Faceted Execution of Policy-Agnostic Programs  
Extended Version  
Technical Report #UCSC-SOE-12-18

Thomas H. Austin¹, Jean Yang², Cormac Flanagan¹, 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 describ-  
ing 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 pro-  
gram. 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 \( (\ell \ ? \ v_H : v_L) \), where \( v_H \) is the high-confidence value, \( v_L \) is the low-confidence value, and guard \( \ell \) 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 \(\lambda\text{jeeves}\), which extends the faceted semantics with a declarative policy language. We prove termination-insensitive non-interference and policy compliance for \(\lambda\text{jeeves}\) 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 \(\lambda\)-calculus with references, facets (\((\ell \ ? \ ExpL : \ ExpH))\), 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:

\[
\text{let name: string = level a in } <a \ ? \ "Alice" : \ "Anonymous" >
\]

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:

\[
\text{let name: string = level a in }
\text{restrict a: \(\lambda(c: \text{User}).(c == \text{alice})\ in <a \ ? \ "Alice" : \ "Anonymous" >}
\]
| $x$ | variables               |
| $l$ | level variables        |
| $p, r$ | primitives, records |
| $k$ | $p$ | $r$ | principals |
| $Level$ | $low$ | $high$ | levels |
| $\tau$ | $int$ | $bool$ | $record$ | $\tau \cdot \tau$ | types |

$x ::= p \mid r$

${\tau_2} \rightarrow {\tau_2} \mid \tau \text{ ref} \mid Level$

$Exp ::= x \mid p \mid r \mid k \mid \lambda x : \tau . Exp \mid Exp_1 \ (op) Exp_2$

$\mid \text{if Exp}_1 \text{ then Exp}_2 \text{ else Exp}_f$

$\mid Exp_1 \ Exp_2$

$\mid \text{ref Exp (wrestrict Exp}_p)$

$\mid \text{! Exp} \mid x := \text{Exp} \text{ in Exp}_b$

$\mid \langle \ell \ ? \ Exp_{high} \ : \ Exp_{low} \rangle$

$\mid \text{level } \ell \text{ in Exp}$

$\mid \text{ restrict } \ell \ : \ Exp_p \text{ in Exp}$

$Stmt ::= \text{let } x : \tau = \text{Exp} < k > \text{ in Exp}_b$ | statements |

| $print \ {\{Exp_p\}Exp}$ |

**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 $\neg (c = \text{alice}) \land (a = \text{low})$. 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 facetted 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: User).(c == 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:

```haskell
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:

```haskell
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:

```haskell
let mkPatient (identity : User) (doctor : User) (meds : Medication list) : Patient =
  level np, dp, mp 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:

```haskell
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

<table>
<thead>
<tr>
<th>Identity</th>
<th>Doctor</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨a ? alice : default⟩</td>
<td>⟨e ? erica : default⟩</td>
</tr>
<tr>
<td>⟨b ? bob : default⟩</td>
<td>⟨f ? fred : default⟩</td>
</tr>
<tr>
<td>⟨c ? claire : default⟩</td>
<td>⟨f' ? fred : default⟩</td>
</tr>
</tbody>
</table>

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 the number of patients with a doctor field matching the doctor argument, on the records in Table 1 with doctor = erica.

```haskell
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 `<e ? erica : default>` yields the expression `<e ? erica : default == erica : default == erica >`, which can be simplified to `<e ? true : false >`. 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

