau_2 \mid \tau \text{ ref } \mid Level$	
Exp	::= $x \mid p \mid r \mid k$	expressions
	$\mid \lambda x : \tau. Exp \mid Exp_1 \text{ (op) } Exp_2$	
	$\mid \text{if } Exp_1 \text{ then } Exp_t \text{ else } Exp_f$	
	$\mid Exp_1 Exp_2$	
	$\mid \text{ref } Exp \text{ (wrestrict } Exp_p)^+$	
	$\mid ! Exp \mid x := Exp \text{ in } Exp_b$	
	$\mid \langle \ell ? Exp_{high} : Exp_{low} \rangle$	
	$\mid \text{level } \ell \text{ in } Exp$	
	$\mid \text{restrict } \ell : Exp_p \text{ in } Exp$	
$Stmt$::= let $x : \tau = Exp < k >$ in Exp_b	statements
	$\mid \text{print } \{Exp_c\} Exp$	

Fig. 1: Jeeves* syntax.

The **restrict** statement introduces a rule that strengthens the policy relating the output channel to the high-confidentiality value. To produce an assignment to level variables, the Jeeves* system translates this rule to the declarative constraint $\neg!(c = \text{alice}) \wedge (a == \text{low})$. This rule is not used until evaluation of **print**, so other policies could further restrict the level variable to be **low**.

In Jeeves* programs, sensitive values can be used as regular program values and effectful statements such as **print** require a context parameter:

```
let msg: string = "Sender is " + name;
print { alice } msg; /* Output: "Sender is Alice" */
print { bob } msg   /* Output: "Sender is Anonymous" */
```

During program evaluation, the Jeeves* runtime ensures that only the user *alice* can see her name appearing as the author in the string *msg*. User *bob* sees the string "Sender is Anonymous".

Unlike Jeeves, which performs symbolic evaluation, Jeeves* evaluation propagates *faceted values*. For instance:

```
<a ? "Sender is Alice" : "Sender is Anonymous">
```

Producing concrete outputs involves finding assignments to level variables that satisfy the policies. The Jeeves* system tries to assign level variables to **high**, setting levels to **low** only if the policies require it. Assigning all level variables to **low** always yields a consistent solution.

Jeeves* allows the output channel to be sensitive:

```
let u: user = level b in
  restrict b:  $\lambda(c: \text{User}).(c = \text{alice})$  in <b ? alice : nobody>;
print {u} u.name
```

There is a circular dependency: the context is a sensitive value guarded by a policy depending on the context. Such a policy allows two outcomes: `b` is **high** and we display `alice.name` to user `alice` and `b` is **low** and we display `nobody.name` to user `nobody`. The Jeeves* runtime ensures maximal functionality: if the policies allow a level variable to be **high** or **low**, the value will be **high**.

2.2 A Health Database in Jeeves*

To show how to use Jeeves for real-world applications, let us build a simple health database with records of the following form:

```
type Patient { identity : User ref
                ; doctor: User ref
                ; meds: (Medication list) ref }
```

In these records, each of the fields `identity`, `doctor`, and `meds` could be sensitive values that show different values of the correct type to low-confidentiality output channels.

In this example, the output context has type `HealthCtxt`, which we define as follows:

```
type HealthCtxt { viewer: User, time: Date }
```

This context contains information not just for the viewer but also for the current date, allowing policies to define activation and expiration times for visibility.

The idiomatic way of attaching policies to a value is to create sensitive values for each field and then attach policies:

```
let mkPatient ( identity : User) (doctor: User) (meds: Medication list) : Patient =
  level np, dp, mp in
  let p = { identity = <np ? identity : nobody>
            ; doctor  = <dp ? doctor : nobody>
            ; meds    = <mp ? meds : []> in
    addNamePolicy      p np;
    addDoctorPolicy    p dp;
    addMedicationsPolicy p mp;
    p
```

This function introduces level variables, creates sensitive values, attaches policies to the level variables, and returns the resulting `Patient` record. The function makes use of the `add... Policy` functions for attaching policies to the level variables. The `add... Policy` functions take a `Patient` record and a level variable and uses the record fields to attach a policy to the level variable. We define `addMedicationsPolicy` as:

```
let addMedicationsPolicy (p: Patient) (mp: level): unit =
  restrict mp: λ(c: HealthCtxt) => (c.viewer == p.identity || c.viewer == p.doctor)
```

This policy sets the level to **low** unless the viewer is the patient or the patient's doctor. Jeeves* automatically handles dependencies between policies and sensitive values: to have access to the medication list, the viewer needs to be able to see that their identity is equal to either `p.identity` or `p.doctor`.

2.3 Faceted Execution and Advantages

Identity	Doctor
$\langle a ? \text{alice} : \text{default} \rangle$	$\langle e ? \text{erica} : \text{default} \rangle$
$\langle b ? \text{bob} : \text{default} \rangle$	$\langle f ? \text{fred} : \text{default} \rangle$
$\langle c ? \text{claire} : \text{default} \rangle$	$\langle f' ? \text{fred} : \text{default} \rangle$

Table 1. Sample patient records.

Explicit representation of facets allows the runtime to prune branches of execution. Consider the following function, which takes a list of patients and a doctor and calls `fold` to count of the number of patients with a doctor field matching the doctor argument, on the records in Table 1 with `doctor = erica`.

```
let countPatients (patients: Patient list) (doctor: User): int =
  fold (λ (p: Patient) . λ (accum: int) .
    (if (p.doctor == doctor) then (accum + 1) else accum))
    0 patients
```

Consider the behavior of this function on the records in Table 1 with a call to `countPatients` with `doctor = erica`. Evaluation of $\langle e ? \text{erica} : \text{default} \rangle = \text{erica}$ yields the expression $\langle e ? \text{erica} = \text{erica} : \text{default} = \text{erica} \rangle$, which can be simplified to $\langle e ? \text{true} : \text{false} \rangle$. Evaluation of faceted function applications creates a new faceted value resulting from applying the function to each facet. If `e` is in the set of path condition assumptions, then only the high facet is used. Evaluation of the conditional produces the expression

$$\langle e ? \text{if (true) then (accum + 1) else accum} : \text{if (false) ...} \rangle,$$

which simplifies to $\langle e ? \text{accum} + 1 : \text{accum} \rangle$. Depending on whether the output user is allowed to see that `p.doctor` is equal to `erica`, the resulting sum is either `accum` or `accum + 1`.

Storing an explicit representation for facets allows the runtime to prune branches. For instance, if the doctor is not equal to `erica` on either facet, then the faceted evaluation only needs to store a single value. The system may also prune facets based on path assumptions: if evaluation is occurring under the assumption that guard `k` is true, then subsequent evaluation can assume guard `k`. This is particularly advantageous when there are a small number of level variables corresponding to a fixed set of principals.

