Racets: Faceted Execution in Racket

KRISTOPHER MICINSKI, Haverford College, USA

ZHANPENG WANG, Haverford College, USA

THOMAS GILRAY, University of Alabama, Birmingham, USA

Faceted Execution is a linguistic paradigm for dynamic information-flow control. Under faceted execution, secure program data is represented by *faceted* values: decision trees that encode how the data should appear to its owner (represented by a label) versus everyone else. When labels are allowed to be first-class (i.e., predicates that decide at runtime which data to reveal), faceted execution enables *policy-agnostic programming*: a programming style that allows privacy policies for data to be enforced independently of code that computes on that data.

To date, implementations of faceted execution are relatively heavyweight: requiring either changing the language runtime or the application code (e.g., by using monads). Following Racket's languages-as-libraries approach, we present Racets: an implementation of faceted execution as a library of macros. Given Racket's highly-expressive macro system, our implementation follows relatively directly from the semantics of faceted execution. To demonstrate how Racets can be used for policy-agnostic programming, we use it to build a web-based game of Battleship. Our implementation sheds light on several interesting issues in interacting with code written without faceted execution. Our Racets implementation is open source, under development, and available online.

Additional Key Words and Phrases: security, faceted execution, macros, information flow, languages as libraries

1 INTRODUCTION

As information systems become more interconnected and complex, they consume an ever-growing amount of private data. System designers communicate to users how their data may be used via a *privacy policy*. Unfortunately, implementing such policies correctly is challenging: users often have partial control over the policy (e.g., whether their phone number is publicly visible or private) and policies can change frequently. Not only can specific privacy policies be highly dynamic (dependent on runtime values), but the process of improving privacy policies can be highly dynamic across time. As policies evolve, developers face massive (re)engineering efforts to ensure that implementations continue to match the policy at every relevant point in the codebase.

Policy-agnostic programming is a linguistic paradigm that decouples the implementation of privacy policies from the code that operates on sensitive data. This frees developers to write programs mostly as they would for insecure code, without inserting specific logic to manage information-flow policies directly into application code. Instead, data is labeled with its policy as it enters the system and such labels propagate through the program,

alongside data, as computation progresses. When a secure value needs to be introspected upon (or propagates outside the application), its policy can be invoked at this point dynamically. This paradigm aims to permit code manipulating sensitive data to be written in a manner entirely orthogonal to policies themselves.

Faceted execution (FE) is a highly expressive language semantics enabling policy-agnostic programming [1]. In FE, dynamic information-flow monitors instrument the program, encoding sensitive values as *faceted values*: decision trees specifying different views of data according to different possible security labels. For example, the faceted value $\langle$ *Alice* ? #t ⋄ #f $\rangle$ represents a value that should appear to Alice as #t and to everyone except Alice as #f. Faceted execution propagates distinct facets of a value by extending core linguistic primitives (such as function application). For example, consider the application (x #t) where x is $\langle$ *Alice* ? λx. #t ⋄ **not** $\rangle$. Racket's standard function application will fail here because Racket's #%app expects a procedure rather than a facet. Instead, the proper way to interpret function application on faceted values is to distribute the application over all (in this case, both) facets, producing $\langle$ *Alice* ? #t ⋄ #f $\rangle$: if Alice is viewing, the application yielded true, otherwise it yielded false, so both are computed until the value is explicitly observed with specified permissions. Many other core forms (such as **if**, **set!**, etc...) require similar changes to handle faceted values correctly.

Policy-agnostic programming promotes the idea that programmers should be able to write programs "normally", without concerning themselves with how privacy policies are enforced. Unfortunately, the relatively foundational linguistic changes required to enable faceted execution have hindered implementations thusfar. Dynamic generation of first-class security labels, tracking an arbitrary number of facets per value, and keeping faceted-value trees in a canonical order, are all central challenges in any practical implementation. For example, the first implementation of FE (by Austin and Flanagan [1]) extended a JavaScript interpreter to account for faceted values. Other implementations use monads [19] or rely upon third-party macro systems [22]. We know of no existing implementation of FE that aims to interoperate seamlessly with code written in the host language. By contrast, Scheme boasts a powerful hygienic macro system that allows essentially any linguistic form to be modified arbitrarily.

In this paper we present Racets, an implementation of policy-agnostic programming in Racket via macros [9, 13]. Racets provides facilities for creating policies and faceting secure data with those policies. Racets also extends several core forms in Racket to work with faceted values (our implementation is detailed in Section 4). We have used Racets to implement a small server-based board-game (detailed in Sections 2 and 4). Relevant related work is presented in section 6. We see Racets as a promising prototype for policy-agnostic programming in Racket, and conclude with discussion of future directions in Section 7.

2 OVERVIEW OF FACETED EXECUTION

To introduce faceted execution more concretely, we present the implementation of Battleship, a small guessing game, in Racets (this section presents a distilled version of our case study in Section 4). In this game each player has a private board of coordinates, at which they place "ships". The players hide their boards from each other as play progresses in rounds. Each turn a player guesses the position of a ship on the other player's board. If the guess is successful the tile is removed from the board and a hit is declared publicly. Play ends once one player's board has no remaining tiles, at which point that player loses.

We implement game boards as lists of cons cells representing the (x, y) coordinates of ships. Board creation simply returns an empty list, and adding a piece is done via **cons**:

```
1  (define (makeboard) '())
2  (define (add-piece board x y) (cons (cons x y) board))
```

Next we define mark-hit, which takes a player's board and removes a piece if the guessed coordinate is present. We return a pair of the updated board and a boolean indicating whether the guess was a hit:

```
3  (define (mark-hit board x y)
```

```
4      (if (null? board)
5          (cons board #f)
6          (let* ([fst (car board)]
7                 [rst (cdr board)])
8            (if (and (= (car fst) x)
9                     (= (cdr fst) y))
10               (cons rst #t)
11               (let ([rst+b (mark-hit rst x y)])
12                 (cons (cons fst
13                             (car rst+b))
14                       (cdr rst+b)))))))
```

