# Information Flow Control on the Web with Racets

Matthew Soulanille

Advisor: Kris Micinski

December 21, 2018

**Abstract**

Faceted execution is a technique for dynamic information flow control that allows programmers to program in a policy-agnostic manner. Racets uses macros to extend Racket with support for faceted execution. We present the beginnings of a web framework built in Racets that links client-side button presses to declassification of faceted values in a way that makes it easy for programmers to understand how private data flows in their applications.

# Contents

# 1 Introduction

Maintaining the confidentiality of sensitive information is a long-standing and increasingly difficult problem for programmers. As new tools are built on top of old ones, programmers are forced to look at an increasing number of ways confidential information may be leaked from their programs. This problem is especially prevalent in web development, where new frameworks become old in as little as half a year's time. [1] This thesis presents an information flow based approach to maintaining confidentiality. It aims to make it easy for programmers to control where sensitive information and its derivatives are accessed without significantly changing the way they program.

## 1.1 Static versus Dynamic Analysis

Information flow can be analyzed during the compilation of the program or during its execution. When done during compilation, it is called static information flow analysis, and when done during execution, it is called dynamic information flow analysis. This paper deals mostly with the latter but provides some background on the former in 2.2.2.

## 1.2 Non-Interference

When designing a secure system, it is necessary to define what constitutes security. For this paper, security will be defined as termination-insensitive non-interference. [2, p. 7] To unpack this, we first define non-interference. Informally, a program that takes public and private inputs and returns public and private outputs satisfies non-interference if an attacker with access to the public inputs and outputs of the program can not tell the difference between the the program being run with one set of private inputs and it being run running with another set [3, p. 2]. In this way, the attacker is isolated from the private context of the program, and can not learn anything about the private inputs or outputs. Since this paper deals with dynamic information flow, we use a downgraded form of non-interference called termination-insensitive non-interference that allows the termination or lack of termination of the program to leak information about private data [4, abstract].

# 2 Background

## 2.1 Implicit vs Explicit Flows

Most of the time, information flows through a program through variable assignments, for example, $x := y + z$. These are called explicit flows because they directly pass information between variables. [3, p. 3] However, information can also flow through the control structures of a program. This is called an implicit flow because information flows from one variable to another without an explicit assignment involving the two.[2, p. 2]

Consider the following program that uses input $s_1$:

$$p_1 := \text{true}$$
$$\textbf{if } s_1 \textbf{ then } p_1 := \text{false}$$

Figure 1: Implicit Flow Negation

This program is equivalent to $p_1 := \neg s_1$, however, instead of using an explicit flow to set $p_1$ equal to $\neg s_1$, it uses the control flow of an **if** statement to set $p_1$ to false only if $s_1$ is true.

The logical conjunction of two values can be computed in a similar manner. With inputs $s_1$ and $s_2$, the following program computes $p_1 := s_1 \vee s_2$.

$$p_1 := \text{false}$$
$$\textbf{if } s_1 \textbf{ then } p_1 := \text{true}$$
$$\textbf{if } s_2 \textbf{ then } p_1 := \text{true}$$

Figure 2: Implicit Flow Conjunction

Since logical negation and logical conjunction can be built from implicit flows, and since negation with conjunction is universal, (and since loops are a form of implicit flow) implicit flows are universal. This means that any value that can be computed with explicit flows can also be computed with implicit flows. Therefore, tracking implicit flows is just as important as tracking explicit flows when performing information flow analyses.

## 2.2 Prior Information Flow Control Techniques

### 2.2.1 Mandatory Access Control

Mandatory access control is a specification for dynamic information flow control that uses security levels to maintain security. Each data item is given a security level, and when a new data item is computed from inputs, it takes on the most restrictive security level out of the security levels of the inputs, thereby preventing explicit flows from leaking information. [3, p. 3]

To handle implicit flows, mandatory access control assigns a "process sensitivity label" to any part of the program where execution depends on the value of a data item (such as in an **if** statement). [3, p. 3] If a variable with a less restrictive security level would be updated in a more restrictive context, mandatory access control terminates with an error.