3 Core Semantics

We model the semantics of `Jeeves*` with $\lambda^{j\text{eeves}}$, a simple core language that extends the faceted execution semantics of Austin and Flanagan [2] with a declarative policy language for confidentiality. The $\lambda^{j\text{eeves}}$ semantics describes how to

Syntax:

$e ::=$		<i>Term</i>
x		variable
c		constant
$\lambda x.e$		abstraction
$e_1 e_2$		application
$\text{ref } e$	reference allocation	
$!e$	dereference	
$e := e$	assignment	
$\langle k ? e_1 : e_2 \rangle$	faceted expression	
level k in e	level variable declaration	
$\text{restrict}(k, e)$	policy specification	
$S ::=$		<i>Statement</i>
let $x = e$ in S	let statement	
print $\{e\} e$	print statement	
$c ::=$		<i>Constant</i>
f	file handle	
b	boolean	
i	integer	
x, y, z		<i>Variable</i>
k, l	Label (aka Level Variable)	

Standard encodings:

$true$	$\stackrel{\text{def}}{=} \lambda x. \lambda y. x$
$false$	$\stackrel{\text{def}}{=} \lambda x. \lambda y. y$
if e_1 then e_2 else e_3	$\stackrel{\text{def}}{=} (e_1 (\lambda d. e_2) (\lambda d. e_3)) (\lambda x. x)$
if e_1 then e_2	$\stackrel{\text{def}}{=} \text{if } e_1 \text{ then } e_2 \text{ else } 0$
let $x = e_1$ in e_2	$\stackrel{\text{def}}{=} (\lambda x. e_2) e_1$
$e_1 \wedge_f e_2$	$\stackrel{\text{def}}{=} \lambda x. e_1 x \wedge e_2 x$
$e_1 \wedge e_2$	$\stackrel{\text{def}}{=} \text{if } e_1 \text{ then } e_2 \text{ else } false$

Fig. 2: The source language λ^{jeves}

evaluate faceted values, store policies, and use the policy environment to provide assignments to level variables for producing concrete outputs. We use these semantics to prove non-interference and policy compliance guarantees.

We show the source syntax in Figure 2. The language λ^{jeeves} extends the λ -calculus with expressions for allocating references ($\text{ref } e$), dereferencing ($!e$), assignment ($e_1 := e_2$), creating faceted expressions ($\langle k ? e_1 : e_2 \rangle$), specifying policy ($\text{restrict}(k, e)$), and declaring level variables ($\text{level } k \text{ in } e$). Additional statements exist for let-statements ($\text{let } x = e \text{ in } S$) and printing output ($\text{print } \{e_1\} e_2$). Conditionals are encoded in terms of function application.

In λ^{jeeves} , values V contain *faceted values* of the form

$$\langle k ? V_H : V_L \rangle$$

A viewer authorized to see k -sensitive data will observe the private facet V_H . Other viewers will instead see V_L . For example, the value $\langle k ? 42 : 0 \rangle$ specifies a value of 42 that should only be viewed when k is **high** according to the policy associated with k . When the policy specifies **low**, the observed value should instead be 0.

A *program counter label* pc records when execution is influenced by public or private facets. For instance, in the conditional test

$$\text{if } (\langle k ? \text{true} : \text{false} \rangle) \text{ then } e_1 \text{ else } e_2$$

our semantics needs to evaluate both e_1 and e_2 . The label k is added to pc during the evaluation of e_1 . By doing so, our semantics records the influence of k on this computation. Similarly, \bar{k} is added to pc during the evaluation of e_2 to record that the execution should have no effects observable to k . A *branch* h is either a level variable k or its negation \bar{k} . Therefore pc is a set of branches that never contains both k and \bar{k} , since that would reflect influences from both the private and public facet of a value.

The operation $\langle pc ? V_1 : V_2 \rangle$ creates a faceted value. The value V_1 is visible when the specified policies correspond with *all* branches in pc . Otherwise, V_2 is visible instead.

$$\begin{aligned} \langle \emptyset ? V_n : V_o \rangle &\stackrel{\text{def}}{=} V_n \\ \langle \{k\} \cup \text{rest} ? V_n : V_o \rangle &\stackrel{\text{def}}{=} \langle k ? \langle \text{rest} ? V_n : V_o \rangle : V_o \rangle \\ \langle \{\bar{k}\} \cup \text{rest} ? V_n : V_o \rangle &\stackrel{\text{def}}{=} \langle k ? V_o : \langle \text{rest} ? V_n : V_o \rangle \rangle \end{aligned}$$

For example, $\langle \{k, l\} ? V_H : V_L \rangle$ returns $\langle k ? \langle l ? V_H : V_L \rangle : V_L \rangle$. We occasionally abbreviate $\langle \{k\} ? V_H : V_L \rangle$ as $\langle k ? V_H : V_L \rangle$.

The semantics are defined via the big-step evaluation relation:

$$\Sigma, e \Downarrow_{pc} \Sigma', V$$

This relation evaluates an expression e in the context of a store Σ and program counter label pc . It returns a modified store Σ' reflecting updates and a value V . In Figure 3 we show the evaluation rules, which uses additional runtime syntax (also shown in Figure 3).

Runtime Syntax	
$e \in Expr$	$::= \dots \mid a$
$\Sigma \in Store$	$= (Address \rightarrow_p Value) \cup (Label \rightarrow Value)$
$R \in RawValue$	$::= c \mid a \mid (\lambda x.e)$
$a \in Address$	
$V \in Val$	$::= R \mid \langle k ? V_1 : V_2 \rangle$
$h \in Branch$	$::= k \mid \bar{k}$
$pc \in PC$	$= 2^{Branch}$

Expression Evaluation Rules	
$\Sigma, R \Downarrow_{pc} \Sigma, R$	[F-VAL]
$\Sigma, e \Downarrow_{pc} \Sigma', V'$ $a \notin dom(\Sigma')$ $V = \langle pc ? V' : 0 \rangle$	[F-REF]
$\Sigma, e \Downarrow_{pc} \Sigma', V$ $V' = deref(\Sigma', V, pc)$	[F-DEREF]
$\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1$ $\Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V_2$ $\Sigma' = assign(\Sigma_2, pc, V_1, V')$	[F-ASSIGN]
$\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1$ $\Sigma_1, e_2 \Downarrow_{pc \cup \{k\}} \Sigma', V_2$ $V' = \langle k ? V_1 : V_2 \rangle$	[F-SPLIT]
$\Sigma, e_1 \Downarrow_{pc} \Sigma', V$ $\Sigma, e_2 \Downarrow_{pc} \Sigma', V$	[F-LEFT]
$\Sigma, e_2 \Downarrow_{pc} \Sigma', V$ $\Sigma, e_1 \Downarrow_{pc} \Sigma', V$	[F-RIGHT]
$\Sigma[k' := \lambda x.true], e[k := k'] \Downarrow_{pc} \Sigma', V$ $\Sigma, level\ k\ in\ e \Downarrow_{pc} \Sigma', V'$	[F-LEVEL]
$\Sigma, e \Downarrow_{pc} \Sigma_1, V$ $\Sigma' = \Sigma_1[k := \Sigma_1(k) \wedge_f \langle pc \cup \{k\} ? V : \lambda x.true \rangle]$	[F-RESTRICT]

Auxiliary Functions	
$deref : Store \times Val \times PC \rightarrow Val$	
$deref(\Sigma, a, pc) = \Sigma(a)$	
$deref(\Sigma, \langle k ? V_H : V_L \rangle, pc) =$	$\begin{cases} deref(\Sigma, V_H, pc) & \text{if } k \in pc \\ deref(\Sigma, V_L, pc) & \text{if } \bar{k} \in pc \\ \langle k ? deref(\Sigma, V_H, pc) : deref(\Sigma, V_L, pc) \rangle & \text{otherwise} \end{cases}$
$assign : Store \times PC \times Val \times Val \rightarrow Store$	
$assign(\Sigma, pc, a, V) = \Sigma[a := \langle pc ? V : \Sigma(a) \rangle]$	
$assign(\Sigma, pc, \langle k ? V_H : V_L \rangle, V) = \Sigma'$	where $\Sigma_1 = assign(\Sigma, pc \cup \{k\}, V_H, V)$ and $\Sigma' = assign(\Sigma_1, pc \cup \{\bar{k}\}, V_L, V)$

Fig. 3: Faceted Evaluation Semantics

Our language includes support for reference cells, which introduce additional complexities in handling implicit flows. The rule [F-REF] handles reference allocation (ref e). It evaluates an expression e , encoding any influences from the program counter pc to the value V , and adds it to the store Σ' at a fresh address a . Facets in V inconsistent with pc are set to 0. (Critically, to maintain non-interference, $\Sigma(a) = 0$ for all a not in the domain of Σ .)

