Improving Information Flow Control Design with Security Contexts

by

Paul Wang Hemberger

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author . . . .
Department of Electrical Engineering and Computer Science
May 14, 2015

Certified by . . . .
Srini Devadas
Professor
Thesis Supervisor

Accepted by . . . .
Albert R. Meyer
Chairman, Department Committee on Graduate Theses

Improving Information Flow Control Design with Security Contexts

by

Paul Wang Hemberger

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis outlines a new language feature for Ruby: the security context, which enables complex information flow control schemes to be written in Ruby without modification to the virtual machine. Security contexts are Ruby objects that act as transparent proxies and can be attached to other objects, allowing them to seamlessly modify parameters and return values into and out of those objects' methods. Security contexts are demonstrated to be simple and effective in creating two flow control applications that would otherwise pose significant challenge to build: taint tracking as a Ruby library, and data flow assertions for Ruby on Rails applications. The performance of these systems was benchmarked while running as a part of a Rails application, and reached acceptable performance: taint tracking had no impact on performance, and data flow assertions saw a 50% throughput decrease, while providing considerable protection against privacy leaks and security vulnerabilities.

Thesis Supervisor: Srini Devadas
Title: Professor

Acknowledgments

I am immensely grateful for all of the support and guidance I've received from Prof. Srini Devadas. Srini has always been quick to help and push my work in the right direction. He taught me that once I have a hammer, I better find some nails. (I think I found a few.)

Of course, I am equally indebted to Victor Costan, my research mentor for the past three years. Since I began as a UROP, I have pinged Victor endlessly for his thoughts not just on research, but on programming practice, classes, careers and whatever is on my mind. No matter what I ask him, it seems that Victor somehow already has substantial experience in the subject, and he has been an invaluable resource and friend. He even helped me wade through my very first open source commit.

Working on my thesis was been a rollercoaster experience: hard oscillations between triumph (when my code started working) and despair (when I immediately uncovered a new error). I must acknowledge and thank all of my friends and family who have put up with me during this time. The M.Eng is not terribly long, but I am sure I made it feel like eternity for those around me.

To Andrés Romero and Kevin White, for being fantastic friends who convincingly pretended to enjoy proofreading my drafts. To Sam Peana, Vo Thanh Minh Tue, and Klaudia Leja, for being goofy, wonderful people. To my fellow M.Eng'ers: Esther Jang, Julián González, and Alex Gutierrez, for sharing ideas and commiserating with me.

To my parents, without whom I would not be here (chuckle). To Chris and Francesca, who bring infectious joy wherever they go. Thank you so much for your endless love and support.

And most of all, to Jessica Fujimori, who makes me a better person and inspires me every day. Thank you, thank you, thank you.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Language-level information flow control (IFC) is a powerful technique, and it has been repeatedly demonstrated in research to be an effective solution to entire classes of security vulnerabilities. But despite its track record, IFC has yet to make an impact on commercial software in any meaningful way. There are no libraries to easily add data flow assertions–which can automatically prevent many vulnerabilities and privacy leaks–into production web applications, which is a loss for both users and developers.

A key challenge is that there is no simple way to write IFC systems in the language of the developer's choice. IFC requires hooking into essential operations of the language's execution, which most languages cannot offer–they lack the *flow control structures* needed to write *flow control schemes*. Without the proper structure to build upon, implementing a scheme becomes a considerable undertaking, often with considerable drawbacks. Adding in the proper hooks then requires out-of-band solutions, like direct modifications to the language's virtual machine, bytecode manipulation or source-to-source transformation. Each is a complicated task, and conceptually distant from actually working on flow control; the desired flow control scheme might be simple, but building it requires expertise in unrelated areas. Even if one of these techniques yields a successful IFC application, the barrier to usage is high, as special transformations and changes to the toolchain make the application difficult to distribute and use. Impeding distribution and deployment will stymie even the most exciting work.

This thesis presents *security contexts*, language-level objects that offer a structure to build arbitrary flow control schemes. The security context offers a high-level perspective on flow control. It allows a scheme to be purely defined as a user-level application, and it has a sensible API that exposes the necessary hooks into the language's execution to build flow control systems. IFC can be written using security contexts, packaged into a library, and included into an application without hesitation. Their high-level design greatly lowers the barrier to prototyping flow control, and eliminates the difficulty of distribution. Complex ideas can be distilled down to simple and readable security contexts and tested quickly.

The central idea of the security context is its interposition on regular method calls with its *pre-hooks* and *post-hooks*, methods that wrap around another object's methods to analyze or modify the data going in and out. The hooks act transparently–a hooked method does not know of the security context's *pre-hooks* or *post-hooks*–and the hooks are written as ordinary language methods, requiring no special techniques. They have a consistent API that makes using them uncomplicated and predictable. Security contexts are designed to have no impact on existing code that does not use them.

