15-316: Software Foundations of Security and Privacy **Spring 2017**

Lecture 16: Policy-Agnostic Programming

Lecturer: Jean Yang

By now, you should be convinced that if we want security and privacy in our systems, we want *information flow* guarantees. Previously in this class, we discussed how *access control* policies that only check values when they come out of the database are not enough, because today's software applications do many things with data and show the results to all kinds of different viewers. Information flow controls are useful because they provide guarantees that sensitive information is not leaking to unauthorized viewers, through any computations.

We have discussed the following examples of information leaks that information flow controls could prevent:

- The HotCRP conference management system allowing users to preview password remind emails—for any user in the system. Presumably there was an access control check, but for the wrong user!
- The course grading system disallowing students from viewing final grades for courses, but leaking the new grade average. An information flow system could have prevented this leak by ensuring no values computed from sensitive final grade information could leak to unauthorized viewers.
- AirBnb disallowing phone numbers from being displayed in messages from hosts to guests, but neglecting to scrub the number in message preview mode. This is another example of a situation where any missing check can lead to an information leak!

In this class, we have seen two approaches for preventing information leaks:

- **Dynamic tracking with labels.** We looked at the Flume system, which used labels to keep track of information flow at the process level. Processes get labeled based on the sensitive values they use, and guards at each output channel mediate the release of values to prevent unintended information disclosure. While Flume provides a nice way to prevent information leaks in legacy software systems, a drawback is that the granularity of tracking is coarse, and so if there is any information leak in a process, there must be an exception or silent failure. (The burden falls on the programmer to make the program not leak information, if they want a fully working application!) Part of the problem is that information flow is *not* a safety property, and so any reference monitor needs to be conservative.
- **Static type-checking with labels.** We talked about Jif for Java information flow, and looked at a simple type system that preserves non-interference. We showed that type-checking can even help us prevent *implicit flows*. While it is incredible that we are able to prevent information leaks using such a simple type system, this approach has the drawback that it not only requires the programmer to correctly implement a program with respect to information flow, but also to correctly label the program!

Both of these approaches provide *safety nets* that programs will not leak information. We can think of them as akin to tools like `valgrind`, which tells you if your program is leaking memory. But just as `valgrind` will not *fix* your program to not leak memory anymore, with the information flow approaches the programmer still needs to do the heavy lifting of putting checks and alternate cases in the right places.

In today's lecture, we will talk about my work on *policy-agnostic programming*, a programming model that *factors out* information flow policies and manages them for the programmer. The goal of policy-agnostic

programming is to provide correct-by-construction guarantees the way memory-managed language provide guarantees of type safety and memory safety for all programs.

16.1 Where Do Existing Approaches Fall Short?

To get us into the right mood, let's bring back the canonical code snippet we like to use to talk about information flow, where y is a sensitive value:

```
x = 0
if (y == 42):
  x = x+1
```

The goal of information flow controls is to protect y at all costs. In this case, we want to prevent a viewer from seeing the incremented value of x if they are not allowed to see y. With Flume, a process that uses y will not be allowed to send its results anywhere not allowed to see y, so that covers this case. With our information flow type system, type-checking keeps track of a program counter variable **pc** that it uses to keep track of the maximum sensitivity of the condition we are currently evaluating under, for the express purpose of using it to prevent these implicit flows.

More fancy example. To make language-based techniques seem actually useful, let's suppose we are building a social calendar application, and that our calendar supports the following functionality:

- Creation of events. Events have the usual event information, as well as a guest list. The guest list is fancy and can be marked as "finalized," allowing the lists to be private only to the hosts until they determine who they *really* want to invite.
- A "friend" network between users.
- Searching over events based on their fields, including guests. Users can search, for instance, about events their friends are attending at a given time.

Events can be visible to everyone, only to hosts, only to guests of events with finalized guests, or to friends of hosts/guests.

To create somewhat high-stakes situation, let's suppose our TA Sam likes grading so much that we are planning a surprise second midterm next Tuesday evening, to give him even more material to grade. If we use our calendar to plan it, we will want to exclude Sam from the guest list, and make sure that only guests can see the event. Matt and I also don't know how much funding we have for second midterm refreshments, so we may want to take advantage of the "finalized" feature and play with different possible guest lists of not only the students in this course, but also Sam's friends. Now let's see what can go wrong with policy enforcement, and how existing approaches don't quite give us everything.