The rule [F-DEREF] for dereferencing (! e) evaluates the expression e to a value V , which should either be an address or a faceted values where all of the “leaves” are addresses. The rule uses a helper function $deref(\Sigma', V, pc)$ (defined in Figure 3), which takes the addresses from V , retrieves the appropriate values from the store Σ' , and combines them in the return value V' . As an optimization, addresses that are not compatible with pc are ignored.

The rule [F-ASSIGN] for assignment ($e_1 := e_2$) is similar to [F-DEREF]. It evaluates e_1 to a possibly faceted value V_1 corresponding to an address and e_2 to a value V' . The helper function $assignOp(\Sigma_2, pc, V_1, V')$ defined in Figure 3 decomposes V_1 into separate addresses, storing the appropriate facets of V' into the returned store Σ' . The changes to the store may come from both V_1 and pc .

The rule [F-LEVEL] handles dynamic allocation of level variables (level k in e), adding a fresh level variable to the store with the policy of $\lambda x.true$, the default for any level variable. Any occurrences of k in e are α -renamed to k' and the expression is evaluated with the updated store. Policies may be further refined (restrict(k, e)) by the rule [F-RESTRICT], which evaluates e to a policy V that should be either a lambda or a faceted value comprised of lambdas. The additional policy check is restricted by pc , so that policy checks cannot themselves leak data. It is then joined with the existing policy for k , ensuring that policies can only become more restrictive.

When a faceted expression ($\langle k ? e_1 : e_2 \rangle$) is evaluated, both sub-expressions must be evaluated in sequence, as per the rule [F-SPLIT]. The influence of k is added to the program counter for the evaluation of e_1 to V_1 and \bar{k} for the evaluation of e_2 to V_2 , tracking the branch of code being taken. The results of both evaluations are joined together in the operation $\langle\langle k ? V_1 : V_2 \rangle\rangle$. As an optimization, only one expression is evaluated if the program counter already contains either k or \bar{k} , as indicated by the rules [F-LEFT] and [F-RIGHT].

Function application ($e_1 e_2$) is somewhat complex in the presence of faceted values. The rule [F-APP] evaluates e_1 to V_1 , which should either be a lambda or a faceted value containing lambdas, and evaluates e_2 to the function argument V_2 . It then delegates the application ($V_1 V_2$) to an auxiliary relation defined in Figure 4:

$$\Sigma, (V_1 V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V'$$

This relation breaks apart faceted values and tracks the influences of the level variables through the rules [FA-SPLIT], [FA-LEFT], and [FA-RIGHT] in a similar manner to the rules [F-SPLIT], [F-LEFT], and [F-RIGHT] discussed previously. The actual application is handled by the [FA-FUN] rule. The body of the lambda ($\lambda x.e$) is evaluated with the variable x replaced by the argument V .

Application Rules	$\Sigma, (V_1 \ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V'$
<div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p>[FA-FUN]</p> $\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', V'}{\Sigma, ((\lambda x.e) V) \Downarrow_{pc}^{\text{app}} \Sigma', V'}$ </div> <div style="width: 45%;"> <p>[FA-LEFT]</p> $\frac{\begin{array}{c} k \in pc \\ \Sigma, (V_H \ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V \end{array}}{\Sigma, (\langle k ? V_H : V_L \rangle V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V}$ </div> </div>	
<div style="display: flex; justify-content: space-between;"> <div style="width: 45%;"> <p>[FA-SPLIT]</p> $\frac{\begin{array}{c} k \notin pc \quad \bar{k} \notin pc \\ \Sigma, (V_H \ V_2) \Downarrow_{pc \cup \{k\}}^{\text{app}} \Sigma_1, V_H \\ \Sigma_1, (V_L \ V_2) \Downarrow_{pc \cup \{\bar{k}\}}^{\text{app}} \Sigma', V_L' \\ V' = \langle\langle k ? V_H' : V_L' \rangle\rangle \end{array}}{\Sigma, (\langle k ? V_H : V_L \rangle V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V'}$ </div> <div style="width: 45%;"> <p>[FA-RIGHT]</p> $\frac{\bar{k} \in pc}{\Sigma, (V_L \ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V}$ $\frac{}{\Sigma, (\langle k ? V_H : V_L \rangle V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V}$ </div> </div>	
Statement Evaluation Rules	$\Sigma, S \Downarrow V_p, f : R$
<p>[F-LET]</p> $\frac{\Sigma, S[x := V] \Downarrow V_p, f : R}{\Sigma, \text{let } x = e \text{ in } S \Downarrow V_p, f : R}$	<p>[F-PRINT]</p> $\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{\emptyset} \Sigma_1, V_f \\ \Sigma_1, e_2 \Downarrow_{\emptyset} \Sigma_2, V_c \\ e_p = \lambda x. \text{true} \wedge_f \Sigma_2(k_1) \wedge_f \dots \wedge_f \Sigma_2(k_n) \\ \Sigma_2, e_p \ V_f \Downarrow_{\emptyset} \Sigma_3, V_p \\ \{ k_1 \dots k_n \} \text{ includes all labels in } V_f, V_c, V_p \\ \text{pick } pc \text{ such that } pc(V_f) = f, \\ pc(V_c) = R, pc(V_p) = \text{true} \end{array}}{\Sigma, \text{print } \{e_1\} \ e_2 \Downarrow V_p, f : R}$
Semantics for Derived Encodings	
<p>[F-IF-TRUE]</p> $\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{pc} \Sigma_1, \text{true} \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V \end{array}}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$	<p>[F-IF-SPLIT]</p> $\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k ? V_H : V_L \rangle \\ e_H = \text{if } V_H \text{ then } e_2 \text{ else } e_3 \\ e_L = \text{if } V_L \text{ then } e_2 \text{ else } e_3 \\ \Sigma_1, \langle k ? e_H : e_L \rangle \Downarrow_{pc} \Sigma', V \end{array}}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$
<p>[F-IF-FALSE]</p> $\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{pc} \Sigma_1, \text{false} \\ \Sigma_1, e_3 \Downarrow_{pc} \Sigma', V \end{array}}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V}$	

Fig. 4: Faceted Evaluation Semantics for Application and Statements

Conditional branches (if e_1 then e_2 else e_3) are Church-encoded as function calls for the sake of simplicity. However, Figure 4 shows direct rules for evaluating conditionals in the presence of faceted values. Under the rule [F-IF-SPLIT], If the condition e_1 evaluates to a faceted value $\langle k ? V_H : V_L \rangle$, the if statement is evaluated twice with V_H and V_L as the conditional tests.