Although `mark-hit` will operate on sensitive data (the game boards), it is written without any special machinery to maintain the secrecy of `board`. Protecting data w.r.t. policies is instead handled automatically and implicitly by a runtime monitor. When Alice and Bob want to play a game, they both create a *label* to protect their data. A label is unique id mapped to a policy predicate that takes a key (e.g., the current user's name) and returns true or false to indicate permission for the label. Alice's label is used to annotate the data she wants to be kept secret. Supposing Alice chooses to be player 1, she may use the following label:

```
15      (define alice-label (let-label l (λ (x) (= 1 x))) l)
```

This code illustrates label creation, policy predicates, and the first-class nature of labels. The policy predicate (λ (x) (= 1 x)) grants permission to player 1 only and is associated with the dynamically generated label l (returned and bound to `alice-label`). Bob would use a similar policy (but for player 2 instead of 1). At runtime, the **let-label** form creates a label ℓ_A and binds it to a closure for its policy predicate. When Alice wants to protect a value, she creates a facet annotated with her label and two *branches*. The positive (left) branch represents the value as it should appear to her, and the negative (right) to everyone else:

```
16      (define alice-board
17              (facet alice-label (add-pieces (makeboard) x₁ y₁ ...) (★)))
```

In the above example, ★ (lazy failure) is used in the negative branch to ensure execution will fail if Bob tries to observe Alice's secret gameboard. To observe Alice's gameboard, Bob can try to use (**obs** e_ℓ e_{key} e_{fac}) form, which takes a label, a key, and a faceted value. Explicit observation projects a single label e_ℓ in faceted value e_{fac} to either its positive or negative facet, depending on whether the policy associated with e_ℓ returns true for key e_{key}. If Bob tries to observe Alice's board, the policy predicate will return false (from (= 1 2)) and Alice's negative facet ★ will result.

In other applications, Alice may choose a sensible default value to reveal to others—she may even want to create a *nested* facet. For example, a social-networking application may use a nested facet consisting of two labels for $\ell_{Friends}$ and ℓ_{Family}. A user can then present three views of her social-media profile: p_1 to her family, containing her phone number and other contact information, p_2 to her friends showing her interests, and p_3 to everyone else, showing only her name and email.

$$\ell_{Family}$$
$$p_1 \quad \ell_{Friends}$$
$$p_2 \quad p_3$$
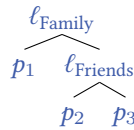
As a game of Battleship progresses, Alice and Bob make guesses in turn, and driver code calls the function `mark-hit` with each of their respective (faceted) game boards to record the attack. However, because Alice and Bob's game boards are both faceted values, `mark-hit` cannot be directly applied as in normal execution. Instead, faceted execution "splits" the evaluation of the function application over both facets, running it first on the

$$
\begin{array}{rcll}
c & \in & const & ::= \quad '() \mid \#\text{t} \mid \#\text{f} \mid \dots \\
x & \in & var & ::= \quad \langle \textit{program variables} \rangle \\
e & \in & exp & ::= \quad c \mid x \\
& & & \mid \quad (\lambda\ (x)\ e) \mid (e\ e) \\
& & & \mid \quad (\textbf{box}\ e) \mid (\textbf{unbox}\ e) \mid (\textbf{set!}\ e\ e) \\
& & & \mid \quad (\textbf{let-label}\ x\ e\ e) \\
& & & \mid \quad (\textbf{facet}\ e\ e\ e) \\
& & & \mid \quad (\textbf{obs}\ e\ e\ e)
\end{array}
$$

Fig. 1. Syntax of λ_{FE}.

positive branch, then again on the negative branch. Finally, the results of each branch are combined again to produce a new faceted value. This allows FE to avoid needing to reason about labels and policies until an explicit observation point where a policy is checked and a faceted value is projected to one of its facets.

Because the applied function can be stateful, faceted execution also maintains the current privilege level in a *program counter* (PC). The program counter is a property of the current evaluation context and is used to build facets when writes are made to the store in a privileged context. For example, if a stateful function "splits" when applied on both the positive and negative facets of a value faceted by a label ℓ, and on the positive branch the function uses set! to mutate a variable x from 2 to 3, FE semantics will set x to $\langle \ell\ ?\ 3 \diamond 2 \rangle$ so that the value 3 cannot be leaked from the secure context (speculative execution under the $+\ell$ facet). This is because, for the duration of the app "split", the evaluation context records that all values are implicitly guarded by $+\ell$ and then $-\ell$, respectively. If the semantics for set! does not make this faceting explicit, a sensitive value can leak from one PC to another. We expand upon these subtleties in Section 3, where we present a complete semantics for faceted execution.

After making various moves, we eventually want to reveal the game boards, pulling the positive view out of `alice-board` to display Alice's board. To do this, we must *observe* the facet with an **obs** form. Because Alice's board is faceted with `alice-label`, we specify that we want to observe `alice-label` and pass in an argument to that label showing that Alice is indeed the person observing the facet:

```
18   (obs alice-label 1 alice-board) ; Returns Alice's board
```

3 A FORMAL SEMANTICS FOR FACETED EXECUTION

We now present a semantics for a core language (λ_{FE}) which includes facets. Our presentation largely mirrors that of Austin et al. [3]. The syntax of our language—reminiscent of Scheme—is shown in Figure 1. λ_{FE} extends the lambda calculus with references (which have interactions with facets in a subtle way) and three forms unique to faceted execution: facet construction, label creation, and facet observation.

Our semantics is shown in Figure 2 [15]. As λ_{FE} is an extension of the lambda calculus with references, we present the parts unique to faceted execution in red, while keeping the lambda calculus with references in blue. Base values in our semantics include addresses (used for boxes), constants, and closures. We also include a kind of lazy failure ($\star$), which is necessary for defining store update within a protected context.