Checking can only do so much. In order for information about the surprise second midterm to stay secret, the system needs to ensure that sensitive information about the event does not leak to Sam in any way, including through a query Sam might issue about events his friends are attending next Tuesday evening. Missing a check on query code can cause this information to leak. Note that while Flume will raise a dynamic error in such a case, and Jif will reject the program at compile-time, the programmer still needs to make sure all of these checks are in place—otherwise there is either no working application at all, or an application with limited functionality. (Getting a 500 error whenever you try to search is not pleasant!)

Labels are low-level. With these label-based approaches, the programmer needs to potentially translate higher-level policies, for instance "Alice's friends can see her event if they are nearby and it is Wednesday evening," into labels. (Even mapping "friends" policies to labels can get complicated!) They *trust* the programmer with this, meaning that this is susceptible to programmer error! In addition, there is a problem we call *leaky enforcement*: when *policies* depend on sensitive values, but neglect to enforce the policies on those values. For instance, event information for Sam's Second Midterm should be visible to the guest list, but the *guest list itself* should only be visible to people on the list after it has been finalized. If the programmer neglects to capture this dependency, we could have the following awkward chain of events:

1. Matt and I add Jordan to the guest list, because even though he is auditing, he should provide good grading material for Sam.
2. Somebody forgot the policy dependency, and the event shows up on Jordan's calendar before we finalize the guest list.
3. Matt and I learn that pizza prices have gone up, and we have to remove Jordan.
4. Jordan notices the event has been removed from his calendar, and is both suspicious and sad.

You might think this is silly and not that important, but there was an analogous bug in the HotCRP conference management system that allowed people to infer whether papers were accepted based on the presence of another field, and companies in the Real World struggle with this problem too (in a way that is too confidential to discuss in these lecture notes). The solution is to have a higher-level policy language than labels. We can think of labels as an *assembly* for information flow policies.

16.2 Policy-Agnostic Programming: Core Tenets

The goal of policy-agnostic programming is to factor out security and privacy policies from other program functionality. There are three core tenets:

1. We want to specify each policy only once, instead of as conditional checks and cases across the entire program.
2. We want the program to do something more interesting—and controllable by the programmer—than raising an exception or silently failing when the program violates a policy.
3. We want intuitive policies!

Because information flow is so deeply intertwined with the program, doing this involves having a technique that changes execution semantics everywhere! We will need something that helps us track sensitive values as they flow through a computation, a way to specify what should happen if policies fail, and higher-level policies than labels!

16.3 Faceted Execution for Information Flow

For the first of our two desired properties, we can look to an execution mechanism called *faceted execution*

Faceted values take the form

$$\langle k \ ? \ V_{\text{H}} : V_{\text{L}} \rangle$$

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 true according to the policy associated with k. When the policy specifies false, the observed value should instead be 0.

16.3.1 Computing with Faceted Values

Computations on faceted values yield new faceted values. Here is an example of the evaluation of addition over faceted values:

$$\begin{aligned} &\langle j \ ? \ 1 : 2 \rangle + \langle k \ ? \ 3 : 4 \rangle \\ \rightarrow &\langle j \ ? \ \langle k \ ? \ 4 : 5 \rangle : \langle k \ ? \ 5 : 6 \rangle \rangle \end{aligned}$$

Faceted execution is also able to handle implicit flows. First, conditional checks involving faceted values become faceted conditionals:

$$\begin{aligned} &\textbf{if } (\langle j \ ? \ 1 : 2 \rangle = 2) \textbf{ then } x := x + 1 \\ \rightarrow &\textbf{if } (\langle j \ ? \ \textbf{false} : \textbf{true} \rangle) \textbf{ then } x := x + 1 \end{aligned}$$

The system then keeps track of path assumptions using a program counter set of variables pc and updates x to be $\langle j \ ? \ x_{old} : x_{old} + 1 \rangle$.