While expressions handle most of the complexity of faceted values, statements in $\lambda^{j\text{eeves}}$ illustrate how faceted values may be concretized when exporting data to an external party. The semantics for statements are defined via the big-step evaluation relation:

$$\Sigma, S \Downarrow_{pc} \Sigma' V$$

The rules for statements are specified in Figure 4. The rule [F-LET] handles let expressions (let $x = e$ in S), evaluating an expression e to a value V , performing the proper substitution in statement S . The rule [F-PRINT] handles print statements (print $\{e_1\} e_2$), where the result of evaluating e_2 is printed to the channel resulting from the evaluation of e_1 . Both the channel V_f and the value to print V_c may be faceted values, and furthermore, we must select the facets that correspond with our specified policies. The expression e_p contains all relevant policies included in the store Σ_2 . It is evaluated and applied to V_f , returning the policy check V_p that is a faceted value containing booleans. A program counter pc is chosen such that the policies are satisfied, which determines the channel f and the value to print R . Note that there exists a $pc' \in PC$ where all branches are set to **low**, which may always be displayed, thereby ensuring that there is always at least one valid choice for pc .

This property allows garbage collection of policies and facets. Because the constraints are always consistent, the only set of policies relevant to an expression e to output are associated with the transitive closure of level variables L_e appearing in e and the policies associated with L_e . Thus any policy associated with an out-of-scope variable may be garbage-collected. In addition, once a policy has been set to the equivalent of $\lambda x.false$ for a level variable k , k -sensitive facets and policies cannot be used in a print statement. Jeeves policies do not have such properties since additional policies could introduce inconsistencies.

4 Properties

We prove that a single execution with faceted values is equivalent to multiple different executions without faceted values. From this we know that if execution terminates on each facet of a sensitive value, then faceted execution terminates. Jeeves does not have this property because execution keeps sensitive values as symbolic; thus Jeeves restricts applications of recursive functions.

We also prove that the system cannot leak sensitive information either via the output or by the choice of output channel.

4.1 Projection Theorem

A key property of faceted evaluation is that it simulates multiple executions. In other words, a single execution with faceted values *projects* to multiple different executions without faceted values.

$$\begin{array}{lcl}
pc : Expr \text{ (with facets)} & \rightarrow & Expr \text{ (with fewer facets)} \\
pc(\langle k ? e_1 : e_2 \rangle) & = & \begin{cases} pc(e_1) & \text{if } k \in pc \\ pc(e_2) & \text{if } \bar{k} \in pc \\ pc(\langle k ? e_1 : e_2 \rangle) & \text{otherwise} \end{cases} \\
pc(\langle k ? V_1 : V_2 \rangle) & = & \begin{cases} pc(V_1) & \text{if } k \in pc \\ pc(V_2) & \text{if } \bar{k} \in pc \\ pc(V_1) & \text{if } pc(V_1) = pc(V_2) \\ pc(\langle k ? V_1 : V_2 \rangle) & \text{otherwise} \end{cases} \\
pc(\dots) & = & \text{compatible closure}
\end{array}$$

We extend pc to project faceted stores $\Sigma \in Store$ into stores with fewer facets.

$$\begin{array}{lcl}
pc : Value & \rightarrow & Value \\
pc(\Sigma) & = & \lambda a. pc(\Sigma(a)) \cup \lambda k. pc(\Sigma(k))
\end{array}$$

Thus pc projection does not remove policies, it only removes some labels on expressions or values. We say that pc_1 and pc_2 are *consistent* if

$$\neg \exists k. (k \in pc_1 \wedge \bar{k} \in pc_2) \vee (\bar{k} \in pc_1 \wedge k \in pc_2)$$

We note some key lemmas regarding projection.

Lemma 1 *If $V = \langle pc ? V_1 : V_2 \rangle$ then $\forall q \in PC$*

$$q(V) = \begin{cases} \langle pc \setminus q ? q(V_1) : q(V_2) \rangle & \text{if } q \text{ is consistent with } pc \\ q(V_2) & \text{otherwise} \end{cases}$$

Lemma 2 *If $V' = deref(\Sigma, V, pc)$ then $\forall q \in PC$ where q is consistent with pc , $q(V') = deref(q(\Sigma), q(V), pc \setminus q)$.*

Lemma 3 *If $\Sigma' = assign(\Sigma, pc, V_1, V_2)$ then $\forall q \in PC$*

$$q(\Sigma') = \begin{cases} assign(q(\Sigma), pc \setminus q, q(V_1), q(V_2)) & \text{if } q \text{ consistent with } pc \\ q(\Sigma) & \text{otherwise} \end{cases}$$

Lemma 4 *Suppose pc and q are not consistent and that either*

$$\begin{array}{l}
\Sigma, e \Downarrow_{pc} \Sigma', V \\
\text{or } \Sigma, (V_1 V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V
\end{array}$$

Then $q(\Sigma) = q(\Sigma')$.

The following projection theorem shows how a single faceted evaluation simulates (or projects) to multiple executions, each with fewer facets, or possibly with no facets at all (if for each label k in the program, either k or \bar{k} is in q).

Theorem 1 (Projection Theorem). *Suppose $\Sigma, e \Downarrow_{pc} \Sigma', V$. Then for any $q \in PC$ where pc and q are consistent $q(\Sigma), q(e) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$*

This theorem significantly extends the projection property of Austin and Flanagan [2], in that it supports dynamic label allocation and flexible, dynamically specified policies, and is also more general in that it can either remove none, some, or all top-level labels in a program, depending on the choice of the projection PC q . A full proof of the projection theorem is available in the appendix.

4.2 Termination-Insensitive Non-Interference

The projection property captures that data from one collection of executions, represented by the corresponding set of branches pc , does not leak into any incompatible views, thus enabling a straightforward proof of non-interference.

Two faceted values are *pc-equivalent* if they have identical values for the set of branches pc . This notion of *pc*-equivalence naturally extends to stores ($\Sigma_1 \sim_{pc} \Sigma_2$) and expressions ($e_1 \sim_{pc} e_2$):

$$\begin{aligned} (V_1 \sim_{pc} V_2) &\text{ iff } pc(V_1) = pc(V_2) \\ (\Sigma_1 \sim_{pc} \Sigma_2) &\text{ iff } pc(\Sigma_1) = pc(\Sigma_2) \\ (e_1 \sim_{pc} e_2) &\text{ iff } pc(e_1) = pc(e_2) \end{aligned}$$

The notion of *pc*-equivalence and the projection theorem enable a concise statement and proof of termination-insensitive non-interference.

Theorem 2 (Termination-Insensitive Non-Interference).

Let pc be any set of branches. Suppose $\Sigma_1 \sim_{pc} \Sigma_2$ and $e_1 \sim_{pc} e_2$, and that:

$$\Sigma_1, e_1 \Downarrow_{\emptyset} \Sigma'_1, V_1 \quad \Sigma_2, e_2 \Downarrow_{\emptyset} \Sigma'_2, V_2$$

Then $\Sigma'_1 \sim_{pc} \Sigma'_2$ and $V_1 \sim_{pc} V_2$.