To demonstrate the practicality and power of security contexts, we have built them into the Ruby language. Ruby is an expressive language that lends itself well to prototyping new ideas, and it is widely used for web applications, due to the popularity of the Ruby on Rails web framework. We specifically chose to use Rubinius, an implementation of Ruby built on top of a custom C++ virtual machine and LLVM JIT compiler. Rubinius is notable for implementing several core language features, as well as all of the Ruby standard library, in Ruby rather than in the virtual machine. This makes it a particularly good platform to experiment with, as its virtual machine is designed for close interaction with Ruby logic to fully define the language. We added security contexts into Rubinius with additions to the virtual machine and standard library.

We demonstrate that security contexts are simple and effective in building two different flow control schemes that normally require significant virtual machine modifications or other techniques: taint tracking and data flow assertions. We then evaluated our designs by applying these libraries to a Ruby on Rails web application, and measured application performance. We found that given demanding information flow control policies, our application reached acceptable performance, with an overhead of 0% to 47% depending on the benchmark. The code required to build these schemes was quite reasonable; taint tracking required only 200 lines of code.

1.1 Thesis Outline

Chapter 2 summarizes two different flow control schemes for securing applications that are unattainable without a proper flow control structure to support them. We illustrate their utility and the challenges posed in building them.

Chapter 3 describes the full design of security contexts, our proposed flow control structure.

Chapters 4 and 5 describes how security contexts were used to build the two schemes outlined in Chapter 2.

Chapter 6 describes the performance and security impacts of using security context-based applications within a production web application.

1.2 Related Flow Control Implementation Work

Significant research has been done on flow control schemes, particularly in the realm of security policy enforcement for web applications [8, 15, 16, 22, 24, 34, 36], and it has also been applied to operating systems [37], distributed systems [7], and browsers [29]. The implementation process of these efforts has varied, but is rarely simple. In this section we, summarize the different techniques used to achieve language-level IFC.

1.2.1 Virtual Machine Support

Languages with any form of flow control have typically had their structures and logic embedded into the language virtual machine. An example of this is seen in scripting languages, such as Perl, PHP and Ruby which support taint tracking. The virtual machine marks runtime objects as *tainted*, and contains all of the taint propagation rules within the virtual machine's codebase. This removes any flexibility for extension to taint tracking as a user of the language–change can only be achieved by modification to the language's virtual machine, not through the language itself.

1.2.2 Language Modifications

With the inherent limitations of most languages in providing facilities to build flow control schemes, naturally many researchers have taken to extending the languages to add in the features they need. This typically involves changes to the virtual machine [15, 16, 22, 24, 36]. Modifying the virtual machine is often a straightforward way to change program execution, but it increases the barrier to widespread adoption. Production software must be reliable, and running a system on top of a non-standard virtual machine is a hard sell. It is also more challenging to disseminate the work, as now users must install the new virtual machine and add it to their toolchain. It can also be difficult to guarantee correctness, Resin [36] referred to having to modify the PHP interpreter in 103 locations, which increases the risk of missing checks and adds inertia when changing the design.

1.2.3 Bytecode Hacking

Bytecode hacking is one solution for changing execution to support flow control without having to modify the virtual machine. The idea is to compile a program normally, take the bytecode of that program and modify it to add in hooks, and output new bytecode which supports the original program with the intended flow control scheme. This technique has been used with the Java Virtual Machine [8, 22]. One downside of bytecode hacking is that the running bytecode no longer exactly corresponds to the source code of the original program, which could make debugging and understanding the program's execution challenging. It also requires deep knowledge of the bytecode of the virtual machine, which could be a barrier for prototyping new ideas.

1.2.4 Source-to-Source Transformations

Source-to-source transformation aims to achieve a similar goal to bytecode hacking, though it takes place one step earlier. The source code of a program is transformed into source code with the necessary hooks for flow control, and then compiled into bytecode with the standard virtual machine. The downsides are similar: the final code no longer represents the source code of the original application, and writing an accurate source-to-source transformation is a non-trivial task. Source-to-source transformation was used in GuardRails [34], a Ruby on Rails policy system.

1.2.5 Static Analysis

Static analysis has been applied to a number of programming languages in the pursuit of guaranteeing flow control properties at compile time [11, 20]. It can be a powerful tool, though it has inherent limitations when applied to dynamic languages, where methods might not even exist until certain code paths are executed. A dynamic application, such as a website whose data policies depend on knowing the permissions of the active user, is therefore not a well-suited target for static analysis.