```haskell
<e ? if (true) then (accum + 1) else accum : if (false) ... >,
```

which simplifies to `<e ? accum + 1 : accum >`. 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 λjeeves, a simple core language that extends the faceted execution semantics of Austin and Flanagan [2] with a declarative policy language for confidentiality. The λjeeves semantics describes how to
Syntax:

\[ e ::= \begin{array}{ll}
    & \text{Term} \\
    & \text{variable} \\
    & \text{constant} \\
    & \text{abstraction} \\
    & \text{application} \\
    & \text{reference allocation} \\
    & \text{dereference} \\
    & \text{assignment} \\
    & \text{faceted expression} \\
    & \text{level variable declaration} \\
    & \text{policy specification} \\
\end{array} \]

\[ S ::= \begin{array}{ll}
    & \text{Statement} \\
    & \text{let statement} \\
    & \text{print statement} \\
\end{array} \]

\[ c ::= \begin{array}{ll}
    & \text{Constant} \\
    & \text{file handle} \\
    & \text{boolean} \\
    & \text{integer} \\
\end{array} \]

\[ x, y, z \quad \begin{array}{ll}
    & \text{Variable} \\
\end{array} \]

\[ k, l \quad \begin{array}{ll}
    & \text{Label (aka Level Variable)} \\
\end{array} \]

Standard encodings:

\[ \begin{array}{ll}
    \text{true} & \text{def} = \lambda x.\lambda y.x \\
    \text{false} & \text{def} = \lambda x.\lambda y.y \\
    \text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \text{def} = (e_1 (\lambda d.e_2) (\lambda d.e_3)) (\lambda x.x) \\
    \text{if } e_1 \text{ then } e_2 & \text{def} = \text{if } e_1 \text{ then } e_2 \text{ else } 0 \\
    \text{let } x = e_1 \text{ in } e_2 & \text{def} = (\lambda x.e_2) e_1 \\
    e_1 \land f \ e_2 & \text{def} = \lambda x.e_1 x \land e_2 x \\
    e_1 \land e_2 & \text{def} = \text{if } e_1 \text{ then } e_2 \text{ else } \text{false} \\
\end{array} \]

\textbf{Fig. 2:} The source language $\lambda$\text{\text{\textit{je}}vees}
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 $\lambda_{jeeves}$ extends the $\lambda$-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 (restrict($k,e$)), and declaring level variables ($\text{level } k$ in $e$). Additional statements exist for let-statements (let $x = e$ in $S$) and printing output (print $\{ e_1 \} e_2$). Conditional statements are encoded in terms of function application.

In $\lambda_{jeeves}$, 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

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

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, $k$ 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 $\overline{k}$. Therefore $pc$ is a set of branches that never contains both $k$ and $\overline{k}$, since that would reflect influences from both the private and public facet of a value.

The operation $\langle \langle \{ k \} \cup \text{rest} \rangle? V_n : V_o \rangle$ 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.

$$\langle \langle \emptyset \rangle ? V_n : V_o \rangle \overset{\text{def}}{=} V_n$$
$$\langle \langle \{ k \} \cup \text{rest} \rangle ? V_n : V_o \rangle \overset{\text{def}}{=} (k? \langle \langle \text{rest} \rangle ? V_n : V_o \rangle : V_o)$$
$$\langle \langle \{ k \} \cup \text{rest} \rangle ? V_n : V_o \rangle \overset{\text{def}}{=} (k? V_o : \langle \langle \text{rest} \rangle ? V_n : V_o \rangle)$$

For example, $\langle \langle \{ k, l \} ? V_n : V_o \rangle$ returns $\langle k? (l? V_n : V_o) : V_o \rangle$. We occasionally abbreviate $\langle \langle \{ k \} ? V_n : V_o \rangle$ as $\langle k? V_n : V_o \rangle$.