Proof. By the Projection Theorem:

$$\begin{aligned} pc(\Sigma_1), pc(e_1) &\Downarrow_{\emptyset} pc(\Sigma'_1), pc(V_1) \\ pc(\Sigma_2), pc(e_2) &\Downarrow_{\emptyset} pc(\Sigma'_2), pc(V_2) \end{aligned}$$

The *pc*-equivalence assumptions imply that $pc(\Sigma_1) = pc(\Sigma_2)$ and $pc(e_1) = pc(e_2)$. Hence $pc(\Sigma'_1) = pc(\Sigma'_2)$ and $pc(V_1) = pc(V_2)$ since the semantics is deterministic.

4.3 Termination-Insensitive Policy Compliance

While we have shown non-interference for a set of level variables, the level variables do not directly correspond to the output revealed to a given observer. In this section we show how we can prove termination-insensitive *policy compliance*; data is revealed to an external observer only if it is allowed by the policy specified in the program. Thus if S_1 and S_2 are terminating programs that differ only in k -labeled components and the computed policy V_i for each program does not permit revealing k -sensitive data to the output channel, then the set of possible outputs from each program is identical. Here, an output $f : v$ combines both the output channel f and the value v , to ensure that sensitive information is not leaked either via the output value or by the choice of output channel.

Theorem 3. *Suppose for $i \in 1, 2$:*

$$\begin{aligned} S_i &= \text{print } \{e\} \ C[\langle k ? e_i : e_i \rangle] \\ \emptyset, S_1 &\Downarrow V_{p1}, f_1 : R_1 \\ \emptyset, S_2 &\Downarrow V_{p2}, f_2 : R_2 \\ \forall pc' \text{ with } k \in pc', \quad &pc'(V_1) \neq \text{true} \text{ and } pc'(V_2) \neq \text{true} \end{aligned}$$

Then $\{ f : R \mid \emptyset, S_1 \Downarrow V_p, f : R \} = \{ f : R \mid \emptyset, S_2 \Downarrow V_p, f : R \}$.

Proof. We show left-to-right containment as follows. (The converse containment holds by a similar argument.) Suppose

$$\begin{aligned} \emptyset, S_1 &\Downarrow V_{p1}, f_1 : R_1 \\ \emptyset, S_2 &\Downarrow V_{p2}, f_2 : R_2 \end{aligned}$$

Then by the [F-PRINT] rule

$$\begin{aligned} \emptyset, e &\Downarrow_{\emptyset} \Sigma_{11}, V_{f1} \\ \Sigma_{11}, C[\langle k ? e_1 : e_1 \rangle] &\Downarrow_{\emptyset} \Sigma_{12}, V_{c1} \\ e_{p1} &= \Sigma_{12}(k_1) \wedge_f \dots \wedge_f \Sigma_{12}(k_n) \text{ where } \{ k_1 \dots k_n \} \\ &\text{includes all labels in } V_{f1}, V_{c1}, V_{p1} \\ \Sigma_{12}, e_{p1} \ V_{f1} &\Downarrow_{\emptyset} \Sigma_{13}, V_{p1} \\ pc_1(V_{f1}) &= f_1, \quad pc_1(V_{c1}) = R_1, \quad pc_1(V_{p1}) = \text{true}, \text{ so } \bar{k} \in pc_1 \end{aligned}$$

Also by the [F-PRINT] rule for the second execution

$$\begin{aligned} \emptyset, e &\Downarrow_{\emptyset} \Sigma_{21}, V_{f2} \\ \Sigma_{21}, C[\langle k ? e_2 : e_2 \rangle] &\Downarrow_{\emptyset} \Sigma_{12}, V_{c2} \\ e_{p2} &= \Sigma_{22}(k_1) \wedge_f \dots \wedge_f \Sigma_{22}(k_n) \text{ where } \{ k_1 \dots k_n \} \\ &\text{includes all labels in } V_{f2}, V_{c2}, V_{p2} \\ \Sigma_{22}, e_{p2} \ V_{f2} &\Downarrow_{\emptyset} \Sigma_{23}, V_{p2} \\ pc_2(V_{f2}) &= f_2, \quad pc_2(V_{c2}) = R_2, \quad pc_2(V_{p2}) = \text{true}, \text{ so } \bar{k} \in pc_2 \end{aligned}$$

By determinism, $\Sigma_{11} = \Sigma_{21} \ V_{f1} = V_{f2}$.

Also, $C[\langle \bar{k} ? e_1 : e_1 \rangle] \sim_{\{\bar{k}\}} C[\langle k ? e_2 : e_2 \rangle]$. Hence by the projection theorem

$$\begin{aligned} \Sigma_{12} &\sim_{\{\bar{k}\}} \Sigma_{22} & V_{c1} &\sim_{\{\bar{k}\}} V_{c2} & e_{p1} &\sim_{\{\bar{k}\}} e_{p2} \\ \Sigma_{13} &\sim_{\{\bar{k}\}} \Sigma_{23} & V_{p1} &\sim_{\{\bar{k}\}} V_{p2} \end{aligned}$$

Pick $pc_2 = pc_1$. Then $R_2 = R_1$ and $f_2 = f_1$ as required.

5 Scala Implementation

We have implemented Jeeves* as an embedded domain-specific language in the Scala programming language [3]. We use Scala’s overloading capabilities to implement faceted execution, constraint collection, and interaction with the Z3 SMT solver [4].³ The implementation defines Scala classes for integers, booleans, objects, and functions that support operations over expressions e or faceted expressions $\langle k ? e_H : e_L \rangle$. The implementation overloads operators on these types so that faceted values can be used interchangeably with concrete values. For instance, the `Expr[Int]` class represents the type of concrete and faceted integer expressions. We use Scala’s implicit type conversions to lift concrete Scala values.

We have implemented a Scala trait that stores a runtime environment to support methods creating level variables, declaring policies, and concretizing expressions. The trait maintains the logical and default constraint environments as lists of functions of type `Expr[T] ⇒ Formula`, where `Formula` is a boolean expression that may contain facets. We have a partial evaluation procedure that simplifies expressions based on the value of each facet and the current path assumptions.

To assign values to level variables, the implementation evaluates policies according to the context and heap state and invokes Z3 for resolving constraints. Our implementation translates constraints to the `QF_LIA` logic of SMT-LIB2 [5]. There are only quantifier-free boolean constraints. Level variables are the only free variables. We use incremental scripting to implement default values according to default logic [6]. The implementation relies on Scala’s support for dynamic invocation to resolve field dereferences. We use zero values (`null`, `0`, or `false`) to represent undefined fields in SMT.

Our Jeeves* library interface supports the introduction of level variables, declaration of policies, creation of sensitive variables, and concretization of sensitive expressions. It also has functions for assignment, conditionals, and function evaluation according to the Jeeves* semantics.