Values in our semantics are either (unfaceted) base values or facets composed of a label and two branches. Facets can nest, allowing trees of faceted values. We use the term *branches* to refer to positive or negated labels. Collections of branches define the program counter pc, which tracks the set of branches in the current context. For example, to apply a faceted function to a value (as in the application of $\langle \ell\ ?\ \lambda x.\ 0 \diamond \lambda x.\ 1 \rangle$), the semantics first applies $\lambda x.\ 0$ while extending pc with $+\ell$, then applies the negative branch extending pc with $-\ell$.

$$\alpha \in \quad addr \quad = \quad \ldots$$
$$bv \in \quad base\text{-}val \quad ::= \quad c \mid \alpha \mid \langle \lambda x.\, e, \rho \rangle \mid \star$$
$$v \in \quad faceted\text{-}val \quad ::= \quad bv \mid \langle\, \alpha \,?\, v \diamond v\, \rangle$$

$$b \in \quad branch \quad ::= \quad +\ell \mid -\ell$$
$$pc \in \quad PC \quad = \quad \wp(branch)$$
$$\rho \in \quad env \quad = \quad var \rightharpoonup v$$
$$\sigma \in \quad store \quad = \quad addr \rightharpoonup v$$

(Expression Evaluation) $\boxed{e, \rho, \sigma \Downarrow_{pc}^{E} \sigma, v}$

APPLY
$$\frac{e_1, \rho, \sigma \Downarrow_{pc}^{E} \sigma', v_1 \qquad e_2, \rho, \sigma' \Downarrow_{pc}^{E} \sigma'', v_2 \qquad (v_1\ v_2), \rho, \sigma'' \Downarrow_{pc}^{A} \sigma''', v'}{(e_1\ e_2), \rho, \sigma \Downarrow_{pc}^{E} \sigma''', v'}$$

CONST
$$\overline{c, \rho, \sigma \Downarrow_{pc}^{E} \sigma, c}$$

VAR
$$\overline{x, \rho, \sigma \Downarrow_{pc}^{E} \sigma, \rho(x)}$$

LAMBDA
$$\overline{\lambda x.\, e, \rho, \sigma \Downarrow_{pc}^{E} \sigma, \langle \lambda x.\, e, \rho \rangle}$$

BOX
$$\frac{e, \rho, \sigma \Downarrow_{pc}^{E} \sigma', v \qquad \alpha \notin dom(\sigma')}{\sigma'' = \sigma'[\alpha \mapsto \langle\langle pc\, ?\, v \diamond \star \rangle\rangle]}{(\textbf{box}\ e), \rho, \sigma \Downarrow_{pc}^{E} \sigma'', \alpha}$$

UNBOX
$$\frac{e, \rho, \sigma \Downarrow_{pc}^{E} \sigma', v \qquad v' = read(\sigma', v, pc)}{(\textbf{unbox}\ e), \rho, \sigma \Downarrow_{pc}^{E} \sigma', v'}$$

SET
$$\frac{e_1, \rho, \sigma \Downarrow_{pc}^{E} \sigma', v_1 \qquad e_2, \rho, \sigma' \Downarrow_{pc}^{E} \sigma'', v_2 \qquad \sigma''' = write(\sigma'', v_1, pc, v_2)}{(\textbf{set!}\ e_1\ e_2), \rho, \sigma \Downarrow_{pc}^{E} \sigma''', v_2}$$

(Facet Creation)

FAC-CREATE-SPLIT
$$\frac{e_1, \rho, \sigma \Downarrow_{pc}^{E} \sigma', \ell \qquad \{+\ell, -\ell\} \cap pc = \varnothing \\ e_2, \rho, \sigma' \Downarrow_{pc \cup \{+l\}}^{E} \sigma'', v_1 \qquad e_3, \rho, \sigma'' \Downarrow_{pc \cup \{-l\}}^{E} \sigma''', v_2 \\ v = \langle\langle pc \cup \{+\ell\}\, ?\, v_1 \diamond v_2 \rangle\rangle}{(\textbf{fac}\ e_1\ e_2\ e_3), \rho, \sigma \Downarrow_{pc}^{E} \sigma''', v}$$

FAC-CREATE-POS
$$\frac{e_1, \rho, \sigma \Downarrow_{pc}^{E} \sigma', \ell \qquad +\ell \in pc \\ e_2, \rho, \sigma' \Downarrow_{pc}^{E} \sigma'', v}{(\textbf{fac}\ e_1\ e_2\ e_3), \rho, \sigma \Downarrow_{pc}^{E} \sigma'', v}$$

FAC-CREATE-NEG
$$\frac{e_1, \rho, \sigma \Downarrow_{pc}^{E} \sigma', \ell \qquad -\ell \in pc \\ e_3, \rho, \sigma' \Downarrow_{pc}^{E} \sigma'', v}{(\textbf{fac}\ e_1\ e_2\ e_3), \rho, \sigma \Downarrow_{pc}^{E} \sigma'', v}$$

(Label Creation / Observation)

LET-LABEL
$$\frac{e_1, \rho, \sigma \Downarrow_{pc}^{E} \sigma', \langle \lambda x.\, e, \rho' \rangle \\ \alpha \notin dom(\sigma') \qquad \sigma'' = \sigma'[\alpha \mapsto \langle \lambda x.\, e, \rho' \rangle] \\ e_2, \rho[\ell \mapsto \alpha], \sigma'' \Downarrow_{pc}^{E} \sigma''', v}{(\textbf{let-label}\ \ell\ e_1\ e_2), \rho, \sigma \Downarrow_{pc}^{E} \sigma''', v}$$

OBS
$$\frac{e_1, \rho, \sigma \Downarrow_{pc}^{E} \sigma', \ell \qquad e_2, \rho, \sigma' \Downarrow_{pc}^{E} \sigma'', v \\ \langle(\lambda x.\, e), \rho'\rangle = \sigma''(\ell) \\ e, \rho'[x \mapsto v], \sigma'' \Downarrow_{pc}^{E} \sigma''', v^{\pm} \\ e_3, \rho, \sigma''' \Downarrow_{pc}^{E} \sigma'''', v' \\ v'' = obs(\ell, v', v^{\pm})}{(\textbf{obs}\ e_1\ e_2\ e_3), \rho, \sigma \Downarrow_{pc}^{E} \sigma'''', v''}$$

(Possibly-Faceted Application) $\boxed{(v_1\ v_2), \rho, \sigma \Downarrow_{pc}^{A} \sigma, v}$