1.2.6 Monadic Types

There are some languages that natively support the type of structures that flow control schemes need. An example of this is Haskell [1], a pure, functional language which supports monads, structures that contain values and define rules for how those values interact with external values. While immensely powerful, Haskell and other ML-derived languages have struggled to reach a wide audience, perhaps do to their foreign nature compared to popular imperative and object-oriented languages. The vast difference in language functionality means that the power of Haskell's monads are not easily translated to an object-oriented language like Ruby. This is unfortunate, as incredible flow control work has been done with Haskell, such as Hails [14], a system for building untrusted web platforms. It uses types to attach policies to data, and the trusted runtime ensures that access control of those policies is always respected, even when running untrusted code.

1.2.7 Runtime Isolation

COWL [29] outlines a plan to bring *coarse-grained* information flow control to any language. With a coarse-grained approach, a program is segmented into isolated computational units, or *tasks*, which run with a specific security label. Tasks can communicate with each other via *send* and *receive* messages, and the IFC manager can impose restrictions on the message transfers based on the labels of each task. The segmenting of the program and communication via explicit channels is similar to that of processes and interprocess communication (IPC) in an operating system. Applying this technique at the language level is powerful; COWL is able to secure complex mashup web application that run both trusted code and untrusted 3rd-party code with minimal modification to the application. The downside is that the coarse-grained approach loses flexibility compared to a fine-grained approach, and applications may not be easy to partition in sensible ways.

1.3 Scope

One aspect of information flow control that we have not aimed to tackle are implicit flows. Implicit flows occur when tainted data is used to govern the control flow of an application. The branching that occurs off of tainted data can then disclose information about the execution of the program or be manipulated to take a different code path for a specific purpose. However, implicit flows are difficult to reason about, as the data flow is no longer direct, and can lead to *taint creep*, where an increasingly large portion of the application is tainted. This can quickly become unwieldy, and the taint of the execution may not correspond clearly to the developer's intended data flow.

1.4 Goals

With the guarantees that it can offer, information flow control belongs in production, everyday software, but has suffered from lack of built-in language support. We want to demonstrate that the right feature can reduce the high barrier to building flow control systems, and be performant and practical to use. We believe that we have identified one with the security context, as it has enabled us to prototype information flow control systems that would be otherwise overwhelming to design, and have demonstrated reasonable performance in a production web application. While the security context itself requires modifications to the virtual machine, we want to show that its construction offers a blueprint for an accessible and usable interface to information flow control design.

We furthermore hope that this work elucidates the distinction between a flow control scheme and structure. Significant research effort has been focused on exploring the capabilities of flow control schemes, but each effort must rehash the implementation process, which is rarely easy. But by focusing on the structure, it becomes fast and practical to experiment with new schemes. Most schemes thus far have been focused on security or privacy applications, but information flow control can benefit many more applications than just these–if more researchers and developers had access to languages with the right flow control structures, then more creative, more powerful and more diversified flow control applications can be explored.

Chapter 2

Overview of Flow Control Schemes

Before continuing on to the design of security contexts, which enable the practical implementation of flow control schemes, let us first briefly recap the functionality of two well-researched flow control schemes to understand their desired behavior and benefits.

2.1 Taint Tracking

Taint tracking is a long-standing feature built into languages like Perl, PHP and Ruby. With taint tracking, objects are either *untainted* or *tainted*. Objects by default are *untainted*. When *untainted* objects are combined with *tainted* objects, they become *tainted*. All values produced by a *tainted* object will also be *tainted*. The program can specifically *untaint*, or *sanitize* a value if it is deemed to be safe.

A common use case for taint tracking is to mark data coming from an untrusted source as *tainted*, and verify that sensitive operations throughout the application are not using *tainted* values. Because taint propagates, a taint check will catch if the value in question belongs anywhere in the entire *tainted data flow*, not just if it was the initial source of taint. This offers great flexibility, since the application can been written without thought of tainting, as taint automatically propagates regardless of the application's logic, and still be protected by it at runtime.

This type of protection, tainting a source and checking values at sensitive opera-

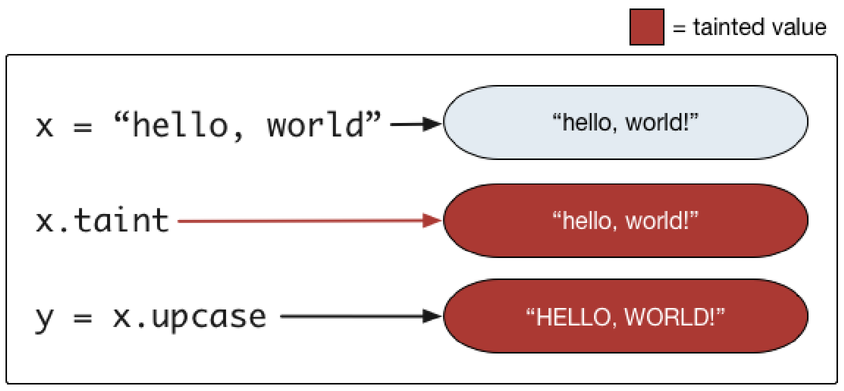

Figure 2-1: Taint flows from variable `x` to `y` automatically.