The library has the following API methods for introducing sensitive values and policies:

```
def mkLevel: LevelVar
def restrict (lvar : LevelVar, f: Expr[T] ⇒ Formula)
def mkSensitive(lvar : LevelVar , high: Expr[T], low: Expr[T]): Expr[T]
```

The programmer introduces level variables, which are boolean logic variables mapped to `HIGH` and `LOW`, into scope by calling the `mkLevel` method. The `restrict` method for introducing policies takes a level variable and a function that takes a context expression and returns a formula. The library stores policy functions and applies them with respect to the output context and output heap state to produce concrete outputs adhering to the policies. The programmer introduces sensitive values through the `mkSensitive` method, which takes a level variable along with high-confidentiality and low-confidentiality views. To support evaluation with sensitive expressions, programs should accommodate values of type `Expr[T]` (e.g. `IntExpr` rather than `BigInt`). The library has methods for producing concrete state:

³ The code is available at <http://code.google.com/p/jeeveslib/>.

```
def concretize [T](ctxt: Expr[T], e: Expr[T]): T
def jprint [T](ctxt: Expr[T], e: Expr[T]): Unit
```

These functions take a context and an expression, both of which may be sensitive, and provides assignments to the level variables to produce concrete views that adhere to the policies. The implementation treats the mutable state as part of the context in the `concretize` call to `ScalaSMT`. All classes that are used in constraint must extend the `JeevesRecord` class. The set of allocated `JeevesRecords` is supplied at concretization. This way, policies that refer to mutable parts of the heap will produce correct constraints for the snapshot of the system at concretization. The library provides support for evaluating conditionals and function applications:

```
def jif [T] (c: Formula, t : Unit => T, f: Unit => T): Unit
def jfun [A, B] ( f : FunctionExpr[A, B], arg : A): B
```

The library stores the path condition as a set of level variables and their negations. The `jif` method evaluates the condition and manages the path condition for each branch appropriately in order to produce a potentially faceted result. The `jfun` method behaves similarly. Both of these methods check against the path condition to avoid performing unnecessary computations.

6 Case Study: Conference Management

We have implemented `JConf`, a conference management system that uses `Jeeves*` for confidentiality guarantees. The `JConf` backend interacts with a web-based frontend and a persistent database store. The original `JConf` implementation, written using an implementation of `Jeeves` (which used symbolic evaluation rather than faceted execution), was up for several hours at a time and a cumulative total of several days, processing submissions for the Student Research Competition for the Programming Language Design and Implementation Conference 2012. Our experience with this system motivated some of the design decisions in `Jeeves*`, including the decision to use faceted execution.

The implementation of `JConf` has a backend written in `Jeeves*` that defines Scala objects corresponding to data types (for instance, for representing users and papers) and associates policies with fields of these objects; object constructors add the policies. The backend contains functionality that supports the creation of, lookup of, updates to, and search over these objects. The frontend web code, written using the `Scalatra` web framework [7], makes calls to the backend functionality and to accessors of the objects. The `JConf` backend contains a layer that interacts with a MySQL database for persistent storage. The frontend web code and database-interaction code remain agnostic to the policies: the same code is used, for instance, to render a page (for instance, displaying appropriately anonymized information about a paper review) for an author, a reviewer, and a program committee member. Interaction with the `Jeeves*` backend takes on the order of seconds; solving in the `Z3` SMT solver takes well under one second. The bulk of execution is involved in propagating sensitive values.

The JConf conference management system provides support for creating new users and updating profiles, creating papers and updating information, submitting papers, assigning reviews, and reviewing papers. We show the breakdown of the system in Table 2: classes describing the data (users, papers, paper reviews, and the context), backend code for accessing the data (including the interface to the database), the Scalatra code for the frontend web request handlers, and the Scalatra Server Page (SSP) code defining the browser pages themselves.

Policy code (calls to `mkLevel`, `mkSensitive`, `restrict`, and `concretize`) is concentrated in the data classes, enabling modular updates to the policy and core functionality. For instance, we can change the review process from double-blind to single-blind simply by tweaking the policies associated with paper and review fields. The policy code makes up less than 5% of the total lines of code.

File	Total LOC	Policy LOC
ConfUser.scala	212	21
PaperRecord.scala	304	75
PaperReview.scala	116	32
ConfContext.scala	6	0
Backend + Squeryl	800	0
Frontend (Scalatra)	629	0
Frontend (SSP)	798	0
Total	2865	128

Table 2. Lines of code vs. policy in JConf.

The programmer defines a *getter*, a *setter*, and a *show function* for each sensitive field. The getter returns the sensitive value, the setter creates a new sensitive value based on the views, and the show function calls `concretize` to return a concrete value of the appropriate type. The programmer creates the sensitive value with a level variable in scope to which policies can be attached. It may make sense to share level variables between field for some applications. The frontend calls the show functions to access concrete versions of values. We use the database only for persistent storage; all queries use Jeeves* to ensure policy compliance.

7 Related Work

Jeeves* follows a line of research in information flow of which Sabelfeld and Myers offer an extensive survey [8]. Information flow analysis traces its roots back to Denning [9]. Volpano *et al.* [10] develop a type system that guarantees non-interference for the language that Denning outlines. Heintze and Riecke [11] design a type system guaranteeing non-interference for a functional language, extended with constructs for reference cells, concurrency, and integrity guarantees. Smith [12] discusses some of the core concepts in information flow analysis.

Languages for verifying information flow security include Jif [13], Fine [14], F* [15], and Ur/Web [16]. Nanevski *et al.* [17] verify information flow policies through the use of dependent types. These static approaches have no dynamic overhead. Jeeves* mitigates programmer burden by guaranteeing that programs adhere to the desired properties by construction, but with dynamic overhead. Systems like Fabric [18] combine static and dynamic techniques, but the focus of the dynamic analyses is on checking rather than on helping the programmer produce correct outputs. Jeeves* is also related to systems that provide support for inserting information flow checks. Broberg and Sands [19] describe flow locks for dynamic information flow policies. Birgisson *et al.* [20] show how capabilities can guarantee information flow policies. The system-level data flow framework Resin [21] allows the programmer to insert checking code to be executed at output channels. Privacy Integrated Queries (PINQ) [22] is a capability-based system that enforces differential privacy policies in declarative database queries.

Capizzi *et al.*'s *shadow executions* [23] maintain confidentiality by running both a public and private copy of the application. The public copy can communicate with the outside world but cannot access private data; the private copy has access to private information but lacks network access. Devriese and Piessens' *secure multi-execution* strategy [24] applies this approach to JavaScript code.

Austin and Flanagan [2] simulate secure multi-execution with a single execution through the use of faceted values, avoiding overhead when code does not depend on confidential data, noticeably improving performance. The same paper also shows how declassification may be performed with facets, though with Jeeves*'s policies, declassification is largely unnecessary. Faceted values are related to the non-interference work by Pottier and Simonet for *Core ML* [25]. Their proof approach involves a Core ML² language that has expression pairs and value pairs, similar to faceted expressions and faceted values respectively.

The automatic policy enforcement is related to work in constraint functional programming and executing specifications. Like constraint functional languages, Jeeves* integrates declarative constraints into a non-declarative programming model. Jeeves* differs from languages such as Mercury [26], Escher [27], Curry [28], and Kaplan [29], which support rich operations over logic variables at the cost of potentially expensive runtime search and undecidability. In Jeeves*, the logical environment is always consistent and the runtime only performs decidable search routines. Jeeves* differs from the Squander system [30] for unified execution of imperative and declarative code in that Jeeves* propagates constraints alongside the core program rather than executing isolated constraint-based sub-procedures. As with relaxed approximate programs [31], Jeeves* nondeterministically provides an acceptable output for a specific class of acceptability properties.

Jeeves* is related to declarative domain-specific languages. Frenetic [32] provides a query language programming distributed collections of network switches. Engage [33] uses constraints to mitigate programmer burden in configuring, installing, and managing applications. Jeeves* differs in that its target domain of privacy is cross-cutting with respect to other functionality.