16.3.2 Policies and Facets

Faceted execution doesn't help us with the problem of labels being low-level, as it assumes some oracle tells us what the label values are supposed to be. With faceted execution alone, we still need policy code across the program. The reasons are the same as the motivations for information flow. First, we may not know who the viewer is going to be right away. Second, the viewer might be computed from the program.

The solution is to allow users to associate labels with *policies* that are a function of the output context, and to keep track of the policies each label maps to. We will not get too much into the policies for this lecture, but the important things are that 1) policies can depend on sensitive values, and 2) the viewer may be computed using sensitive values. Note that this introduces the potential for circular dependencies. We account for this in our semantics and guarantees.

16.4 λ^{jeeves} Semantics

We now describe the semantics of faceted execution for policy-agnostic programming.

16.4.1 Core Semantics

To describe faceted execution for policy-agnostic programming we introduce λ^{jeeves}, a simple core language that combines a faceted execution semantics with a declarative policy language for confidentiality. We show the source syntax in Figure 16.1. The language λ^{jeeves} extends the λ-calculus with expressions for allocating references (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 labels (label k in e). Additional statements exist for let-statements (let $x = e$ in S) and printing output (print $\{e_1\}$ e_2). Conditionals are encoded in terms of function application.

Figure 16.1: The source language λ^{jeeves}

Syntax:

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

Standard encodings:

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

Figure 16.2: Runtime syntax for λ^{jeeves}evalution.

Runtime Syntax

$$
\begin{array}{rcll}
e & \in & \mathit{Expr} & ::= \quad ... \mid a \\
\Sigma & \in & \mathit{Store} & = \quad (\mathit{Address} \rightarrow_p \mathit{Value}) \cup (\mathit{Label} \rightarrow \mathit{Value}) \\
R & \in & \mathit{RawValue} & ::= \quad c \mid a \mid (\lambda x.e) \\
a & \in & \mathit{Address} & \\
V & \in & \mathit{Val} & ::= \quad R \mid \langle k \ ?\ V_1 : V_2 \rangle \\
h & \in & \mathit{Branch} & ::= \quad k \mid \overline{k} \\
pc & \in & \mathit{PC} & = \quad 2^{\mathit{Branch}}
\end{array}
$$

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

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

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, $\overline{k}$ 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 label 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 $\langle\!\langle\, pc \ ?\ V_1 : V_2 \,\rangle\!\rangle$ 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. For example, $\langle\!\langle\, \{k, l\} \ ?\ V_{\text{H}} : V_{\text{L}} \,\rangle\!\rangle$ returns $\langle k \ ?\ \langle l \ ?\ V_{\text{H}} : V_{\text{L}} \rangle : V_{\text{L}} \rangle$. We occasionally abbreviate $\langle\!\langle\, \{k\} \ ?\ V_{\text{H}} : V_{\text{L}} \,\rangle\!\rangle$ as $\langle\!\langle\, k \ ?\ V_{\text{H}} : V_{\text{L}} \,\rangle\!\rangle$.

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. We show the runtime syntax in Figure 16.2, the evaluation rules in Figure 16.3, and the auxiliary functions for dereference and assignment in Figure 16.4. In class we covered a subset of the rules; we show the full rules here.

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 $\mathit{deref}(\Sigma', V, pc)$ (defined in Figure 16.4), 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 $\mathit{assign}(\Sigma_2, pc, V_1, V')$ defined in Figure 16.4 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-LABEL] dynamically allocates a label (label k in e), adding a fresh label to the store with the