In computing Figure 1 with secure input $s_1$ and insecure output $p_1$, mandatory access control assigns the **if** statement a process sensitivity label equal to $s_1$'s label. If $s_1$ is true, then the **if** statement attempts to assign $p_1$ to false, halting the program since $p_1$'s label is less restrictive than $s_1$'s label. If $s_1$ is false, then the **if** statement is never run, so the program continues. This behavior leaks $s_1$. An attacker can perform a low-sensitivity process after the **if** statement, and if that process occurs, they know that the program did not halt, so $s_1$ must have been false. [3, p. 3] While this behavior technically satisfies termination-insensitive non-interference, it is undesirable since it results in a large number of reasonable programs terminating and leaking secure information. An alternative is to keep the process sensitivity label high after the **if** statement regardless of whether it was executed. This prevents the $s_1 = \text{false}$ case (in which the program does not halt) from leaking information into an insecure context because all contexts after the **if** statement are now as secure as the **if** statement itself. However, this also produces an effect called *label-creep*, in which the security of the process sensitivity label monotonically increases, preventing certain variable assignments after the branch, which is undesirable. [3, p. 3]

As an example of label creep, the following program with secure input $s_1$ and public $p_1, p_2$ always results in an error in mandatory access control because $p_1$, a public value, is assigned within a higher security context (since the process sensitivity level remains at the value it was during the **if** statement).

$p_1 := \text{true}$
$p_2 := \text{false}$
**if** $s_1$ **then** do nothing
$p_1 := p_2$

Figure 3: Rejected by Mandatory Access Control

### 2.2.2 An Information Flow Control Type System

A security-type system is a method for static information flow control that implements security labels as types. In a security-typed language, a type represents what kind of data a variable stores *and* how that data may be accessed. [3, p. 4] At compile time, the type checker verifies that the program can not contain insecure information flows at run time.

By looking at all possible execution paths instead of a single path as in 2.2.1, a security-type system can prove that no set of inputs to a program could result in improper information flow. In this way, security-type systems solve the problem of label-creep that mandatory access control exhibits. Instead of increasing the process sensitivity label after a branch statement to the label of that branch statement because the branch statement may have altered variables

that appear after it, a security-type system proves that all possible alterations would not create insecure information flows. A security-type system can compile the program in Figure 3 because it has verified that regardless of the value of $s_1$, the **if** statement does not change the value of $p_2$ that is assigned to $p_1$. If the two branches of the **if** statement changed $p_2$ in different ways, then static analysis would reject the program.

Security-typed languages cover many cases of information flow analysis, but they are not a perfect solution. They can not handle variables whose security levels are not known at compile time, nor do they support multiple actors of different security levels observing the same variable and seeing different values. In some cases, such as in a game (like blackjack), it is desirable to have a variable appear one way to those with access to it and another way to the public.

## 2.3 Faceted Execution

Faceted execution is a system for dynamic information flow control that satisfies termination-insensitive non-interference while allowing programmers to program in a policy-agnostic manner.[5, p. 1] In contrast to information flow control type systems, which impose strict limitations on how private and public variables can interact, faceted execution allows any operation that can be performed on public data to be performed on any mix of public and private data through the same syntax. To accomplish this, faceted execution stores private data in faceted values, which manage access to private data at the time the data is used or observed. [2]

### 2.3.1 Faceted Values

A faceted value, written $\langle k \ ? \ s_1 \diamond p_1 \rangle$, is a decision tree containing a private facet $s_1$ and a public facet $p_1$ controlled by an access policy $k$.[5, p. 2]. The facets of a faceted value can take on any type of data that can be represented by a variable, including another faceted value. When viewed in a context with access to $k$, a faceted value $\langle k \ ? \ s_1 \diamond p_1 \rangle$ appears as $s_1$. Otherwise, it appears as $p_1$. Because of this, faceted values can be treated exactly like the data they store. Any operations that would be valid on a faceted value's stored data are valid on the faceted value itself.

### 2.3.2 Semantics and Correctness

In order to guarantee termination-insensitive non-interference, faceted execution must prevent information from leaking in explicit flows and implicit flows.