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 $\Sigma$ and program counter label $pc$. It returns a modified store $\Sigma'$ reflecting updates and a value $V$. In Figure 3 we show the evaluation rules, which uses additional runtime syntax (also shown in Figure 3).
Expression Evaluation Rules

- **Runtime Syntax**
  
  \[
  e \in \text{Expr} \quad ::= \quad \ldots \mid a
  \]
  
  \[
  \Sigma \in \text{Store} \quad ::= \quad (\text{Address} \to_p \text{Value}) \cup (\text{Label} \to \text{Value})
  \]
  
  \[
  R \in \text{RawValue} \quad ::= \quad c \mid a \mid (\lambda x. e)
  \]
  
  \[
  a \in \text{Address}
  \]
  
  \[
  V \in \text{Val} \quad ::= \quad R \mid \langle k \mid V_1 : V_2 \rangle
  \]
  
  \[
  h \in \text{Branch} \quad ::= \quad k \mid \overline{k}
  \]
  
  \[
  pc \in \text{PC} \quad ::= \quad 2^{\text{Branch}}
  \]

- **Expression Evaluation Rules**

  \[
  \begin{align*}
  \Sigma, R \psi_{pc} \Sigma, R & \quad \quad [\text{F-VAL}] \\
  \Sigma, e \psi_{pc} \Sigma', V' & \quad \quad [\text{F-ASSIGN}] \\
  a \notin \text{dom}(\Sigma') & \quad \quad [\text{F-REFER}]
  \end{align*}
  \]

  - **P-VAL**
    
    \[
    \begin{align*}
    & \frac{\Sigma, R \psi_{pc} \Sigma, R}{\Sigma, e \psi_{pc} \Sigma', V'} \\
    & \quad \quad \text{if } e = \Sigma, e_1 \psi_{pc} \Sigma, V_1 \\
    & \quad \quad \text{and } \Sigma, e_2 \psi_{pc} \Sigma, V_2 \\
    & \quad \quad \text{and } \Sigma, V_1 \uparrow_{pc} \Sigma', V' \\
    & \quad \quad \text{and } \Sigma, V_2 \uparrow_{pc} \Sigma', V'
    \end{align*}
    \]

  - **P-REFER**
    
    \[
    \begin{align*}
    & \frac{\Sigma, (\text{ref } e) \psi_{pc} \Sigma'[a := V], a}{\Sigma, e \psi_{pc} \Sigma', V'} \\
    & \quad \quad \text{if } e = \text{deref}((\Sigma', V, pc)) \\
    & \quad \quad \text{and } \Sigma, V \text{ is derefable}
    \end{align*}
    \]

  - **P-DEREF**
    
    \[
    \begin{align*}
    & \frac{\Sigma, e \psi_{pc} \Sigma', V'}{\Sigma, \text{level } k \text{ in } e \psi_{pc} \Sigma', V'} \\
    & \quad \quad \text{if } k \text{ is fresh}
    \end{align*}
    \]

  - **P-ASSIGN**
    
    \[
    \begin{align*}
    & \frac{\Sigma, e_1 \psi_{pc} \Sigma_1, V_1}{\Sigma, e_2 \psi_{pc} \Sigma_2, V'} \\
    & \quad \quad \text{if } k \text{ is fresh}
    \end{align*}
    \]

  - **P-LEVEL**
    
    \[
    \begin{align*}
    & \frac{\Sigma[k' := \lambda x. \text{true}], e[k := k'] \psi_{pc} \Sigma', V'}{\Sigma, \text{level } k \text{ in } e \psi_{pc} \Sigma', V'} \\
    & \quad \quad \text{if } k \text{ is fresh}
    \end{align*}
    \]

  - **P-RESTRICT**
    
    \[
    \begin{align*}
    & \frac{\Sigma[k := \Sigma_l(k) \land \langle \langle pc \cup \{k\} \mid \text{V : \lambda x. \text{true}} \rangle \rangle}{\Sigma, \text{restrict}(k, e) \psi_{pc} \Sigma', V'} \\
    & \quad \quad \text{if } k \text{ is fresh}
    \end{align*}
    \]

- **Auxiliary Functions**

  \[
  \begin{align*}
  \text{deref} : \text{Store} \times \text{Val} \times \text{PC} & \to \text{Val} \\
  \text{deref}(\Sigma, a, pc) & = \Sigma(a)
  \end{align*}
  \]

  \[
  \begin{align*}
  & \text{deref}(\Sigma, (k \mid V_1 : V_2), pc) = \begin{cases}
  \text{deref}(\Sigma, V_1, pc) & \text{if } k \notin pc \\
  \text{deref}(\Sigma, V_2, pc) & \text{if } k \in pc \\
  \langle \langle k \mid \text{deref}(\Sigma, V_1, pc) : \text{deref}(\Sigma, V_2, pc) \rangle \rangle & \text{otherwise}
  \end{cases}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{assign} : \text{Store} \times \text{PC} \times \text{Val} \times \text{Val} & \to \text{Store} \\