8 Conclusions

Jeeves* allows the programmer to implement core functionality separately from confidentiality policies. Our execution strategy exploits the structure of sensitive values to facilitate reasoning about runtime behavior. We present a semantics for faceted execution of Jeeves* in terms of the λ^{jeeves} core language, and prove non-interference and policy compliance for confidentiality. We describe how Jeeves* enables reasoning about termination, policy consistency, and policy independence. Finally, we describe our implementation of Jeeves* in Scala and our experience using Jeeves* to implement an end-to-end conference management system.

References

1. Yang, J., Yessenov, K., Solar-Lezama, A.: A language for automatically enforcing privacy policies. In: Symposium on Principles of Programming Languages (POPL). (2012)
2. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Symposium on Principles of Programming Languages (POPL). (2012)
3. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the scala programming language. Technical report, Citeseer (2004)
4. Moura, L.D., Björner, N.: Z3: An efficient SMT solver. In: Tools and algorithms for the construction and analysis of systems (TACAS). (2008)
5. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard: Version 2.0. In: SMT Workshop. (2010)
6. Antoniou, G.: A tutorial on default logics. *ACM Computing Surveys (CSUR)* **31**(4) (1999)
7. : Scalatra: A tiny Sinatra-like web framework for Scala. <http://www.scalatra.org/>
8. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* **21**(1) (2003)
9. Denning, D.E.: A lattice model of secure information flow. *Communications of the ACM* **19**(5) (1976)
10. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. *Journal of Computer Security* **4**(2-3) (1996)
11. Heintze, N., Riecke, J.G.: The SLam calculus: Programming with secrecy and integrity. In: Symposium on Principles of Programming Languages (POPL). (1998)
12. Smith, G.: Principles of secure information flow analysis. In Christodorescu, M., Jha, S., Maughan, D., Song, D., Wang, C., eds.: *Malware Detection*. Volume 27 of *Advances in Information Security*. Springer (2007) 291–307
13. Myers, A.C.: JFlow: Practical mostly-static information flow control. In: Symposium on Principles of Programming Languages (POPL). (1999)
14. Chen, J., Chugh, R., Swamy, N.: Type-preserving compilation of end-to-end verification of security enforcement. In: Conference on Programming Language Design and Implementation (PLDI). (2010)
15. Swamy, N., Chen, J., Fournet, C., Strub, P.Y., Bhargavan, K., Yang, J.: Secure distributed programming with value-dependent types. In: International Conference on Functional Programming (ICFP). (2011)

16. Chlipala, A.: Static checking of dynamically-varying security policies in database-backed applications. In: USENIX Conference on Operating Systems Design and Implementation (OSDI). (2012)
17. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: IEEE Symposium on Security and Privacy. (2011)
18. Liu, J., George, M.D., Vikram, K., Qi, X., Waye, L., Myers, A.C.: Fabric: a platform for secure distributed computation and storage. In: Symposium on Operating System Principles (SOSP). (2009)
19. Broberg, N., Sands, D.: Flow locks: Towards a core calculus for dynamic flow policies. In: European Symposium on Programming (ESOP). (2006)
20. Birgisson, A., Russo, A., Sabelfeld, A.: Capabilities for information flow. In: Workshop on Programming Languages and Analysis for Security (PLAS). (2011)
21. Yip, A., Wang, X., Zeldovich, N., Kaashoek, M.F.: Improving application security with data flow assertions. In: Symposium on Operating System Principles (SOSP). (2009)
22. McSherry, F.: Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In: International Conference on Management of Data (SIGMOD). (2009)
23. Capizzi, R., Longo, A., Venkatakrishnan, V., Sistla, A.: Preventing information leaks through shadow executions. In: Annual Computer Security Applications Conference (ACSAC). (dec 2008)
24. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. IEEE Symposium on Security and Privacy (2010)
25. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Transactions on Programming Languages and Systems (TOPLAS) **25**(1) (2003)
26. Somogyi, Z., Henderson, F.J., Conway, T.C.: Mercury, an efficient purely declarative logic programming language. In: Australian Computer Science Conference. (1995)
27. Lloyd, J.W.: Programming in an integrated functional and logic language. Journal of Functional and Logic Programming **3** (1999)
28. Hanus, M., Kuchen, H., Moreno-Navarro, J.J., Aachen, R., Ii, I.: Curry: A truly functional logic language (1995)
29. Köksal, A., Kuncak, V., Suter, P.: Constraints as control. In: Symposium on Principles of Programming Languages (POPL). (2012)
30. Milicevic, A., Rayside, D., Yessenov, K., Jackson, D.: Unifying execution of imperative and declarative code. In: International Conference on Software Engineering (ICSE). (2011)
31. Carbin, M., Kim, D., Misailovic, S., Rinard, M.: Proving acceptability properties of relaxed nondeterministic approximate programs. In: Conference on Programming Language Design and Implementation (PLDI). (2012)
32. Foster, N., Harrison, R., Freedman, M.J., Monsanto, C., Rexford, J., Story, A., Walker, D.: Frenetic: a network programming language. In: International Conference on Functional Programming (ICFP). (2011)
33. Fischer, J., Majumdar, R., Esmacilsabzali, S.: Engage: a deployment management system. In: Conference on Programming Language Design and Implementation (PLDI). (2012)

A Proof of Projection

With these properties established, we now prove projection. For convenience, we restate the projection theorem here.

Theorem 1. *Suppose $\Sigma, e \Downarrow_{pc} \Sigma', V$. Then for any $q \in PC$ where pc and q are consistent $q(\Sigma), q(e) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$*

Proof. We prove a stronger inductive hypothesis, namely that for any $q \in PC$ where $\neg \exists k. (k \in pc \wedge \bar{k} \in q) \vee (\bar{k} \in pc \wedge k \in q)$

1. If $\Sigma, e \Downarrow_{pc} \Sigma', V$ then $q(\Sigma), q(e) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.
2. If $\Sigma, (V_1 \ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V$ then $q(\Sigma), (q(V_1) \ q(V_2)) \Downarrow_{pc \setminus q}^{\text{app}} q(\Sigma'), q(V)$.

The proof is by induction on the derivation of $\Sigma, e \Downarrow_{pc} \Sigma', V$ and the derivation of $\Sigma, (V_1 \ V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V$, and by case analysis on the final rule used in that derivation.

- For case [F-LEVEL], $e = \text{level } k \text{ in } e'$.
By the antecedents of this rule:

$$\begin{array}{c} k' \text{ fresh} \\ \Sigma[k' := \lambda x. \text{true}], e'[k := k'] \Downarrow_{pc} \Sigma', V \end{array}$$

By induction, $q(\Sigma[k' := \lambda x. \text{true}]), q(e'[k := k']) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

Since $k' \notin \Sigma$, we know that $k' \notin q(\Sigma)$.

Therefore, $q(\Sigma)[k' := \lambda x. \text{true}] = q(\Sigma[k' := \lambda x. \text{true}])$.

By α -renaming, we assume $k \notin q, \bar{k} \notin q, k' \notin q$, and $\bar{k}' \notin q$.

Therefore $q(e'[k := k']) = q(e'[k := k'])$.