**Explicit Flows:** To prevent leaks from explicit flows, faceted execution branches when the result of an operation depends on a faceted value but the context in which the result will be viewed is not yet known. Each facet in the original faceted value is used in an isolated copy of the operation and placed into its corresponding location in a new faceted value that has the same access policy

as the original. This new faceted value is then returned as the result of the operation.[2, p. 2]

As an example, the operation

$$\langle k \ ? \ 32 \diamond 0 \rangle + 10$$

evaluates to

$$\langle k \ ? \ 42 \diamond 10 \rangle$$

by evaluating the branch private on $k$ as $42 = 32 + 10$ and the branch public on $k$ as $10 = 0 + 10$. We call the private branch a principal $k$ and the public branch a negation $\overline{k}$. [2, p. 5] When an observer with access to $k$ looks at the faceted value, it appears as 42. An observer without access to $k$ sees 10.

**Implicit Flows:** Leaks from implicit flows are similarly prevented by branching. When the flow of a program depends on a faceted value, both possible branches of execution are taken. For example, the implicit flow negation program in Figure 1 with input $s_1 = \langle k \ ? \ S_1 \diamond \text{false} \rangle$ (reproduced below)

$$p_1 := \text{true}$$
$$\textbf{if } s_1 \textbf{ then } p_1 := \text{false}$$

results in

$$p_1 := \langle k \ ? \ \neg S_1 \diamond \text{true} \rangle$$

by evaluating the **if** statement on the $k$ branch as

$$\textbf{if } S_1 \textbf{ then } p_1 := \text{false}$$

which reduces to $p_1 := \neg S_1$ and the $\overline{k}$ branch as

$$\textbf{if } \text{false} \textbf{ then } p_1 := \text{false}$$

which reduces to $p_1 := \text{true}$. [2, p. 2] When an observer views $p_1$, they see $\neg S_1$ if they have access to $k$ and "true" otherwise. In this way, faceted execution prevents implicit flows from leaking private data.

This branching method also allows faceted execution to correctly run programs that security-typed languages can not run. Consider Figure 4, a modified version of Figure 3 with secure input $s_1$ and public $p_1, p_2$:

$$p_1 := \text{true}$$
$$p_2 := \text{false}$$
$$\textbf{if } s_1 \textbf{ then } p_2 := \text{true}$$
$$p_1 := p_2$$

Figure 4: Rejected by Security-Type Languages

A security-type system would complain that $p_2$, a public variable, is being assigned in the context of an **if** statement that depends on the private variable $s_1$.

In faceted execution, the program would look the same, but the input $s_1$ would be packaged in a faceted value as $\langle k \ ? \ S_1 \diamond \bot \rangle$ (where $S_1$ is the raw input value of $s_1$). The **if** statement, which depends on $s_1$, would cause $p_2$ to become $\langle k \ ? \ S_1 \stackrel{?}{=} \text{true} \diamond \bot \rangle$, preserving (where "$\stackrel{?}{=}$" is the "check if equal" operator).

### 2.3.3 Program Counter

As detailed in 2.3.2, branching in faceted execution must occur when the context that the result of an operation will be viewed in is not yet known. Since branching is inefficient, it is desirable to branch as little as possible. In some cases, branching may be unnecessary. If it is known what context a faceted value (and all side effects) will be viewed in, it is sufficient to compute only the corresponding branch of the facet. The following program exhibits such a situation.

$$
\begin{aligned}
&z := \langle k \ ? \ \text{true} \diamond \text{false} \rangle \\
&a := \langle k \ ? \ 100 \diamond 2 \rangle \\
&\textbf{if } z \textbf{ then} \\
&\quad c := a + b \\
&\textbf{else} \\
&\quad c := a * b
\end{aligned}
$$

evaluates immediately to

$$c = \langle k \ ? \ 100 + b \diamond 2 * b \rangle$$

without going through the step of

$$c = \langle k \ ? \ \langle k \ ? \ 100 \diamond 2 \rangle + b \diamond \langle k \ ? \ 100 \diamond 2 \rangle * b \rangle$$