APP-SPLIT
$$\frac{\{+\ell, -\ell\} \cap pc = \varnothing \qquad (v^+\ v), \rho, \sigma \Downarrow_{pc \cup \{+\ell\}}^{A} \sigma', v^{+'} \\ (v^-\ v), \rho, \sigma' \Downarrow_{pc \cup \{-\ell\}}^{A} \sigma'', v^{-'} \qquad v' = \langle\langle \{+\ell\}\, ?\, v^{+'} \diamond v^{-'} \rangle\rangle}{(\langle \ell\, ?\, v^+ \diamond v^- \rangle\ v), \rho, \sigma \Downarrow_{pc}^{A} \sigma'', v'}$$

APP-★
$$\overline{(\star\ v), \rho, \sigma \Downarrow_{pc}^{A} \sigma, \star}$$

APP-BASE
$$\frac{e, \rho'[x \mapsto v], \sigma \Downarrow_{pc}^{E} \sigma', v'}{(\langle \lambda x.\, e, \rho' \rangle\ v), \rho, \sigma \Downarrow_{pc}^{A} \sigma', v'}$$

APP-FACET-POS
$$\frac{+\ell \in pc \\ (v^+\ v), \rho, \sigma \Downarrow_{pc}^{A} \sigma', v'}{(\langle \ell\, ?\, v^+ \diamond v^- \rangle\ v), \rho, \sigma \Downarrow_{pc}^{A} \sigma', v'}$$

APP-FACET-NEG
$$\frac{-\ell \in pc \\ (v^-\ v), \rho, \sigma \Downarrow_{pc}^{A} \sigma', v'}{(\langle \ell\, ?\, v^+ \diamond v^- \rangle\ v), \rho, \sigma \Downarrow_{pc}^{A} \sigma', v'}$$

Fig. 2. Semantics of Faceted Execution

$$\langle\langle \varnothing \,?\, v^+ \diamond v^d \rangle\rangle \;=\; v^+$$
$$\langle\langle \{+\ell\} \cup rest \,?\, v^+ \diamond v^d \rangle\rangle \;=\; \langle \ell \,?\, \langle\langle rest \,?\, v^+ \diamond v^d \rangle\rangle \diamond v^d \rangle$$
$$\langle\langle \{-\ell\} \cup rest \,?\, v^+ \diamond v^d \rangle\rangle \;=\; \langle \ell \,?\, v^d \diamond \langle\langle rest \,?\, v^+ \diamond v^d \rangle\rangle \rangle$$

$$write(\sigma, \alpha, pc, v) \;=\; \sigma[\alpha := \langle\langle pc \,?\, v \diamond \sigma(\alpha) \rangle\rangle]$$
$$write(\sigma, \langle \ell \,?\, v_1 \diamond v_2 \rangle), pc, v \;=\; \sigma'' \qquad\qquad where\ \sigma' = write(\sigma, v_1, pc \cup \{+\ell\}, v)$$
$$and\ \sigma'' = write(\sigma', v_2, pc \cup \{-\ell\}, v)$$

Fig. 3. Meta-functions used in our semantics

The reduction relation $e, \rho, \sigma \Downarrow^E_{pc} \sigma, v$ reduces an expression, environment, and store to a resulting store and value. The first three rules (all in blue) are unchanged from the standard interpretation in the lambda calculus. The Apply rule calls out to the helper relation $(v\ v), \rho, \sigma \Downarrow^A_{pc} \sigma, v$, which applies a possibly-faceted value to an argument: if the value being applied is a plain (unfaceted) closure, the App-Base rule (in blue, as it is unchanged from the lambda calculus) applies it and returns immediately to the $\Downarrow^E_{pc}$ relation.

In the case that a faceted value is applied, $\Downarrow^A_{pc}$ performs one of three functions, based on the relation of ℓ to pc. If there is no occurrence of either $+\ell$ or $-\ell$ in pc, then the semantics has not yet branched on ℓ, and therefore must split the application. To do this, it applies both the positive and negative branches after extending pc. After reducing both branches to values, the results are formed into a facet. If $+\ell \in pc$, then the semantics has already branched on the label ℓ, so splitting would be redundant. In this case, $\Downarrow^A_{pc}$ simply selects the positive branch to apply and continues without splitting. The symmetric case occurs in App-Facet-Neg. Facet formation follows this pattern, accounting for the relation of ℓ to pc.

The rules Box, Unbox, and Set appear similar to the standard implementation of boxes, but employ several meta-functions to do their work. This is because box creation, reads, and writes may occur within a privileged context, and care must be taken to form facets when pc is nonempty. To understand why, consider the following example[1]:

```
1  (define x (box 0))
2  (if (= (facet alice 0 1) 0)
3    (set! x 0)
4    (set! x 1))
5  (unbox x)
```

If we do nothing special to account for the fact that the program branches on the facet, control flow implicitly launders the value through the box to an unfaceted value. To fix this, we form a facet by taking into account pc and forming a facet using the meta-function $\langle\langle \ell \,?\, v^+ \diamond v^d \rangle\rangle$. This meta-function is defined in Figure 3, and takes three arguments: the current pc, a positive view (v^+), and a "default" view (v^d). Facet construction builds a facet with a spine corresponding to all of the labels in pc, and inserts v^+ at the focus corresponding to pc, putting the default value v^d along all other branches. In the **box** form, the facet uses a default value of $\star$. In terms of our above example, this means that along the positive branch x would be set to $\langle alice \,?\, 1 \diamond \star \rangle$ (as the false branch of the **if** is taken), and along the subsequent negative branch x is extended to $\langle alice \,?\, 1 \diamond 0 \rangle$.

Label creation allocates a label as a fresh address in the store, binding the specified label predicate and adding it to the environment. Labels must be store-allocated rather than bound in the lexical environment, as the latter would allow the label to be rebound by anyone using the facet:

---

[1]Our formal semantics elides **if**, though it may be obtained via a Church encoding if desired as in Austin et al.[1]. Our implementation includes **if** but not other constructs such as **cond**

```
1 (define alice-label (let-label l (λ (x) (= x alice)) l))
2 (define x (facet alice-label 1 0))
3 (let ([alice-label (let-label l (λ (x) #t) l)])
4   (obs alice-label 1 x)) ; Should return ⟨alice-label ? 1 ⋄ 0⟩
```

The **obs** form in the above example ought to return $\langle$ alice-label $?\,1\diamond 0\,\rangle$. But if we pull labels from the lexical environment, the binding on line 3 shadows the policy originally associated with the facet.