tions, is particularly effective in catching injection vulnerabilities. Many varieties of web vulnerabilities fall under this category, such as SQL injection, cross-site scripting and mass-assignment attacks, where malicious user data is processed by the application without sanitization. In SQL injection, user-supplied data is substituted into the application's database queries, allowing an adversary direct access to the database. Cross-site scripting occurs with user data reaches the HTML response of a web server without checks, allowing the adversary to distribute malicious scripts that run within the origin of the website. Mass-assignment attacks exploit how object-relational mapper systems update database entries, and allow an adversary to insert unintended attribute updates with modified request parameters. Clearly, executing any user-supplied data without sanitization can be regrettable. However, using taint tracking to mark user-supplied data as tainted, and the proper checks at the sensitive operations can prevent all of these vulnerabilities. Such techniques have been used to secure web applications before [8, 15, 24].

Despite its potential, taint tracking leaves something to be desired. As an object is either *tainted* or *untainted*, there is no way to know when or why an object was tainted, and it is difficult to differentiate how to sanitize or untaint different types of objects. Perhaps more pressing is that the overall semantics of the taint tracking system is not consolidated into one place, and it is instead expressed as checks throughout the application, and those checks as a whole define the system. Forgetting a single check at a critical junction in the application could be disastrous, and each check must know how to sanitize the data at that point in the code. A developer must then not only be conscious of the features she is building, but always be mindful of inserting taint checks at the proper places. As the application grows large, this becomes increasingly difficult to do correctly and consistently.

2.2 RESIN Data Flow Assertions

RESIN [36] details a much more powerful and thorough policy system using data flow assertions. Data flow assertions expand on taint tracking by allowing for more information to be propagated than the binary *untainted* or *tainted* value.

With data flow assertions, on object can be annotated with a *policy object*, which itself can contain code or metadata. In contrast to tainting, *policy objects* are language-level objects, so they can contain arbitrary data, such as marking a value as sensitive and where it came from. The *policy object* then propagates with the data flow of its host object, much like how *taint* propagates. RESIN then has *filter objects*, which are objects that demarcate data flow boundaries–points at which sensitive data might be stored or sent out on the network, or where user-supplied data might be executed within the program. *Filter objects* situate themselves on I/O channels and function calls, and check data for *policy objects*. If any data is annotated with *policy objects*, then the policies on those objects are enforced by the *filter object*.

This system allows for remarkable protection against programmer error and malicious attacks. Sensitive data in a web application can be annotated with *policy objects* that contain highly specific policies pertaining to their contents. For instance, a password could be annotated with a policy that says it can only be sent to a user with a particular email address. After that data propagates throughout the application, the *filter object* can examine the contents of the response, see its *policy object*, and check if the intended recipient's email address matches that of the policy. If not, the web server can send an error message instead of disclosing sensitive information. If it does match, then the response can continue as it would normally. The granularity with which data flow assertions can operate is a significant improvement to simple taint tracking.

Resin takes considerable care to prevent a number of distinct attack vectors. By labeling sensitive user data with *policy objects*, Resin is able to automatically enforce access control checks site-wide, which mitigates programmer errors that lead to information disclosure and attacks like directory traversal. By labeling untrusted data, it is able to prevent the execution of malicious data in attacks such as SQL injection and cross-site scripting. The Open Web Application Security Project, known as OWASP, regularly publishes a Top 10 list of the most common web application vulnerabilities [35]. Against the most recent list, Resin outright prevents five classes of vulnerabilities, and helps alleviate the threat from two others.

Data flow assertions scale well across increasingly large applications, where it becomes difficult to coordinate cross-cutting data policies reliably. In a large application, many features may make use of the same data in different ways. If that data is sensitive, then each feature must take care to use proper access control. This means continually repeating ad-hoc policy checks in increasingly separated parts of the application. Unfortunately, even the best, security-conscious programmers can make mistakes, and it seems inevitable that the data will be leaked. Resin solves this challenge as the data and their policies become inextricably attached. As long as the policies are reasonable, and the *filter objects* check in the right places, then data cannot be disclosed to the wrong users. This removes a significant burden from the programmer, and can greatly increase overall security.

The primary aspect of taint tracking and Resin-like data flow assertions that makes them hard to implement is the propagation of taint and policy objects. Propagating these values requires executing propagation logic as the program runs–the application source code should not have to specify any explicit propagation calls. Propagation thus becomes an aspect of the language runtime's execution, which most languages cannot hook into. Taint tracking gets around this limitation by placing the propagation logic into the virtual machine directly, so that it is pre-baked into the language runtime. Resin required significant virtual machine changes to reach a similar effect with its policy objects.