Expression Evaluation Rules $\boxed{\Sigma, e \Downarrow_{pc} \Sigma', V}$

$$\frac{}{\Sigma, R \Downarrow_{pc} \Sigma, R} \quad [\text{F-VAL}]$$

$$\frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma', V' \\ a \notin dom(\Sigma') \\ V = \langle\!\langle\, pc\ ?\ V' : 0 \,\rangle\!\rangle\end{array}}{\Sigma, (\text{ref } e) \Downarrow_{pc} \Sigma'[a := V], a} \quad [\text{F-REF}] \qquad \frac{\begin{array}{c}\Sigma, e \Downarrow_{pc} \Sigma', V \\ V' = deref(\Sigma', V, pc)\end{array}}{\Sigma, !e \Downarrow_{pc} \Sigma', V'} \quad [\text{F-DEREF}]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1 \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V' \\ \Sigma' = assign(\Sigma_2, pc, V_1, V')\end{array}}{\Sigma, e_1 := e_2 \Downarrow_{pc} \Sigma', V'} \quad [\text{F-ASSIGN}]$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, V_1 \\ \Sigma_1, e_2 \Downarrow_{pc} \Sigma_2, V_2 \\ \Sigma_2, (V_1\ V_2) \Downarrow_{pc}^{\texttt{app}} \Sigma', V'\end{array}}{\Sigma, (e_1\ e_2) \Downarrow_{pc} \Sigma', V'} \quad [\text{F-APP}]$$

$$\frac{\begin{array}{c}k \notin pc \ \text{ and } \overline{k} \notin pc \\ \Sigma, e_1 \Downarrow_{pc \cup \{k\}} \Sigma_1, V_1 \\ \Sigma_1, e_2 \Downarrow_{pc \cup \{\overline{k}\}} \Sigma', V_2 \\ V' = \langle\!\langle\, k\ ?\ V_1 : V_2 \,\rangle\!\rangle\end{array}}{\Sigma, \langle k\ ?\ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V'} \quad [\text{F-SPLIT}]$$

$$\frac{k \in pc \qquad \Sigma, e_1 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k\ ?\ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad [\text{F-LEFT}]$$

$$\frac{\overline{k} \in pc \qquad \Sigma, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \langle k\ ?\ e_1 : e_2 \rangle \Downarrow_{pc} \Sigma', V} \quad [\text{F-RIGHT}]$$

$$\frac{\begin{array}{c}k' fresh \\ \Sigma[k' := \lambda x.true], e[k := k'] \Downarrow_{pc} \Sigma', V\end{array}}{\Sigma, \text{label } k \text{ in } e \Downarrow_{pc} \Sigma', V'} \quad [\text{F-LABEL}]$$

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

Figure 16.3: λ^{jeeves} expression evaluation.

Auxiliary Functions

$$
\begin{array}{lcl}
\mathit{deref} : \mathit{Store} \times \mathit{Val} \times \mathit{PC} & \rightarrow & \mathit{Val} \\
\mathit{deref}(\Sigma, a, pc) & = & \Sigma(a) \\
\mathit{deref}(\Sigma, \langle k \ ?\ V_{\text{H}} : V_{\text{L}} \rangle, pc) & = & \begin{cases} \mathit{deref}(\Sigma, V_{\text{H}}, pc) & \text{if } k \in pc \\ \mathit{deref}(\Sigma, V_{\text{L}}, pc) & \text{if } \overline{k} \in pc \\ \langle\!\langle\, k \ ?\ \mathit{deref}(\Sigma, V_{\text{H}}, pc) : \mathit{deref}(\Sigma, V_{\text{L}}, pc) \,\rangle\!\rangle & \text{o/w} \end{cases}
\end{array}
$$

$$
\begin{array}{lcll}
\mathit{assign} : \mathit{Store} \times \mathit{PC} \times \mathit{Val} \times \mathit{Val} & \rightarrow & \mathit{Store} & \\
\mathit{assign}(\Sigma, pc, a, V) & = & \Sigma[a := \langle\!\langle\, pc \ ?\ V : \Sigma(a) \,\rangle\!\rangle] & \\
\mathit{assign}(\Sigma, pc, \langle k \ ?\ V_{\text{H}} : V_{\text{L}} \rangle, V) & = & \Sigma' & \text{where } \Sigma_1 = \mathit{assign}(\Sigma,\ \ pc \cup \{k\}, V_{\text{H}}, V) \\
 & & & \text{and} \quad \Sigma' = \mathit{assign}(\Sigma_1, pc \cup \{\overline{k}\}, V_{\text{L}}, V)
\end{array}
$$

Figure 16.4: Auxiliary functions for λ^{jeeves}evaluation.

default policy of $\lambda x.true$. 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 $\overline{k}$ 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 $\overline{k}$, 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 16.5:

$$\Sigma, (V_1\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V'$$

This relation breaks apart faceted values and tracks the influences of the labels 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.