Last, observation evaluates the label expression to an address and executes the associated predicate. Once this is done, Obs uses the *obs* meta-function to select the appropriate branch based on the value returned by the predicate associated with the label. This meta-function accounts for the fact that the label being observed may appear arbitrarily deep in the facet (or not at all). As our implementation of *obs* is unchanged from its definition in [1], we elide it here.

4 FACETED EXECUTION AS MACROS

The semantics of faceted execution is an extension of the lambda calculus, leading to a natural question: can we use Racket's macros [9, 13] to extend Racket to faceted execution? We will see that the answer is yes, and the translation from the big-step rules is surprisingly straightforward. This section of our paper describes the design of Racets, a prototype implementation of faceted execution using Racket macros. In Section 7 we remark upon current directions scaling Racets to the whole of Racket.

Choosing a Representation for Facets, Labels, and Program Counters. In setting out to implement facets, we must first choose how we will represent facets, labels, and program counters. We have chosen to implement facets simply as Racket **struct**s, containing a label along with positive and negative branches:

```
1 (struct facet (labelname left right))
```

Next, we must choose a representation of labels. At first consideration, it appears sensible to represent labels simply as closures. After all, labels are simply used as predicates testing whether or not to reveal a facet's positive or negative branch. Therefore, we represent labels as a pair of a name and a policy:

```
2 (struct labelpair (name pol))
```

Now that we have defined labels, we can define branches, which are positive or negative labels:

```
3 (struct pos (lab))
4 (struct neg (lab))
```

Similarly, program counters are sets of branches. However, we must still ask how we will keep track of the "current" program counter. Our implementation uses Racket's parameters, though other mechanisms (such as continuation marks [6], to which parameters macro-expand) can also be used. Racets defines the parameter current-pc, and updates it as computation progresses:

```
5 (define current-pc (make-parameter (set)))
```

Facet Creation. Facet creation appears as three separate rules in Figure 2, which we recapitulate here in three distinct colors for each case:

Fac-Create-Split

$$e_1, \rho, \sigma \Downarrow_{pc}^E \sigma', \ell \qquad \boxed{\{+\ell, -\ell\} \cap pc = \varnothing}$$

$$e_2, \rho, \sigma' \Downarrow_{pc\cup\{+\ell\}}^E \sigma'', v_1$$

Fac-Create-Pos   Fac-Create-Neg $$e_3, \rho, \sigma'' \Downarrow_{pc\cup\{-\ell\}}^E \sigma''', v_2 \qquad v = \langle\langle pc\,?\,v_1 \diamond v_2 \rangle\rangle$$

$$e_1, \rho, \sigma \Downarrow_{pc}^E \sigma', \ell \quad \boxed{+\ell \in pc}$$ $$e_1, \rho, \sigma \Downarrow_{pc}^E \sigma', \ell \quad \boxed{-\ell \in pc}$$

$$e_2, \rho, \sigma \Downarrow_{pc}^E \sigma'', v$$ $$e_3, \rho, \sigma \Downarrow_{pc}^E \sigma'', v$$

$$\overline{(\textbf{fac }e_1\,e_2\,e_3), \rho, \sigma \Downarrow_{pc}^E \sigma'', v}$$ $$\overline{(\textbf{fac }e_1\,e_2\,e_3), \rho, \sigma \Downarrow_{pc}^E \sigma'', v}$$ $$\overline{(\textbf{fac }e_1\,e_2\,e_3), \rho, \sigma \Downarrow_{pc}^E \sigma''', v}$$

```
6  (define-syntax-rule (facet l e1 e2)
7    (cond
8      [(set-member? (current-pc) (pos (facet-labelname l))) e1]
9      [(set-member? (current-pc) (neg (facet-labelname l))) e2]
10     [else
11      (let ([left (parameterize
12                    ([current-pc (set-add (current-pc)
13                                          (pos (facet-labelname l)))])])
14             [right (parameterize
15                     ([current-pc (set-add (current-pc)
16                                           (neg (facet-labelname l)))])])
17             (mkfacet (set-union (set (pos (labelpair-name l)))
18                                 (current-pc))
19                      v1 v2))]))
```

Fig. 4. Facet creation as a macro

Translating these rules to Racket involves observing that each one will apply under one of three disjoint circumstances (each of them boxed in the above rules): $+\ell \in pc$, $-\ell \in pc$, or else $\{+\ell, -\ell\} \cap pc = \varnothing$. This is a common idiom in our faceted semantics, as we often want to select the appropriate branch of a facet if its label already exists in pc.

At first glance, it may not be obvious that we even *need* a macro for facet creation. But according to our semantics, the following snippet should produce #t if $+\ell \in pc$:

```
(facet l #t (error "this shouldn't get evaluated if +ℓ ∈ pc"))
```

If we were to implement `fac` as a function, it would force evaluation of the negative branch, inconsistent with the semantics of FAC-CREATE-POS. We can implement each of these conditions as a Racket macro by considering whether $\ell \in pc$, as shown in Figure 4. Each color in the listing corresponds to the analogous semantic rule. The implementation of FAC-CREATE-POS and FAC-CREATE-NEG is relatively straightforward, but FAC-CREATE-SPLIT extends pc for each branch and subsequently forms a facet. This function implements canonicalizing facet construction, and (as the implementation is a transliteration of that in Figure 6 of Austin et al. [1]) we omit its definition here.

Label Creation. As we chose a representation of labels as pairs of symbols (the label's name) and closures (the predicate corresponding to the label), label creation is relatively straightforward from the semantics:

```
20  (define-syntax-rule (let-label l (λ xs e) body)
21    (let ([l (labelpair (gensym 'lab)
22                        (λ xs e))])
23      body))
```

Faceted Boxes, Writes, and Observations. Our faceted semantics includes explicit `box` and `unbox` forms. This differs from Racket's semantics, where any variable may be treated as a box due to assignment conversion. We have two main options:

- Introduce an explicit `unbox` form in RACETS, trusting the programmer to explicitly use our implementation of `unbox` on potentially-faceted objects.
- Walk over Racket code (after macro-expansion via `local-expand`) transforming variable references to use explicit `unbox` forms from RACETS.