Chapter 3

Security Contexts

Information flow control can enforce invariants or policies about an application's runtime execution without having to interfere with any of the application's logic. It can be thought of as a shadow computation, or type of metaprogram that executes in the background as a part of the language runtime. Security contexts provide a hook into the language runtime that allow a developer to write custom methods that operate behind-the-scenes of regular program execution. This allows complex IFC to be written in the same way as a normal program; security contexts are a re-useable structure that can be fitted to numerous flow control schemes.

3.1 Identifying a Re-Useable Structure

To build a re-useable flow control structure, we first must identify what exactly a flow control scheme needs to operate. In a program, data flows between variables via function calls and statements. A flow control scheme wants to enact certain properties about a program's data flows, so it must have hooks into these operations to control or track the flow.

Working with Ruby simplifies this task: all statements are expressions, every value in Ruby is an object, and consequently all functions are method calls on objects. This makes method calls the only data flow operation in Ruby. Hooking into Ruby's data flow is then a matter of hooking into method calls.

One existing template for hooking into method calls in a dynamic language is ECMAScript 6's notion of a Proxy class [23]. The security context is inspired by the Proxy, which offers a simple API to wrap an object and hook into its methods.

3.2 Design

The layout of security contexts is simple: they are Ruby objects, any Ruby object can have a security context object, and a security context may define *pre-hooks* and *post-hooks*, which wrap an object's method calls. The hooks are written in Ruby, exposing the flow control scheme definition to a high level. An object with a security context attached is known as a *contextualized object*, and a security context can use its *pre-hooks* and *post-hooks* to control contextualized objects' methods and data flow. This is all the security context needs to offer–though the scale of its features are minimal, this functionality enables powerful applications that would be otherwise challenging to write and require extensive metaprogramming.

Practically, a contextualized Ruby object has a `#security_context` property that points to a security context object, which then wraps its methods. On this level, it appears that an object *has* a security context. However, conceptually it may be enlightening to see this as the object executing *within* a particular context. The exact nature of that context is defined by the security context, and varies with the intended flow control scheme.

3.2.1 Ruby Objects

An important aspect of the security context is that it is a Ruby object, so using it requires normal Ruby code. Allowing a developer to use the same language and runtime for both the IFC and target application enormously simplifies the task of implementing flow control. This is in strong contrast to many of the techniques that were outline in Section 1.2.

3.2.2 Method Hooks

The power of the security context lies in its *pre-hooks* and *post-hooks*. This is the important mechanism that allows a security context to trace the execution of an application. When an object executes a method, the Ruby virtual machine searches through the object's hierarchy to find a method by that name. Then it executes that method, with the object as the receiver. *Pre-hooks* and *post-hooks* surround this process. When a method is called on a *contextualized* object, the VM locates the relevant method in the object's hierarchy, then it instead executes the *pre-hook* of the security context, executes the original method with the original object as a receiver, and then executes the *post-hook* of the security context. Security contexts can therefore monitor messages passed into and out of an object, and can contextualize the returned objects, allowing a security context to expand and cover all objects within a particular data flow.

`before_method(receiver, *args, &block)`
the *pre-hook* receives the arguments to the original method and returns new (or the same) arguments to be used instead, the hooked arguments.
`after_method(receiving_object, return_value, *hooked_args)`
the *post-hook* receives the return value of the original method and returns a new one, the hooked return value. It also receives the hooked arguments that went into the original method.

Table 3.1: The API for security contexts' *pre-hooks* and *post-hooks*

A *pre-hook* is a Ruby method given three arguments: the receiver of the original method call, the arguments intended for that method, and the block intended for that method. It can then perform any logic to transform or record the arguments and block, and then return the hooked arguments that will instead reach the original method.[1]

A *post-hook* is also a Ruby method given three arguments: the receiver of the method call, the return value from the method call, and the hooked arguments from

[1]To prevent potential ambiguities when passing modified blocks to the original method, which is outside of Ruby's syntax, they must be given at the end of the list of return values, and marked with a special `hooked_block?` property. In practice, we never had to use `hooked_block` when using security contexts.