- For case [F-RESTRICT], $e = \text{restrict}(k, e')$. By the antecedents of this rule:

$$\begin{array}{c} \Sigma, e' \Downarrow_{pc} \Sigma_1, V \\ \Sigma' = \Sigma_1[k := \Sigma_1(k) \wedge_f \langle\langle pc \cup \{k\} ? V : \lambda x. \text{true} \rangle\rangle] \end{array}$$

By induction, $q(\Sigma), q(e') \Downarrow_{pc \setminus q} q(\Sigma_1), q(V)$.

$$\begin{aligned} q(\Sigma') &= q(\Sigma_1[k := \Sigma_1(k) \wedge_f \langle\langle pc \cup \{k\} ? V : \lambda x. \text{true} \rangle\rangle]) \\ &= q(\Sigma_1)[k := q(\Sigma_1(k)) \wedge_f q(\langle\langle pc \cup \{k\} ? V : \lambda x. \text{true} \rangle\rangle)] \\ &= q(\Sigma_1)[k := q(\Sigma_1(k)) \wedge_f \langle\langle pc \cup \{k\} \setminus q ? q(V) : \lambda x. \text{true} \rangle\rangle] \\ &\quad \text{by Lemma 1} \end{aligned}$$

- For case [F-VAL], $e = V$. Since $\Sigma, V \Downarrow_{pc} \Sigma, V$ and $q(\Sigma), q(V) \Downarrow_{pc \setminus q} q(\Sigma), q(V)$, this case holds.
- For case [F-REF], $e = \text{ref } e'$. Then by the antecedents of the [F-REF] rule:

$$\begin{array}{c} \Sigma, e' \Downarrow_{pc} \Sigma'', V' \\ a \notin \text{dom}(\Sigma'') \\ V'' = \langle\langle pc ? V' : 0 \rangle\rangle \\ \Sigma' = \Sigma''[a := V''] \\ V = a \end{array}$$

By induction, $q(\Sigma), q(e') \Downarrow_{pc \setminus q} q(\Sigma''), q(V')$.

Since $a \notin \text{dom}(\Sigma'')$, $a \notin \text{dom}(q(\Sigma''))$.

By Lemma 1, $q(V'') = \langle\langle pc \setminus q ? q(V') : q(0) \rangle\rangle$.

Since $\Sigma' = \Sigma''[a := V'']$, $q(\Sigma') = q(\Sigma'')[a := q(V'')]$.

Therefore $q(\Sigma), \text{ref } q(e') \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- For case [F-DEREF], $e = !e'$. Then by the antecedents of the [F-DEREF] rule:

$$\begin{array}{c} \Sigma, e' \Downarrow_{pc} \Sigma', V' \\ V = \text{deref}(\Sigma', V', pc) \end{array}$$

By induction, $q(\Sigma), q(e') \Downarrow_{pc \setminus q} q(\Sigma'), q(V')$.

By Lemma 2, $q(V) = \text{deref}(q(\Sigma'), q(V'), pc \setminus q)$.

Therefore $q(\Sigma), q(!e') \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- For case [F-ASSIGN], $e = (e_a := e_b)$.

By the antecedents of the [F-ASSIGN] rule:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, V_1 \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V \\ \Sigma' = \text{assign}(\Sigma_2, pc, V_1, V) \end{array}$$

By induction

$$\begin{array}{c} q(\Sigma), q(e_a) \Downarrow_{pc \setminus q} q(\Sigma_1), q(V_1) \\ q(\Sigma_1), q(e_b) \Downarrow_{pc \setminus q} q(\Sigma_2), q(V) \end{array}$$

By Lemma 3, $q(\Sigma') = \text{assign}(q(\Sigma_2), pc \setminus q, q(V_1), q(V))$. Therefore $q(\Sigma), q(e_a := e_b) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- For case [F-APP], $e = (e_a e_b)$. By the antecedents of the [F-APP] rule:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc} \Sigma_1, V_1 \\ \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V_2 \\ \Sigma_2, (V_1 V_2) \Downarrow_{pc}^{\text{app}} \Sigma', V \end{array}$$

By induction

$$\begin{array}{c} q(\Sigma), q(e_a) \Downarrow_{pc \setminus q} q(\Sigma_1), q(V_1) \\ q(\Sigma_1), q(e_b) \Downarrow_{pc \setminus q} q(\Sigma_2), q(V_2) \\ q(\Sigma_2), (q(V_1) q(V_2)) \Downarrow_{pc \setminus q}^{\text{app}} q(\Sigma'), q(V) \end{array}$$

Therefore $q(\Sigma), q(e_a e_b) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- For case [F-LEFT], $e = \langle k ? e_a : e_b \rangle$. By the antecedents of this rule

$$\begin{array}{c} k \in pc \\ \Sigma, e_a \Downarrow_{pc} \Sigma', V \end{array}$$

- If $k \in q$, then $q(\langle k ? e_a : e_b \rangle) = q(e_a)$.
By induction $q(\Sigma), q(e_a) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- Otherwise $k \notin q$ and $\bar{k} \notin q$.
Therefore $q(\langle k ? e_a : e_b \rangle) = \langle k ? q(e_a) : q(e_b) \rangle$.
Since $k \in pc \setminus q$, it holds by induction that

$$q(\Sigma), \langle k ? q(e_a) : q(e_b) \rangle \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$$

- Case [F-RIGHT] holds by a similar argument as [F-LEFT].
- For case [F-SPLIT], $e = \langle k ? e_a : e_b \rangle$. By the antecedents of the [F-SPLIT] rule:

$$\begin{array}{c} \Sigma, e_a \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1 \\ \Sigma_1, e_b \Downarrow_{pc \cup \{\bar{k}\}} \Sigma', V_2 \\ V = \langle k ? V_1 : V_2 \rangle \end{array}$$

- Suppose $k \in q$. Then $q(e) = q(e_a)$ and $q(V_1) = q(V)$.
By induction, $q(\Sigma), q(e_a) \Downarrow_{pc \cup \{k\} \setminus q} q(\Sigma_1), q(V_1)$.
Lemma 4 implies $q(\Sigma_1) = q(\Sigma')$, so this case holds.
- If $\bar{k} \in q$, Then $q(e) = q(e_b)$ and $q(V_2) = q(V)$.
By Lemma 4 we know that $q(\Sigma) = q(\Sigma_1)$.
By induction, $q(\Sigma_1), q(e_b) \Downarrow_{pc \cup \{k\} \setminus q} q(\Sigma'), q(V_2)$.
- If $k \notin q$ and $\bar{k} \notin q$, then by induction

$$\begin{array}{c} q(\Sigma), q(e_a) \Downarrow_{pc \cup \{k\} \setminus q} q(\Sigma_1), q(V_1) \\ q(\Sigma_1), q(e_b) \Downarrow_{pc \cup \{\bar{k}\} \setminus q} q(\Sigma'), q(V_2) \end{array}$$

By Lemma 1, $q(V) = \langle pc \setminus q ? q(V_1) : q(V_2) \rangle$.

- For case [FA-FUN], $V_1 = \lambda x. e'$. By the antecedent of this rule

$$\Sigma, e'[x := V_2] \Downarrow_{pc} \Sigma', V$$

We know that $q(\lambda x. e' V_2) = q(e'[x := V_2])$.

By induction $q(\Sigma), q(e'[x := V_2]) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- Both cases [FA-LEFT] and [FA-RIGHT] hold by a similar argument as [F-LEFT].
- Case [FA-SPLIT] holds by a similar argument as [F-SPLIT].