For our prototype of RACETS, we have chosen to implement the first. This leads to a relatively simple implementation, but essentially trusts the programmer to use RACETS' `unbox` forms when necessary.

```
24   (define-syntax (ref-set! stx)
25     (syntax-case stx ()
26       [(_ var e)
27        #`(let ([v e])
28            (let write ([var var]
29                       [pc (current-pc)])
30              (if (box? var)
31                  ; write(σ, α, pc, v)
32                  (set-box! var (construct-facet (current-pc) v (unbox var)))
33                  ; Else split
34                  (mkfacet
35                    (facet-labelname (unbox var))
36                    ; write(σ, α, pc ∪ {+l}, v)
37                    (write
38                      (facet-left (unbox var))
39                      (set-add pc (pos (facet-labelname var))))
40                    ; write(σ, α, pc ∪ {-l}, v)
41                    (write
42                      (facet-right (unbox var))
43                      (set-add pc (neg (facet-labelname var)))))))))]))
```

Fig. 5. Racets' implementation of **set!**

Racets defines a **box** macro, along with **unbox** and **set!**. We include the definition of **set!** in Figure 5, which inlines the definition of the *write* metafunction from Section 3 to consider the case under which a facet is used when an address is expected.

Facet observation is handled similarly, first evaluating the label to produce a policy predicate, followed by evaluating the policy's argument and a possibly-faceted value to observe. After applying the policy its argument, we produce the value $v^{\pm}$ and descend down the facet until reaching either a base value or finding the selected label (at which point we select the appropriate branch).

Faceted Function Application. By now we can anticipate a predictable pattern for implementing faceted execution: check pc to decide whether to branch left, right, or split. This is largely our strategy for handling faceted function application, with a small twist: we need to be able to apply functions from outside of Racets. For example, if we want to apply builtin functions such as display, we need to be mindful of the fact that these functions cannot work with faceted arguments.

To handle this, we implement a macro for the λ form, to tag Racets closures specifically (so that they are differentiated from functions outside of the current module). In the case that a foreign function is applied to a faceted value, our implementation of function application wraps the function to be able to handle facets by distributing the function through each branch of the facet.

In general it is unsafe to apply an unknown function to a faceted value. This is because the unknown function may leak the facet's private information as a side-effect. Therefore, our current implementation of Racets allows programmers to apply external functions, but does not make any guarantee of safety. A better strategy may be to perform an **obs** before each call to a potentially-unsafe function. In general, we believe module interactions are a challenging problem in faceted execution, and we leave its study to future work.

5 IMPLEMENTATION AND EVALUATION

We have implemented Racets as a set of Racket macros which can be employed as a language using Racket's #lang reader facility. So far, we have included macros for many of Racket's core forms including application,
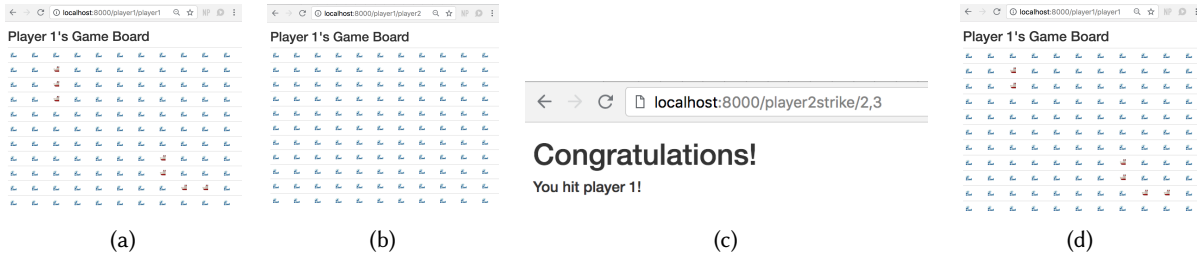
Fig. 6. Screenshots from our Battleship case study.

if, λ, and references. We leave others (including continuation marks) to future work. Additionally, RACETS does not support first-class control. There has been recent work in handling exceptions in the context of faceted execution [2], however reconciling first-class control in general remains (to our knowledge) an open problem.

Because of Racket's flexible macro system, our implementation of faceted execution is much smaller than other systems: our core macros comprise roughly 170 source lines of Racket, with another 120 lines of library code to perform various facet-related functions.

Our implementation is currently available on Github at https://github.com/fordsec/racets

Case Study: Battleship in Racets. To gain perspective on how Racets enables policy-agnostic programming, we scaled our implementation from Section 2 to a web-based game of Battleship written in Racets. Our implementation uses Racket's web-server framework [14], which defines an API for writing HTTP-based server applications.

Figure 6 shows several screenshots of our Battleship application. The first screenshot 6a shows the board as viewed by player 1 (using the route /player1/player1), while the second shows the empty board observed when player 2 attempts to view player 1's board. The screenshot in 6c shows the response seen by player 2 upon a successful hit. Finally screenshot 6d shows player one's board with the ship on $(2, 3)$ removed.

When running, our game server provides several routes that a user can access:

```
1  (define-values (dispatch generate-url)
2    (dispatch-rules
3      [("player1" (string-arg)) player1board]
4      [("player2" (string-arg)) player2board]
5      [("player1strike" (string-arg)) p1strike]
6      [("player2strike" (string-arg)) p2strike]))
```

The route /player1/<id> (or /player2/<id>) renders player 1's game board when viewed as <id>. We facet game boards with policies that reveal player 1's board when <id> is player1, and do the same with player 2:

```
7   (define p1l (mkpol "player1"))
8   (define p1board
9     (box (facet p1l
10               (add-pieces (makeboard) '(1 2 ldots))
11               (makeboard))))
```

The board is explicitly made into a box: this is because the board's state will change as player 2 makes moves and eliminates pieces from their board. The player1board function implements the logic to render player

1's board as an HTML table. The argument `viewer` corresponds to the `<id>` route argument, and is passed to `player1board` by the framework. This argument is then used to observe the board game

```
12  (define player1board
13    (ext-λ (request viewer)
14          (http-response "<h1>Player␣1's␣Game␣Board</h1>"
15                         (pretty-print
16                           (obs p1l name (deref p1board))))))
```

The implementation of `player1board` uses a special form `ext-λ`, discussed at the end of this section, to allow code from Racets to be executed by the framework (which is not prepared to execute faceted code).