  \text{assign}(\Sigma, pc, a, V) & = \Sigma[a := \langle pc \mid V : \Sigma(a) \rangle]
  \end{align*}
  \]

  \[
  \begin{align*}
  & \text{assign}(\Sigma, pc, (k \mid V_1 : V_2), V) = \Sigma' \quad \text{where } \Sigma_l = \text{assign}(\Sigma, pc \cup \{k\}, V_1) \\
  & \quad \quad \text{and } \Sigma' = \text{assign}(\Sigma_l, pc \cup \{k\}, V_2)
  \end{align*}
  \]

**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 \( \Sigma' \) 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 \( \Sigma \).)

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 \( \Sigma'' \), 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 (e₁ := e₂) is similar to [f-deref]. It evaluates e₁ to a possibly faceted value V₁ corresponding to an address and e₂ to a value V'. The helper function assignOp(\( \Sigma_2 \), pc, V₁, V') defined in Figure 3 decomposes V₁ into separate addresses, storing the appropriate facets of V' into the returned store \( \Sigma'' \). The changes to the store may come from both V₁ 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 \( \alpha \)-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 (\( k \ ? \ e₁ : e₂ \)) 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 e₁ to V₁ and \( \overline{K} \) for the evaluation of e₂ to V₂, tracking the branch of code being taken. The results of both evaluations are joined together in the operation \( \{ k \ ? \ V₁ : V₂ \} \). As an optimization, only one expression is evaluated if the program counter already contains either \( k \) or \( \overline{K} \), as indicated by the rules [f-left] and [f-right].

Function application (e₁ e₂) is somewhat complex in the presence of faceted values. The rule [f-app] evaluates e₁ to V₁, which should either be a lambda or a faceted value containing lambdas, and evaluates e₂ to the function argument V₂. It then delegates the application (V₁ V₂) to an auxiliary relation defined in Figure 4:

\[ \Sigma', (V₁ V₂) ⟦_\text{app} \Sigma'' \rightarrow V' \]

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

\[
\begin{align*}
\text{[fa-fun]} & : \Sigma, e[x := V] \triangleright_{\text{app}} \Sigma', V' \\
\Sigma, ((\lambda x. e) V) & \triangleright_{\text{app}} \Sigma', V'
\end{align*}
\]

\[
\begin{align*}
\text{[fa-left]} & : k \in \text{pc} \\
\Sigma, (V H V) & \triangleright_{\text{app}} \Sigma', V
\end{align*}
\]

\[
\begin{align*}
\text{[fa-salt]} & : k \notin \text{pc} \\
\Sigma, (V H V) & \triangleright_{\text{app}} (k) \Sigma', V' \\
V' & = \langle k ? V' \rangle
\end{align*}
\]

\[
\begin{align*}
\Sigma, ((k ? V H V) V) & \triangleright_{\text{app}} \Sigma', V'
\end{align*}
\]

### Statement Evaluation Rules

\[
\begin{align*}
\text{[f-let]} & : \Sigma, S \triangleright V p, f : R \\
\Sigma, x := e & \triangleright V p, f : R \\
\Sigma, S & \triangleright V p, f : R
\end{align*}
\]

\[
\begin{align*}
\text{[f-print]} & : \Sigma, e_1 \triangleright \Sigma_1, V_f \\
\Sigma_1, e_2 & \triangleright \Sigma_2, V_c \\
\Sigma_2 & \triangleright \Sigma_3, V_p \\
\Sigma_3 & \triangleright \Sigma_4, V_p \\
\{ k_1 \ldots k_n \} & \text{ includes all labels in } V_f, V_c, V_p \\
pick \text{pc such that } \text{pc}(V_f) = f, \text{pc}(V_c) = R, \text{pc}(V_p) = \text{true}
\end{align*}
\]

\[
\begin{align*}
\Sigma, \text{print} \{ e_1 \} & \triangleright V p, f : R
\end{align*}
\]

### Semantics for Derived Encodings

\[
\begin{align*}
\text{[f-if-true]} & : \Sigma, e_1 \triangleright \Sigma_1, \text{true} \\
\Sigma_1, e_2 & \triangleright \Sigma_2, \Sigma_3, V' \\
\Sigma & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \triangleright \Sigma_4, V'