Conditional branches (if e_1 then e_2 else e_3) are Church-encoded as function calls for the sake of simplicity. However, Figure 16.6 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 λ^{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:

$$\Sigma, S \Downarrow V_p, f : R$$

The rules for statements are specified in Figure 16.5. 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

Figure 16.5: Faceted evaluation semantics for application and statements.

Application Rules $\boxed{\Sigma, (V_1\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V'}$

[FA-FUN]

$$\frac{\Sigma, e[x := V] \Downarrow_{pc} \Sigma', V'}{\Sigma, ((\lambda x.e)\ V) \Downarrow^{\texttt{app}}_{pc} \Sigma', V'}$$

[FA-LEFT]

$$\frac{k \in pc \qquad \Sigma, (V_{\text{H}}\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V}{\Sigma, (\langle k\ ?\ V_{\text{H}} : V_{\text{L}} \rangle\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V}$$

[FA-SPLIT]

$$\frac{\begin{array}{c} k \notin pc \qquad \overline{k} \notin pc \\ \Sigma, (V_{\text{H}}\ V_2) \Downarrow^{\texttt{app}}_{pc \cup \{k\}} \Sigma_1, V'_{\text{H}} \\ \Sigma_1, (V_{\text{L}}\ V_2) \Downarrow^{\texttt{app}}_{pc \cup \{\overline{k}\}} \Sigma', V'_{\text{L}} \\ V' = \langle\!\langle\, k\ ?\ V'_{\text{H}} : V'_{\text{L}} \,\rangle\!\rangle \end{array}}{\Sigma, (\langle k\ ?\ V_{\text{H}} : V_{\text{L}} \rangle\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V'}$$

[FA-RIGHT]

$$\frac{\overline{k} \in pc \qquad \Sigma, (V_{\text{L}}\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V}{\Sigma, (\langle k\ ?\ V_{\text{H}} : V_{\text{L}} \rangle\ V_2) \Downarrow^{\texttt{app}}_{pc} \Sigma', V}$$

Statement Evaluation Rules $\boxed{\Sigma, S \Downarrow V_p, f : R}$

[F-LET]

$$\frac{\Sigma, e \Downarrow_{\emptyset} \Sigma', V \qquad \Sigma, S[x := V] \Downarrow V_p, f : R}{\Sigma, \text{let } x = e \text{ in } S \Downarrow V_p, f : R}$$

[F-PRINT]

$$\frac{\begin{array}{c} \Sigma, e_1 \Downarrow_{\emptyset} \Sigma_1, V_f \\ \Sigma_1, e_2 \Downarrow_{\emptyset} \Sigma_2, V_c \\ \{\ k_1\ ...\ k_n\ \} = \mathit{closeK}(\mathit{labels}(e_1)\ \cup\ \mathit{labels}(e_2), \Sigma_2) \\ e_p = \lambda x.\mathit{true}\ \wedge_f\ \Sigma_2(k_1)\ \wedge_f\ ...\ \wedge_f\ \Sigma_2(k_n) \\ \Sigma_2, e_p\ V_f \Downarrow_{\emptyset} \Sigma_3, V_p \\ \text{pick } pc \text{ such that } pc(V_f) = f, pc(V_c) = R,\ pc(V_p) = \mathit{true} \end{array}}{\Sigma, \text{print } \{e_1\}\ e_2 \Downarrow V_p, f : R}$$

Semantics for Derived Encodings

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \mathit{true} \quad \Sigma_1, e_2 \Downarrow_{pc} \Sigma', V}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V} \text{[F-IF-TRUE]}$$

$$\frac{\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \mathit{false} \quad \Sigma_1, e_3 \Downarrow_{pc} \Sigma', V}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V} \text{[F-IF-FALSE]}$$

$$\frac{\begin{array}{c}\Sigma, e_1 \Downarrow_{pc} \Sigma_1, \langle k \ ?\ V_H : V_L \rangle \\ e_H = \text{if } V_H \text{ then } e_2 \text{ else } e_3 \\ e_L = \text{if } V_L \text{ then } e_2 \text{ else } e_3 \\ \Sigma_1, \langle k \ ?\ e_H : e_L \rangle \Downarrow_{pc} \Sigma', V\end{array}}{\Sigma, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow_{pc} \Sigma', V} \text{[F-IF-SPLIT]}$$