In this example, branching occurs at the faceted value $z$ but does not occur at $a$. When branching at $z$, one branch has access to $k$ and one does not, so when the faceted value $a$ is evaluated on one of these branches, the security context is already known to be $k$ or $\overline{k}$. Since all the information that $a$ depends on is known, $a$ takes on the value that corresponds to the security context instead of branching.[2, p. 6]

To keep track of the security context, we use a *program counter*, denoted *pc*, which we define to be a set of branch labels, each either a principal $k$ or a negation $\overline{k}$. [2, p. 5] As execution of the program branches on faceted values, the program counter is updated to reflect which principals or negations were taken. Then, when another faceted value is encountered, if its security policy is already a principal or negation in the program counter, branching does not

occur and only the corresponding facet is executed. To illustrate this, we repeat the previous example while tracking the program counter.

$$z := \langle k \ ? \ \text{true} \diamond \text{false} \rangle \qquad pc = \{\}$$
$$a := \langle k \ ? \ 100 \diamond 2 \rangle \qquad pc = \{\}$$

Branch on faceted value $z$

| | | | |
|---|---|---|---|
| **if** true **then** | $pc = \{k\}$ | **if** false **then** | $pc = \{\overline{k}\}$ |
| $c := 100 + b$ | $pc = \{k\}$ | $c := 2 + b$ | $pc = \{\overline{k}\}$ |
| **else** | | **else** | |
| $c := 100 * b$ | $pc = \{k\}$ | $c := 2 * b$ | $pc = \{\overline{k}\}$ |

Merge to form a new facet for every modified variable

$$c = \langle k \ ? \ 100 + b \diamond 2 * b \rangle$$

When execution reaches the if statement, the program counter is $\{\}$, which includes neither $k$ nor $\overline{k}$. Consequently, branching occurs on $z$ into the $k$ branch shown on the left (with $pc \leftarrow pc \cup \{k\}$) and the $\overline{k}$ branch shown on the right (with $pc \leftarrow pc \cup \{\overline{k}\}$).[2, p. 6 F-SPLIT] Each of these branches is then executed in the context of its program counter. When the second faceted value is reached in $c := a + b$ or $c := a * b$, the program counter already contains $k$ or $\overline{k}$, so $a$ becomes the value of one of its facets without further branching. [2, p. 6 F-LEFT, F-RIGHT]

### 2.3.4 Declassification

Declassification involves migrating information from one faceted value to another faceted value with a different access policy, usually in order to make that information more public.[2, p. 10] If this second faceted value's access policy is public to the world, then declassification is equivalant to migrating information from a faceted value to a normal variable. Declassification must be implemented in a careful manner to avoid invalidating all the security properties that faceted execution provides.

**Naive Declassification:** A simple way to declassify a faceted value $e$ given a label $k$ is to recursively cut out the public branch of any faceted values of label $k$ under $e$, leaving the private branch accessable to all observers. This process is expressed in the following algorithm.

**declassify**$(k,\ e)$ :
  **match** $e$ :
    **if** $\langle k \ ? \ V_1 \diamond V_2 \rangle$ **then declassify**$(k,\ V_1)$
    **if** $\langle l \ ? \ V_1 \diamond V_2 \rangle$ **then** $\langle l \ ?$ **declassify**$(k,\ V_1) \diamond$ **declassify**$(k,\ V_2) \rangle$
    **default:** $e$

When a faceted value $\langle s \ ? \ V_1 \diamond V_2 \rangle$ is declassified with label $k$, the result is

$V_1$. On the other hand, if the facet has label $l \neq k$ as in $\langle l \ ? \ V_1 \diamond V_2 \rangle$, it remains unchanged unless $V_1$ or $V_2$ contains a faceted value with policy $k$.