\end{align*}
\]

\[
\begin{align*}
\text{[f-if-false]} & : \Sigma, e_3 \triangleright \Sigma_1, \text{false} \\
\Sigma_1, e_2 & \triangleright \Sigma_3, \Sigma_4, V' \\
\Sigma & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \triangleright \Sigma_4, V'
\end{align*}
\]

\[
\begin{align*}
\text{[f-if-split]} & : \Sigma, e_1 \triangleright \Sigma_1, (k ? V H V) \\
e_1 & \text{ if } V H \text{ then } e_2 \text{ else } e_3 \\
e_2 & \text{ if } V L \text{ then } e_2 \text{ else } e_3 \\
\Sigma_1, (k ? e_1 \triangleright e_2) & \triangleright \Sigma_3, V' \\
\Sigma_3 & \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \triangleright \Sigma_4, V'
\end{align*}
\]

**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 $\lambda$-jeeves 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 $\Sigma_2$. 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 $\lambda x.\text{false}$ 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.

\[
\text{pc} : \text{Expr} \ (\text{with facets}) \rightarrow \text{Expr} \ (\text{with fewer facets})
\]

\[
\text{pc}(\langle k \ ? \ e_1 : e_2 \rangle) =
\begin{cases}
\text{pc}(e_1) & \text{if } k \in \text{pc}\\
\text{pc}(e_2) & \text{if } \overline{k} \in \text{pc}\\
\text{pc}(\langle k \ ? \ e_1 : e_2 \rangle) & \text{otherwise}\\
\text{pc}(V_1) & \text{if } k \in \text{pc}\\
\text{pc}(V_2) & \text{if } \overline{k} \in \text{pc}\\
\text{pc}(V_1) & \text{if } \text{pc}(V_1) = \text{pc}(V_2)\\
\text{pc}(\langle k \ ? \ V_1 : V_2 \rangle) & \text{otherwise}\\
\end{cases}
\]

\[
\text{pc}(\ldots) = \text{compatible closure}
\]

We extend \( \text{pc} \) to project faceted stores \( \Sigma \in \text{Store} \) into stores with fewer facets.

\[
\text{pc} : \text{Value} \rightarrow \text{Value}
\]

\[
\text{pc}(\Sigma) = \lambda a. \text{pc}(\Sigma(a)) \cup \lambda k. \text{pc}(\Sigma(k))
\]

Thus \( \text{pc} \) projection does not remove policies, it only removes some labels on expressions or values. We say that \( \text{pc}_1 \) and \( \text{pc}_2 \) are \textit{consistent} if

\[
\neg \exists k. (k \in \text{pc}_1 \land \overline{k} \in \text{pc}_2) \lor (\overline{k} \in \text{pc}_1 \land k \in \text{pc}_2)
\]

We note some key lemmas regarding projection.

**Lemma 1** If \( V = \langle \text{pc} \ ? \ V_1 : V_2 \rangle \) then \( \forall q \in \text{PC} \)

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

**Lemma 2** If \( V' = \text{deref}(\Sigma, V, \text{pc}) \) then \( \forall q \in \text{PC} \) where \( q \) is consistent with \( \text{pc} \),

\[
q(V') = \text{deref}(q(\Sigma), q(V), \text{pc} \ \setminus \ q).
\]

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

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

**Lemma 4** Suppose \( \text{pc} \) and \( q \) are not consistent and that either

\[
\Sigma, e \Downarrow_{\text{pc}} \Sigma', V \quad \text{or} \quad \Sigma, (V_1 V_2) \Downarrow_{\text{pc}} \Sigma', V
\]

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 $\overline{k}$ is in $q$).

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

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 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 ($e_1 \sim_{pc} e_2$):

- $(V_1 \sim_{pc} V_2)$ iff $pc(V_1) = pc(V_2)$
- $(\Sigma_1 \sim_{pc} \Sigma_2)$ iff $pc(\Sigma_1) = pc(\Sigma_2)$
- $(e_1 \sim_{pc} e_2)$ iff $pc(e_1) = pc(e_2)$

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:

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

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