The function `p1strike` allows player 1 to make a guess as to the position of ships on player 2's board. The function parses the position into two coordinates and then calls `mark-hit` to perform the hit, updating player 2's board and then observing the result to answer (to player 1) whether the result was a hit or not:

```
17  (define p1strike
18    (ext-λ
19      (request position)
20      (let* ([x (char-to-num (string-ref position 0))]
21             [y (char-to-num (string-ref position 2))]
22             [ans (mark-hit p2board x y)]
23        (set! p2board (car ans))
24        (http-response
25          (if (cdr (obs p2l "player2" ans))
26              "<h1>Congratulations!</h1>␣<h4>You␣hit␣player␣2!</h4>"
27              "<p>No␣hit␣:(</p>")))))))
```

Note that we need to use an explicit **obs** form on line 25. This is because—as `p2board` is a facet—the result will also be a faceted value. When we want to display the output to player 1, our code needs to explicitly observe the answer, as `http-response` cannot accept a faceted value.

Module Interactions in Racets. There is a wealth of existing Racket code we may like to incorporate into Racets programs. For example, our case study uses the `web-server` framework for building web applications. However, in general, we believe that interacting with code not written using faceted execution is a challenging open problem, and we do not know of any principled solutions in the literature.

One immediate problem in Racets is how to pass functions from Racets to plain Racket code. For example, the `web-server` framework is written in Racket, and does not know how to call tagged closures from Racets. As a stopgap, we added an `ext-λ` form to Racets. This form allows creating a Racket-style lambda in Racets that will be used by functions in other modules, necessary for the implementation of our case study.

We plan to explore interactions with unfaceted code more in the future, and believe it will an exciting direction. For example, once execution escapes Racets, we have no guarantee that the privacy policy won't be violated. One solution may be to implicitly perform an **obs** based on the current *pc* at points where Racets interacts with unfaceted modules. But we do not fully understand the ramifications or ergonomics of this choice, and suspect there may be a wide array of design choices to handle these module interactions including security type systems and blame (to track which module violated the privacy policy).

6   RELATED WORK

To the best of our knowledge, we are the first authors to present an implementation of faceted execution using hygienic macros. There are several threads of related work in dynamic information flow and programming paradigms for information flow.

Information-flow was first formalized by Denning [7]. In her seminal work on a lattice model for information flow, she outlined challenges and potential solutions to static information-flow checking. Subsequently, Goguen

and Meseguer [10] defined noninterference, which formalized the idea that privileged data should not influence publicly observable outputs. Clarkson and Schneider [5] later recognized that information-flow properties fit into a class of program properties that could not be characterized by a single trace of a program, but rather a set of traces, and called these hyperproperties.

Along with definitions of information flow, there has also been significant interest in mechanisms for enforcing information flow. This work can be broadly divided into static and dynamic enforcement mechanisms for information flow security. Of the mechanisms for static information flow, security type systems have gained the most use. First introduced by Volpano and Smith [21], these type systems augment the binding environment to track the privilege of variables and prevent writes to variables that would violate noninterference. Myers leveraged this idea to produce Jif, a variant of Java with an information-flow type system [16]. Security type systems have been subsequently extended to accommodate concurrent programs [24] and flow sensitivity [11]. Faceted execution does not require annotating the program with security types, but at the expense of losing a static characterization of the program's security in its type system.

Devriese and Piessens [8] first introduced secure multi-execution as a dynamic enforcement technique for information flow. Secure multi-execution runs 2^k copies of a program in parallel, where each run represents a subset of $\wp(Prin)$, where $Prin$ is a set of principals. For example, if the principals in the program are Alice and Bob, secure multi-execution executes four copies of the program: one that replaces all secret inputs by $\bot$, one that replaces Bob's input by $\bot$ but Alice's input by the true input, one for Bob's input, and one with access to all privilaged information. When external effects are made (e.g., writing to disc), the runtime can select which variant to use based on a policy. Secure multi-execution prevents information flow violations at runtime by ensuring that observations which violate the information-flow policy receive a view of the data computed without access to the secret inputs. Secure multi-execution has been extended in a variety of ways, e.g., scaling to its implementation in web browsers [4], adding declassification in a granular way [18], and even preventing side-channel attacks [12].

As the number of principals increases, secure multi-execution's overhead increases exponentially, unnecessarily duplicating work not influenced by secret inputs. Austin et al. introduced faceted execution as an optimization of secure multi-execution in [1]. Instead of treating the whole program as a potentially-secret computation, faceted execution realizes that influence can be tracked and propagated in a granular way using facets. Notably, Austin et al.'s work does not include first-class labels, as it was simulating secure multi-execution, where the principals could not be dynamically generated.

At the same time, Yang et al. first implemented Jeeves, a language allowing policy-agnostic programming [23]. Policy-agnostic programming takes the view that programs should be written without regard to a particular privacy policy, because as the policy changes, correctly updating program logic is cumbersome and error-prone. Policy-agnostic programming was first implemented in the domain-specific language Jeeves, using an SMT solver to decide which view of secret data to reveal based on a policy. Later, both authors collaborated to implement Jeeves using faceted execution. [3]. This formulation includes first-class labels, and is the basis for our concrete semantics.

Several other efforts into dynamic analysis for information flow are worth noting. Stefan et al. [20] first presented `LIO`—a monad (with implementation in Haskell) that tracks privilege of the current program counter and forbids effects that would violate the security policy. It may be surprising that `LIO` works well for Haskell programs, given that faceted execution is more precise than `LIO`—allowing values to become faceted rather than halting the program. One key difference is that Haskell programs emphasize purity while languages such as JavaScript (the original target of faceted execution) does not, so much of the machinery for faceted execution's effect on the store is less interesting. Several authors have implemented related systems to `LIO`, including variants of faceted execution [19] and variants of `LIO` that extend its power to arbitrary monad transformers [17].