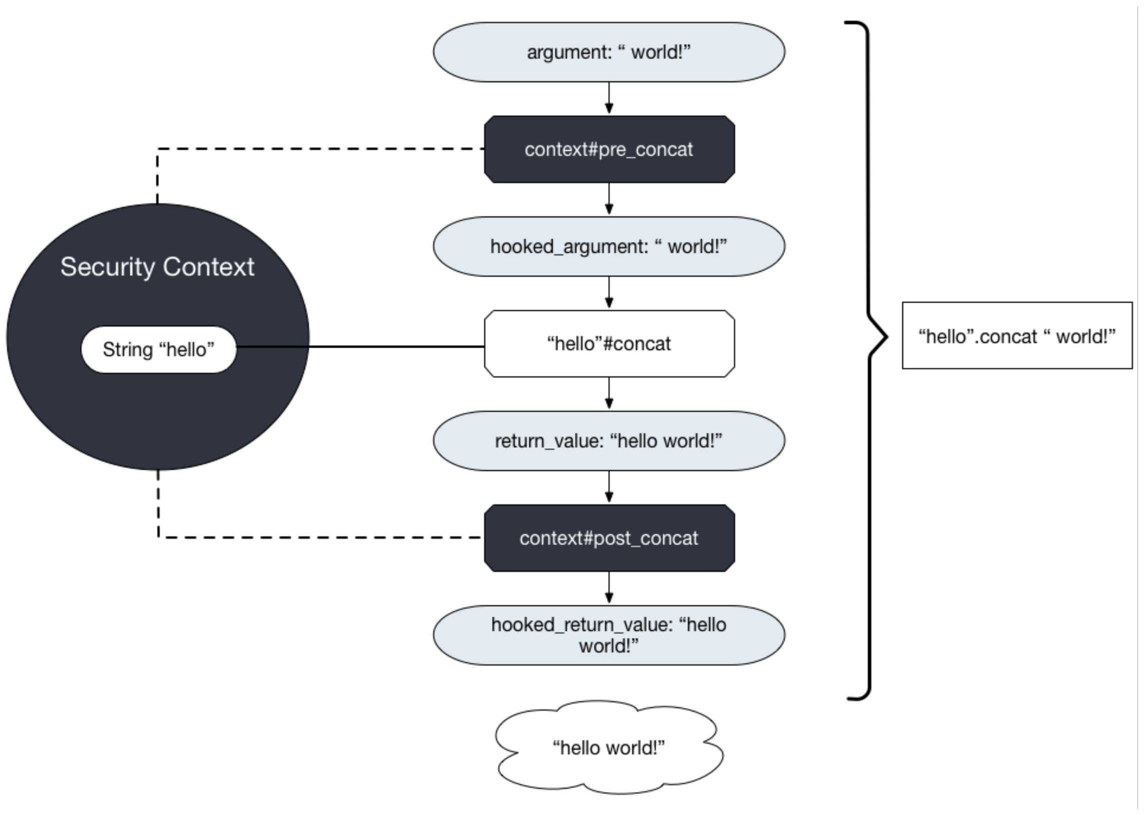

Figure 3-1: The `String` "hello" calling its `#concat` method when it has a security context. The security context executes its *pre-hook* `#pre_concat` and *post-hook* `#post_concat` before and after the method call, which can change the arguments into and return value from `"hello"#concat`. Here, the hook methods transparently pass the values on.

the accompanying *pre-hook*, if available. Similarly, it can transform or record any values, and return a value that stands in (or is the same as) as the return value from the original method call.

To clearly spell out the terminology used with hooks: the *pre-hook* receives the arguments intended for the method call, and returns the arguments that the method call will actually receive, called the *hooked arguments*. The method receives the *hooked arguments*, and emits the return value that the *post-hook* receives. The *post-hook* then returns a *hooked return value*, which is the value returned.

If a security context does not define any hooks on a contextualized object's method, then the method will execute normally. This minimizes the performance hit of code execution outside of a context.

Writing a hook is straightforward: if a method were named *#foo*, then the *pre-hook* method is named *#before_foo*, and the *post-hook* named *#after_foo*[2]. A security context can define *pre-hooks* and *post-hooks* on any or all methods.

3.3 Implementation

3.3.1 Virtual Machine Modifications

Several modifications to the Rubinius virtual machine were made to add security contexts into the language. We defined security contexts to be a property of Ruby objects much in the same way an object can be tainted, and then modified method execution to call *pre-hooks* and *post-hooks* when the receiver is contextualized.

Object Header

The Rubinius VM keeps a 32-bit header for every Ruby object. The header's bits are used to denote various properties and metadata of that object, such as whether or not the object is frozen, tainted or trusted, as well as tags for the garbage collector. Fortunately, the header did not make use of all 32-bits, so one bit was allocated to mark whether or not an object has a security context.

Primitives

Rubinius couples its Ruby-defined language features and standard library to its virtual machine through *primitives*. When a Ruby method calls a primitive, the interpreter will generate bytecode that tells the virtual machine to directly call one of its own functions to operate on the data on the stack. This allows Ruby code to dip into the virtual machine for certain operations that its better suited for, such as object allocation, deep copying, and fast string building methods.

[2]There exists one caveat when defining *pre-hooks* and *post-hooks* that involve Ruby's non-alphanumeric method names. Methods named `*`, `[]=`, `+@` are valid, but are not valid when combined with alphanumeric characters (e.g. `after_[]=` is an invalid name). For this reason, the *pre-hooks* and *post-hooks* involving these methods are translated into alphanumeric versions, such as `after_*` becoming `after_op__multiply`.