**Proof.** By the Projection Theorem:

- $pc(\Sigma_1), pc(e_1) \searrow_{\emptyset} pc(\Sigma'_1), pc(V_1)$
- $pc(\Sigma_2), pc(e_2) \searrow_{\emptyset} pc(\Sigma'_2), pc(V_2)$

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 $i \in 1, 2$:

- $S_i = \text{print} \{ e \} \ C([k ? e_i : e_i])$
- $\emptyset, S_1 \vdash V_{p_1}, f_1 : R_1$
- $\emptyset, S_2 \vdash V_{p_2}, f_2 : R_2$

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

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

$$\emptyset, S_1 \vdash V_{p_1}, f_1 : R_1$$
$$\emptyset, S_2 \vdash V_{p_2}, f_2 : R_2$$

Then by the [F-PRINT] rule

$$\emptyset, e \not\in \emptyset, \Sigma_{11}, V_{f_1}$$
$$\Sigma_{11}, C([k ? e_1 : e_1]) \not\in \emptyset, \Sigma_{12}, V_{c_1}$$
$$e_{p_1} = \Sigma_{12}(k_1) \land_f \ldots \land_f \Sigma_{12}(k_n) \text{ where } \{ k_1 \ldots k_n \}$$
includes all labels in $V_{f_1}$, $V_{c_1}$, $V_{p_1}$
$$\Sigma_{12}, e_{p_1} V_{f_1} \not\in \emptyset, \Sigma_{13}, V_{p_1}$$
$$pc_1(V_{f_1}) = f_1, \ pc_1(V_{c_1}) = R_1, \ pc_1(V_{p_1}) = \text{true}, \text{ so } \overline{c_1} \in pc_1$$

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

$$\emptyset, e \not\in \emptyset, \Sigma_{21}, V_{f_2}$$
$$\Sigma_{21}, C([k ? e_2 : e_1]) \not\in \emptyset, \Sigma_{12}, V_{c_2}$$
$$e_{p_2} = \Sigma_{22}(k_1) \land_f \ldots \land_f \Sigma_{22}(k_n) \text{ where } \{ k_1 \ldots k_n \}$$
includes all labels in $V_{f_2}$, $V_{c_2}$, $V_{p_2}$
$$\Sigma_{22}, e_{p_2} V_{f_2} \not\in \emptyset, \Sigma_{23}, V_{p_2}$$
$$pc_2(V_{f_2}) = f_2, \ pc_2(V_{c_2}) = R_2, \ pc_2(V_{p_2}) = \text{true}, \text{ so } \overline{c_2} \in pc_2$$

By determinism, $\Sigma_{11} = \Sigma_{21}, V_{f_1} = V_{f_2}$.

Also, $C([\overline{c_1} ? e_1 : e_1]) \sim (\overline{c_1}) C([k ? e_2 : e_1])$. Hence by the projection theorem

$$\Sigma_{12} \sim (\overline{c_1}) \Sigma_{22} \ V_{c_1} \sim (\overline{c_1}) V_{c_2} \ e_{p_1} \sim (\overline{c_1}) e_{p_2}$$
$$\Sigma_{13} \sim (\overline{c_1}) \Sigma_{23} \ V_{p_1} \sim (\overline{c_1}) V_{p_2}$$

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

We have implemented Jeeves\textsuperscript* 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].\textsuperscript{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 \( \text{Expr[Int]} \) 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 \( \text{Expr[T]} \Rightarrow \text{Formula} \), where \( \text{Formula} \) 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\textsuperscript* 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\textsuperscript* semantics.

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

\begin{verbatim}
def mkLevel: LevelVar
def restrict (lvar: LevelVar, f: Expr[T] \Rightarrow Formula)
def mkSensitive(lvar: LevelVar, high: Expr[T], low: Expr[T]): Expr[T]
\end{verbatim}

The programmer introduces level variables, which are boolean logic variables mapped to \texttt{HIGH} and \texttt{LOW}, into scope by calling the \texttt{mkLevel} method. The \texttt{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 \texttt{mkSensitive} method, which takes a level variable along with high-confidence and low-confidence views. To support evaluation with sensitive expressions, programs should accommodate values of type \( \text{Expr[T]} \) (e.g. \texttt{IntExpr} rather than \texttt{BigInt}). The library has methods for producing concrete state:

\textsuperscript{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

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 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 \texttt{mkLevel}, \texttt{mkSensitive}, \texttt{restrict}, and \texttt{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.