We believe that it would be possible to implement a variant of our technique that would give similar insights to programs using `LIO`, though much of the interesting machinery for handling state may be unnecessary.

7   CONCLUSION AND FUTURE WORK

In this paper, we have reviewed the operation of faceted execution, a linguistic paradigm enabling policy-agnostic programming, and showed how it may be implemented within the Racket programming system as a library of macros. As Racket macros permit core language forms (including function application, λ-abstraction, conditionals, mutation, etc.) to be rewritten arbitrarily, it is possible to modify the meaning of these forms to support a faceted semantics directly. We call our prototype system Racets: Racket with Facets.

The advantage of this approach is that faceted, policy-agnostic, programs may be written directly in Racket, making use of the wealth of Racket code already available. A central challenge of this then, is how to ensure there is a sound (w.r.t. secure multi-execution) and practical inter-operation between Racets and standard Racket (or other languages written as libraries in Racket). Our approach to this has been to use a tagging scheme that identifies values from Racets so untagged values may be treated by Racets as originating from a non-Racets language. For example, a pure Racket function that is not tagged, being applied at a `#%app` form in Racets, can be automatically lifted to support FE (so that it can split when applied on a faceted value).

Our hypothesis is that this tagging scheme is key to permitting inter-operation between Racets and Racket, and we have implemented a faceted, web-based game of Battleship to explore this idea. We suspect that a more thorough investigation of likely idioms for faceted, non-faceted module interaction is needed and that purely functional code plays a special role as a degenerate case where arbitrary non-faceted code may be lifted to operate over facted values without potential unsoundness. In the future, we plan to explore these design choices in a more principled way and also to scale a static analysis [15] of faceted execution to fully expanded Racket so it may be applied directly to Racets.

REFERENCES

[1] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 165–178. DOI : http://dx.doi.org/10.1145/2103656.2103677

[2] Thomas H. Austin, Tommy Schmitz, and Cormac Flanagan. 2017. Multiple Facets for Dynamic Information Flow with Exceptions. *ACM Trans. Program. Lang. Syst.* 39, 3, Article 10 (May 2017), 56 pages. DOI : http://dx.doi.org/10.1145/3024086

[3] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '13)*. ACM, New York, NY, USA, 15–26. DOI : http://dx.doi.org/10.1145/2465106.2465121

[4] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. 2011. Reactive non-interference for a browser model. In *2011 5th International Conference on Network and System Security*. IEEE, 97–104. DOI : http://dx.doi.org/10.1109/ICNSS.2011.6059965

[5] M. R. Clarkson and F. B. Schneider. 2008. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*. IEEE, 51–65. DOI : http://dx.doi.org/10.1109/CSF.2008.7

[6] John Clements, Matthew Flatt, and Matthias Felleisen. 2001. Modeling an Algebraic Stepper. In *Proceedings of the 10th European Symposium on Programming Languages and Systems (ESOP '01)*. Springer-Verlag, London, UK, UK, 320–334. http://dl.acm.org/citation.cfm?id=645395.651947

[7] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (May 1976), 236–243.

[8] D. Devriese and F. Piessens. 2010. Noninterference through Secure Multi-execution. In *2010 IEEE Symposium on Security and Privacy (Oakland '10)*. 109–124. DOI : http://dx.doi.org/10.1109/SP.2010.15

[9] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1992. Syntactic Abstraction in Scheme. *Lisp Symb. Comput.* 5, 4 (Dec. 1992), 295–326. DOI : http://dx.doi.org/10.1007/BF01806308

[10] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. DOI : http://dx.doi.org/10.1109/SP.1982.10014

[11] Sebastian Hunt and David Sands. 2006. On Flow-sensitive Security Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 79–90. DOI : http://dx.doi.org/10.1145/1111037.1111045

[12] V. Kashyap, B. Wiedermann, and B. Hardekopf. 2011. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *2011 IEEE Symposium on Security and Privacy (Oakland '11)*. 413–428. DOI:http://dx.doi.org/10.1109/SP.2011.19

[13] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP '86)*. ACM, New York, NY, USA, 151–161. DOI:http://dx.doi.org/10.1145/319838.319859

[14] Jay McCarthy. Web Applications in Racket. (????). https://docs.racket-lang.org/web-server/index.html (Accessed 7/23/18).

[15] K. Mikinski, D. Darais, and T. Gilray. 2019. Abstracting Faceted Execution.. In *(In submission to) SIGPLAN Symposium on Principles of Programming Languages*.

[16] Andrew C. Myers. 1999. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. ACM, New York, NY, USA, 228–241. DOI:http://dx.doi.org/10.1145/292540.292561

[17] James Parker. 2014. *LMonad: Information Flow Control for Haskell Web Applications*. Master's thesis. University of Maryland, College Park, Maryland.

[18] W. Rafnsson and A. Sabelfeld. 2013. Secure Multi-execution: Fine-Grained, Declassification-Aware, and Transparent. In *2013 IEEE 26th Computer Security Foundations Symposium*. 33–48. DOI:http://dx.doi.org/10.1109/CSF.2013.10

[19] Thomas Schmitz, Dustin Rhodes, Thomas H. Austin, Kenneth Knowles, and Cormac Flanagan. 2016. Faceted Dynamic Information Flow via Control and Data Monads. In *Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635 (POST '16)*. Springer-Verlag New York, Inc., New York, NY, USA, 3–23. DOI:http://dx.doi.org/10.1007/978-3-662-49635-0_1

[20] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. ACM, New York, NY, USA, 95–106. DOI:http://dx.doi.org/10.1145/2034675.2034688

[21] Dennis M. Volpano and Geoffrey Smith. 1997. A Type-Based Approach to Program Security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT '97)*. Springer-Verlag, London, UK, UK, 607–621. http://dl.acm.org/citation.cfm?id=646620.697712

[22] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-backed Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 631–647.

[23] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 85–96. DOI:http://dx.doi.org/10.1145/2103656.2103669

[24] S. Zdancewic and A. C. Myers. 2003. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop, 2003. (CSF '13)*. 29–43. DOI:http://dx.doi.org/10.1109/CSFW.2003.1212703