This simple declassification method has the potential to invalidate the security properties of faceted execution. If used incorrectly, it may allow an adversary to read the private branch of more faceted values than intended. Consider the following program.

$$\begin{aligned}
&\text{secret} := \langle s \ ? \ V_1 \diamond V_2 \rangle \\
&\text{toBeRevealed} := \langle s \ ? \ V_3 \diamond V_4 \rangle \\
&\text{untrusted} := \textbf{get-adversary-input}() \\
&\text{result} := \textbf{declassify}(s, \ \text{toBeRevealed} + \text{untrusted})
\end{aligned}$$

This program behaves as expected for "normal" inputs. toBeRevealed is added to untrusted to get $\langle s \ ? \ V_3 + \text{untrusted} \diamond V_4 + \text{untrusted} \rangle$, which is then declassified to $V_3 + \text{untrusted}$. This changes, however, if the adversary chooses "secret $-$ toBeRevealed" as their input. Now, toBeRevealed $+$ untrusted reduces to secret, which is then declassified! Even worse, the adversary can use the same technique to declassify any faceted value with label $s$. [2, p. 10]

**Robust Declassification:** A particularly attractive goal for declassification that fixes the above issue is robustness. Robustness stipulates that an *active attacker*, who can introduce code and influence inputs, is no more powerful than a *passive attacker*, who can only observe results. [2, p. 10].

Since a passive attacker has no inputs to the program, they can not influence what faceted values are declassified. To achieve robustness, an active attacker's inputs must likewise have no effect on what values are declassified.[2, p. 10] In realizing this requirement, it is useful to consider the different views of execution.

At this point, it is necessary to differentiate between the terms *principal* and *label*. A *label* is used by faceted values to determine which facet an observer is allowed to see. A *principal* $P$ is a set of labels that control access to data considered secret and a set of untrusted labels that do the same for data considered untrusted.[2, p. 10]

We consider one principal $P$ with two labels $S^P$ and $U^P$. The label $S^P$ labels data secret to $P$ that will be declassified, and the label $U^P$ labels data untrusted by $P$ (in our example, it is data that may have been modified by an attacker). A *view* is a set of labels, and corresponds to the set of information accessible given those labels. There are four views on $P$: $\{\}, \{S^P\}, \{U^P\}, \{S^P, U^P\}$, each an element of the power set of labels. The views that contain $U^P$ may have been affected by the attacker since data the attacker can write to is visible in them. On the other hand, the views without $U^P$ are free of the attacker's influence.[2, p. 10]

Each principal implements a function $\textbf{declassify}_{\mathbf{P}}$ that, when applied to a (potentially faceted) value, checks that the execution context does not include $U^P$ and then returns a value where anything previously visible only to observers with $S^P$ is now visible to any observer that does not have $U^P$.[2, p. 10,11]

Austin's implementation of $\mathbf{declassify_P}$ is shown below.[2, fig. 9]

$\mathbf{declassify_P}(e)$ :
  **match** $e$ :
    **if** $\langle S^P \ ? \ V_1 \diamond V_2 \rangle$ **then** $\langle U^P \ ? \ \langle S^P \ ? \ V_1 \diamond V_2 \rangle \diamond V_1 \rangle$
    **if** $\langle U^P \ ? \ V_1 \diamond V_2 \rangle$ **then** $\langle U^P \ ? \ V_1 \diamond \mathbf{declassify_P}(V_2) \rangle$
    **if** $\langle l \ ? \ V_1 \diamond V_2 \rangle$ **then** $\langle l \ ? \ \mathbf{declassify_P}(V_1) \diamond \mathbf{declassify_P}(V_2) \rangle$
    **default:** $e$

It is crucial that principals be defined before any untrusted data enters the program. Otherwise, an attacker might define a principal $A$ that trusts $U^P$ and has secret $S^P$ in order to obtain a declassification function $\mathbf{declassify_A}$ that works in the attacker's context and declassifies $S^P$.

While robust declassification preserves non-interference for views containing $U^P$, it may be too restrictive for certain applications as it disallows an untrusted source from ever learning anything about a secret value. A looser version of robust declassification that keeps the requirement that declassification only occur if the execution context is not influenced by the untrusted source but loosens the requirement that secret information not travel to the untrusted source might be desirable in some cases.