<table>
<thead>
<tr>
<th>File</th>
<th>Total LOC</th>
<th>Policy LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfUser.scala</td>
<td>212</td>
<td>21</td>
</tr>
<tr>
<td>PaperRecord.scala</td>
<td>304</td>
<td>75</td>
</tr>
<tr>
<td>PaperReview.scala</td>
<td>116</td>
<td>32</td>
</tr>
<tr>
<td>ConfContext.scala</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Backend + Squeryl</td>
<td>800</td>
<td>0</td>
</tr>
<tr>
<td>Frontend (Scalatra)</td>
<td>629</td>
<td>0</td>
</tr>
<tr>
<td>Frontend (SSP)</td>
<td>798</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2865</strong></td>
<td><strong>128</strong></td>
</tr>
</tbody>
</table>

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

The programmer defines a \textit{getter}, a \textit{setter}, and a \textit{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 \texttt{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 show 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 λjeeves 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

A Proof of Projection

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

**Theorem 1.** Suppose \( \Sigma, e \not\vdash_{pc} \Sigma', V \) Then for any \( q \in PC \) where \( pc \) and \( q \) are consistent \( q(\Sigma), q(e) \not\vdash_{pc\setminus q} q(\Sigma'), q(V) \)

**Proof.** We prove a stronger inductive hypothesis, namely that for any \( q \in PC \) where \( \neg\exists k.(k \in pc \land k \in q) \lor (k \in pc \land k \in q) \)

1. If \( \Sigma, e \not\vdash_{pc} \Sigma', V \) then \( q(\Sigma), q(e) \not\vdash_{pc\setminus q} q(\Sigma'), q(V) \).
2. If \( \Sigma, (V_1 V_2) \not\vdash_{pc} \Sigma', V \) then \( q(\Sigma), (q(V_1) q(V_2)) \not\vdash_{pc\setminus q} q(\Sigma'), q(V) \).

The proof is by induction on the derivation of \( \Sigma, e \not\vdash_{pc} \Sigma', V \) and the derivation of \( \Sigma, (V_1 V_2) \not\vdash_{pc} \Sigma', V \), and by case analysis on the final rule used in that derivation.

- For case \([f-level] \), \( e = \text{level} k \) in \( e' \).
  By the antecedents of this rule:
  \[
  k' \text{ fresh} \\
  \Sigma[k' := \lambda x.\text{true}], e'[k := k'] \not\vdash_{pc} \Sigma', V
  \]
  By induction, \( q(\Sigma[k' := \lambda x.\text{true}]), q(e'[k := k']) \not\vdash_{pc\setminus q} q(\Sigma'), q(V) \).
  Since \( k' \not\in \Sigma \), we know that \( k' \not\in q(\Sigma) \).
  Therefore, \( q(\Sigma)[k' := \lambda x.\text{true}] = q(\Sigma[k' := \lambda x.\text{true}]) \).
  By \( \alpha \)-renaming, we assume \( k \not\in q, k \not\in q', k' \not\in q, \) and \( k' \not\in q' \).
  Therefore \( q(e'[k := k']) = q(e'[k := k']) \).

- For case \([f-restrict] \), \( e = \text{restrict}(k, e') \). By the antecedents of this rule:
  \[
  \Sigma, e' \not\vdash_{pc} \Sigma_1, V \\
  \Sigma' = \Sigma_1[k := \Sigma_1(k) \land_f \langle pc \cup \{k\} : V : \lambda x.\text{true} \rangle]
  \]
  By induction, \( q(\Sigma), q(e') \not\vdash_{pc\setminus q} q(\Sigma_1), q(V) \).
  \[
  q(\Sigma') = q(\Sigma_1[k := \Sigma_1(k) \land_f \langle pc \cup \{k\} : V : \lambda x.\text{true} \rangle]) \\
  = q(\Sigma_1[k := q(\Sigma_1(k)) \land_f \langle pc \cup \{k\} : V : \lambda x.\text{true} \rangle]) \\
  = q(\Sigma_1[k := q(\Sigma_1(k)) \land_f \langle pc \cup \{k\} \setminus q : q(V) : \lambda x.\text{true} \rangle])
  \]
  by Lemma 1

- For case \([f-val] \), \( e = V \). Since \( \Sigma, V \not\vdash_{pc} \Sigma, V \) and \( q(\Sigma), q(V) \not\vdash_{pc\setminus q} q(\Sigma), q(V) \), this case holds.