Fundamental properties of objects, such as frozen and taint, are written with primitives that execute VM code that finds the header bits on the internal object and returns a `Boolean`. We added similar primitives to find whether or not an object has a security context. Because a security context is a Ruby object and cannot fit into the header, we also made a special instance variable to store the security context, and primitives to access its contents.

Call Sites

The virtual machine's `Call Site` class handles the actual execution call of a Ruby method. This class was modified to check if the receiver of the method had a security context. If so, it would then check if it had a *pre-hook* or *post-hook* for the method, by doing method lookups on the security context object. It then handled the calls to those hooks by calling `#send` on the security context, with the arguments specified in Section 3.2.2.

3.3.2 String Interpolation

One challenge we encountered was properly handling Ruby string interpolation. This was similarly a sticking point for the developers of SafeWeb, a Ruby policy library [16]. String interpolation happens when the `String` result of a Ruby expression is executed and substituted–interpolated–into a new `String`. Rubinius natively builds the logic for this function into the virtual machine, and when it compiles a Ruby program, generates bytecode that directly calls this VM function. Because security contexts wrap around Ruby methods, not virtual machine methods, they cannot intercept the results from the Ruby expressions that are executed during the interpolation process. Unfortunately, this means that the resulting Strings escape the scope of the security context, so security context cannot track this important data flow.

```ruby
w = "world!".taint
"Hello #{w}"
```

Listing 3.1: Ruby snippet that taints a `String` and interpolates it into a new `String`.

The result should also be tainted.

To address this, we modified Rubinius's bytecode generation to instead call a string builder method written in Ruby when doing interpolation. When this operation is moved into Ruby, security contexts can hook the method and handle the data flow properly. A bytecode comparison can be seen in Listings A.1 and A.2. Not surprisingly, moving string building from the virtual machine to Ruby is costly in terms of performance, and is detailed in Section 6.1.3.

3.3.3 Ruby Standard Library

```
x = "hello, world"
y = x.match(/hello/.taint)
=> "hello"
y.tainted?
=> true
```

Listing 3.2: Using a *tainted* argument in an *untainted* object's method call can taint the output.

Security contexts hook into the method calls of a contextualized object, so in order for the *pre-hooks* and *post-hooks* to run, the contextualized object must be the method receiver. This brings up one challenging task of flow control in Ruby: handling the case where a method's *argument* is a contextualized object, but *the receiver of the method call is not.* To be able to fully cover all data flow paths, the security context of the contextualized object–even as an argument to a method–must be able to hook the method call. Listing 3.2 illustrates this scenario with taint tracking: a tainted regular expression is used to match text out of an untainted `String`. The `String` is the receiver, and is not tainted (and would not have a security context), but the regular expression is tainted and is passed in as an argument. Because the result might be the same as the tainted argument depending on the match, the result should be tainted, too.

```ruby
module Rubinius
  module Type
    class << self
      alias_method :old_infect, :infect
      def infect(host, source)
        if source.secure_context?
          source.secure_context.infect host, source
        end
        old_infect(host, source)
      end
    end
  end
end
```

Listing 3.3: Aliasing the `Rubinius::Type#infect` method to propagate the scheme defined by the security context

```ruby
module Police
  module DataFlow
    class SecureContext
      def infect(other, source)
        source.propagate_labels other
      end
    end
  end
end
```

Listing 3.4: POLICE's security context `#infect` method propagates the labels of the object onto the receiver of the method. It will be called each time the snippet in Listing 3.3 is called.

To handle this case, we hooked into Rubinius's `#infect` method. `#infect` is the method used to transfer taint and trust between objects. Rubinius has two variants of `#infect`, one that exists in the virtual machine, and one that exists in the Ruby bootstrapping code. The latter is just a primitive over the virtual machine code, but it offers us the ability to hook into its call as it is written in Ruby.

Rubinius already places calls to `#infect` throughout its standard library at all the points where taint or trust might propagate. We extended the method to call `#infect` on a security context, if source of the infection has one. The security context can then define its own `#infect` that defines how to propagate itself when one of its contextualized objects is used as an argument to a method. Had we not had the luxury of Rubinius's `#infect` to piggyback, we could have added a call to a security context's `#infect` method in the `Call Site` when a contextualized object is passed as an argument, to achieve the same effect.

Listing 3.3 shows how the built-in Rubinius `#infect` is extended to call `#infect` on a security context, and Listing 3.4 shows how POLICE, our data flow assertions library for Ruby on Rails applications, used this method to propagate its labels.