#### 2.3.5 Racets: Faceted Execution in Racket

Racets is a language extension of Racket (a language based on Scheme) that leverages Racket's macro system to support faceted execution.[5, p. 7] Racets differs from the prior definition of faceted execution in a few minor ways. It extends the definition of labels to be first class functions instead of symbols. All operations with faceted values remain the same. However, observing faceted values is different. Racets includes an observation function **(obs label label-arg value)** that reveals the branch of **value** private to **label** depending on whether **(label label-arg)** returns true or false.[5, p. 5 OBS]

# 3 Motivation

Industry programmers must make their code conform to the privacy policies that have been set in place by the companies they work for. Since these policies change frequently, a code base where information flow is easy to understand and control is indispensable.

Faceted execution provides a clear way to implement the information flow restrictions given by privacy policies into code. Programmers can implement the core logic of their task without worrying about how information flows through their implementation. Then, they can control information flow by placing sensitive information in faceted values and returning results that have been observed with the appropriate labels. Privacy policy changes can be implemented by

changing the labels attached to incoming information and the labels used to observe outgoing information without changing the core logic at all.

# 4 Our Approach

## 4.1 Client-side Buttons Declassify Faceted Values

To facilitate the realization of privacy policies on the web, we propose a web framework that ties client-side interactions to server-side use of faceted values. Clients may submit forms that contain private data, which would be stored in faceted values on the server. Through button presses on the client side, clients may choose to selectively declassify their private data.

## 4.2 Use Case

An example use case is in an academic grading server. A professor may not want students to be able to see their grades until some time after they have been entered online. In this scenario, the grades are stored in a faceted value private to the professor. Students can not observe their grade until the professor presses a button that declassifies the faceted value containing the grades.

## 4.3 Implementation in Racets

As a first step toward realizing correct declassification, we extend Racets with support for naive declassification via the **fac-force-declassify** macro. Additionally, we show an example of declassification by extending the Racets Battleship case study, which provided a web interface for two players to play battleship, with a declassification option that allows players to see each others boards.[1]

# 5 Conclusion

Faceted execution is a dynamic information flow control system that generalizes mandatory access control to achieve termination insensitive non-interference. Instead of using an information flow control type system to verify that no set of inputs could cause information to be leaked, it stores sensitive data in faceted values, denoted $\langle s \ ? \ V_1 \diamond V_2 \rangle$, which control at runtime how sensitive data can be accessed, meaning no programs are rejected as unsafe. Faceted values allow programmers to write code as they normally would without worrying about leaking private information because they handle all access checks at the time the results are observed or declassified. This means that in order for faceted execution to be correct, declassification must satisfy some degree of termination insensitive non-interference. Neither naive declassification nor robust declassification is a perfect solution, and declassification remains an open research topic.

[1] Code available at `https://github.com/mattsoulanille/racets/tree/declassify`

## 5.1 Future Work

Future work involves designing a version of declassification that sufficiently maintains non-interference while not being so restrictive as to prevent any information from being shared. Implementing the web framework described in 4.1 on top of an existing racket web framework is also a goal.

# References


[1] I. Allen, "The brutal lifecycle of javascript frameworks," https://stackoverflow.blog/2018/01/11/brutal-lifecycle-javascript-frameworks/, accessed: 2018-10-25.

[2] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *ACM Sigplan Notices*, vol. 47, no. 1. ACM, 2012, pp. 165–178, definition of Faceted Execution. Valuable exposition on how to read lambda expressions.

[3] A. M. A. Sabelfeld, "Language-based information-flow security," in *Selected Areas in Communications, Journal on*, vol. 21. IEEE, 2003, pp. 5–19.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *European symposium on research in computer security*. Springer, 2008, pp. 333–348, accessed: 2018-10-25.

[5] K. Micinski, Z. Wang, and T. Gilray, "Racets: Faceted execution in racket," *arXiv preprint arXiv:1807.09377*, 2018, faceted execution in Racket. Includes definition of faceted execution and implementation details.

[6] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 132–141, how to extend Racket and write new languages.

[7] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Security and privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 224–238.