- For case \([f-ref] \), \( e = \text{ref} e' \). Then by the antecedents of the \([f-ref] \) rule:
  \[
  \Sigma, e' \not\vdash_{pc} \Sigma'', V' \\
  a \not\in \text{dom}(\Sigma'') \\
  V'' = \langle pc ? V' : 0 \rangle \\
  \Sigma' = \Sigma''[a := V''] \\
  V = a
  \]
By induction, $q(\Sigma), q(e') \Downarrow_{pc \setminus q} q(\Sigma'), q(V')$.

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

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

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

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

- For case $[p\text{-deref}]$, $e = le'$.
  Then by the antecedents of the $[p\text{-deref}]$ rule:

  $\Sigma, e' \Downarrow_{pc} \Sigma', V' \ni 
  V = \text{deref}(\Sigma', V', pc)$

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 $q(\Sigma), q(le') \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.

- For case $[p\text{-assign}]$, $e = (e_a := e_b)$.
  By the antecedents of the $[p\text{-assign}]$ rule:

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

By induction

$q(\Sigma), q(e_a) \Downarrow_{pc \setminus q} q(\Sigma_1), q(V_1)$

$q(\Sigma_1), q(e_b) \Downarrow_{pc \setminus q} q(\Sigma_2), q(V)$

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 $[p\text{-app}]$, $e = (e_a e_b)$.
  By the antecedents of the $[p\text{-app}]$ rule:

  $\Sigma, e_a \Downarrow_{pc} \Sigma_1, V_1 \ni 
  \Sigma_1, e_b \Downarrow_{pc} \Sigma_2, V_2 \ni 
  \Sigma_2, (V_1 V_2) \Downarrow_{pc} \Sigma', V \ni 
  \Sigma' = \text{app}(\Sigma_2, V_1, V_2)$

By induction

$q(\Sigma), q(e_a) \Downarrow_{pc \setminus q} q(\Sigma_1), q(V_1)$

$q(\Sigma_1), q(e_b) \Downarrow_{pc \setminus q} q(\Sigma_2), q(V_2)$

$q(\Sigma_2), (q(V_1) q(V_2)) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$

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

- For case $[p\text{-left}]$, $e = (k \? e_a : e_b)$.
  By the antecedents of this rule

  $k \in pc \ni 
  \Sigma, e_a \Downarrow_{pc} \Sigma', V \ni 
  \Sigma, e_a \Downarrow_{pc} \Sigma', V \ni 

  \begin{itemize}
  \item If $k \in q$, then $q(k \? e_a : e_b) = q(e_a)$.
    By induction $q(\Sigma), q(e_a) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.
  \item Otherwise $k \notin q$ and $F \notin q$.
    Therefore $q(k \? e_a : e_b) = \langle k \? q(e_a) : q(e_b) \rangle$.
    Since $k \in pc \setminus q$, it holds by induction that
    $q(\Sigma), (k \? q(e_a) : q(e_b)) \Downarrow_{pc \setminus q} q(\Sigma'), q(V)$.
  \end{itemize}
- 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:
  \[
  \Sigma, e_a \downarrow_{pc \cup \{k\}} \Sigma_1, V_1 \\
  \Sigma_1, e_b \downarrow_{pc \cup \{k\}} \Sigma', V_2 \\
  V = \langle k \ ? \ V_1 : V_2 \rangle
  \]

  - 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 $\overline{k} \in q$, then $q(e) = q(e_b)$ and $q(V_2) = q(V)$.
    By Lemma 4 we know that $q(\Sigma_1) = q(\Sigma_1')$.
    By induction, $q(\Sigma_1), q(e_b) \downarrow_{pc \cup \{k\} \setminus q} q(\Sigma'), q(V_2)$.
  - If $k \not\in q$ and $\overline{k} \not\in q$, then by induction
    \[
    q(\Sigma), q(e_a) \downarrow_{pc \cup \{k\} \setminus q} q(\Sigma_1), q(V_1) \\
    q(\Sigma_1), q(e_b) \downarrow_{pc \cup \{k\} \setminus q} q(\Sigma'), q(V_2)
    \]
    By Lemma 1, $q(V) = \langle \langle pc \setminus q \ ? \ q(V_1) \rangle : q(V_2) \rangle$.
- For case [f-a-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 [f-a-left] and [f-a-right] hold by a similar argument as [f-left].
- Case [f-a-split] holds by a similar argument as [f-split].