Auxiliary Functions

$$\begin{array}{rcl} \mathit{closeK}(K, \Sigma) & = & \text{let } K' = \bigcup_{k \in K} \mathit{labels}(\Sigma(k)) \text{ in} \\ & & \text{if } K' = K \\ & & \quad \text{then } K \\ & & \quad \text{else } \mathit{closeK}(K', \Sigma) \end{array}$$

Figure 16.6: Semantics for derived encodings.

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. We determine the set of relevant labels through the *closeK* function, which is then used to construct e_p from the relevant policies in the store Σ_2. e_p 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 false, which may always be displayed, thereby ensuring that there is always at least one valid choice for *pc*.

16.4.2 Properties

We have the guarantees that a single execution with faceted values is equivalent to multiple different executions without faceted values and that the system cannot leak sensitive information either via the output *or* by the choice of the output channel, even if policies depend on sensitive values.

Policy-agnostic programming with faceted execution yields *modified non-interference guarantee*. Classical non-interference states that if a viewer is not authorized to see some sensitive value, then the output should be equivalent no matter what that value is. In this programming model, there is a complication that results from the fact that the runtime is in charge of computing labels, and *labels may be computed from sensitive values*! For instance, a viewer may be able to see a sensitive guest list only if they are a member of the list (and it is finalized).

To handle this, we need to modify the non-interference theorem to take into account the fact that labels may themselves depend on sensitive values. We do this by partitioning the space of possible high-confidentiality values into those that require a given label to be low-confidentiality, and those that do not. What the non-interference theorem says is that the first kind of high-confidentiality value should not affect low-confidentiality outputs. If a viewer is not allowed to see the sensitive guest list, then they should not be able to tell that list apart from any other sensitive guest list they are not allowed to see.

16.4.3 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.

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

We extend pc to project faceted stores $\Sigma \in \mathit{Store}$ into stores with fewer facets. We say that pc_1 and pc_2 are *consistent* if $\neg\exists k.\ (k \in pc_1 \wedge \overline{k} \in pc_2)\ \vee\ (\overline{k} \in pc_1 \wedge k \in pc_2)$. The following theorem shows how a single faceted evaluation simulates (or projects) to multiple executions, each with fewer facets, or possibly with no facets at all.

Theorem 1 (Projection Theorem) *Suppose*

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

Then for any $q \in PC$ where pc and q are consistent

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

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

$$\Sigma_1, e_1 \Downarrow_{\emptyset} \Sigma_1', V_1 \qquad \Sigma_2, e_2 \Downarrow_{\emptyset} \Sigma_2', V_2$$

Then $\Sigma_1' \sim_{pc} \Sigma_2'$ and $V_1 \sim_{pc} V_2$.

16.4.5 Termination-Insensitive Policy Compliance

We also 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.

Before we formally prove this property, we introduce the notion of *k-security*. A program S is *k-secure* if it terminates and its computed policy never permits revealing k-sensitive data.

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

$$S_i = \mathit{print}\ \{e\}\ C[\langle k\ ?\ e_i : e_l \rangle]$$

where each S_i is k-secure. Then
$\{\ f : R \mid \exists V.\ \emptyset, S_1 \Downarrow V, f : R\ \} = \{\ f : R \mid \exists V.\ \emptyset, S_2 \Downarrow V, f : R\ \}$.

16.5 How These Semantics Have Been Used

We initially implemented these semantics for the Jeeves programming language, as embedded domain-specific languages in Scala [1] and Python [2]. We also extended the Python version to include SQL databases. We developed a corresponding semantics and extended the guarantees. The strategy for performing faceted execution in databases involves storing a faceted value in multiple rows, and using meta-data columns to track label assignments. We are currently working on a static, type-driven approach for policy-agnostic programming based on the dynamic faceted execution semantics.

References

[1] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama. Faceted execution of policy-agnostic programs. page 15, 2013.

[2] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong. Precise, dynamic information flow for database-backed applications. In *PLDI*, 2016.