Chapter 4

Prototyping Taint Tracking

We reimplemented Ruby's taint tracking behavior as a first test of security contexts as a re-useable flow control structure. We mimicked the same propagation rules as native tainting, and satisfied the language tests for tainting defined by RubySpec. The performance of this application is detailed in Section 6.1.3.

4.1 Security Context Design

Taint tracking propagates *taint* within a data flow. To track a data flow with security contexts, one can use its *post-hooks* and `#infect` methods. An object can be tainted by giving it a security context. The security context then hooks into all of that object's methods, and if one of those methods can propagate taint, then the security context will attach itself to the resulting values. If that new object propagates taint, then yet again, the security context will hook its methods and attach itself to the new values. In this way, a security context can model the behavior of taint tracking by tracking data flow, with its presence on an object standing in for taint.

Successfully building track tracking from the ground up requires understanding all of the possible flows of data where taint might propagate. As discussed in Section 3.1, in Ruby this means knowing the method calls that propagate taint among the basic data types. If all of the basic types support propagation, then any developer-defined object will support it appropriately as well, since those objects will simply be a composition of the basic types. In particular, the Ruby `String` has numerous methods that propagate taint and must be considered carefully.

Fortunately, we have a template for Ruby's propagation rules: Rubinius's native taint tracking. We searched through the Rubinius VM and its Ruby standard library to find all of the methods that handle tainting or propagation of taint. We then wrote a security context that defines *post-hooks* for each of those methods. The *post-hooks* take in the return values from those methods, taint the results according to Ruby's propagation rules, put those values into the same security context, and pass the values back. We did not need *pre-hooks* because the arguments into the methods should not be modified with taint checking.

```ruby
class TaintSecurityContext
  def after_upcase(obj, retval, hooked_args)
    retval.taint
    retval.security_context = self
  end
end

x = "hello, world"
y = x.upcase
=> "HELLO, WORLD"
y.tainted?
=> false

x.security_context = TaintSecurityContext.new

y = x.upcase
=> "HELLO, WORLD"
y.tainted?
=>true
```

Listing 4.1: The `TaintSecurityContext` defines a *post-hook* to propagate *taint* after `String#upcase`. The variable `x` starts without a security context, so the variable `y` which results from `x#upcase` is not tainted. However when `x` executes within `TaintSecurityContext`, taint from `x#upcase` propagates to `y`, and `y` is also covered by the security context.

The resulting code is pleasantly simple. Listing 4.1 illustrates one of these tainting *post-hooks* for the Ruby method `String#upcase`, which returns a `String`'s value in all upper-case letters. The intent is clear from the security context definition, and furthermore, as the security context is extended to support all of Ruby's taint propagation rules, all of the logic is contained within a single class. This is in stark contrast to the current scattering of taint checks in Rubinius, which hovers at 52 explicit method calls for tainting in the standard library, and 18 in the virtual machine–the rules for tainting are difficult to piece together when they are so widely dispersed. Listing A.3 contains the full tainting security context we developed, and contains a near-perfect reproduction of Ruby's native taint tracking.

To actively use our replacement taint tracking in Ruby, we redefined `Kernel#taint` and related methods to use our security contexts. `Kernel` is a `Module` near the top of Ruby's object hierarchy, so nearly all runtime objects will contains its methods. Ruby allows for this type of "monkey-patching" where core objects can be redefined at runtime, which made our own code readily runnable.

4.2 RubySpec Compatibility

RubySpec is a project formerly maintained by the Rubinius team that serves as a living specification for the Ruby language [28]. It contains thousands of tests that in total constitute and verify the behavior of a correct implementation of Ruby. RubySpec serves as an excellent barometer for functional correctness when making changes to the language, and especially when trying to emulate existing behavior with a new feature.

Using our modified Ruby, known as RBX-SC, with the tainting methods monkey-patched to use the `TaintSecurityContext`, we verified that our implementation passed all of the relevant RubySpec tests pertaining to tainting. There were a few corner cases where our design propagated tainted where native Ruby would not, caused by the internals of how Rubinius distributes of logic across the C++ virtual machine and Ruby standard library. These could be changed if necessary, however the propagation rules it did exhibit seemed reasonable and so no further changes were made.

Chapter 5

Prototyping Police

The next system we prototyped was Police, a security framework that brings data flow assertions to Ruby on Rails applications. The design is modeled after Resin, but takes advantage of the rigid structure available in Ruby on Rails applications to decrease the cost of adding it into an existing application.

To build and test Police, we used *Seven*, a Ruby on Rails application used by MIT's Introduction to Algorithms course to manage homework submissions, grading, generating grade reports and more. It is a complex application with many components, and constantly moves sensitive data. This makes it an excellent real-world candidate for taint checking and policy enforcement.

Testing the correctness of Police involved building a suite of unit tests, and then running it with *Seven* to verify that was compatible with a full-fledged application. Adding Police to *Seven* took under 5 lines of code, and could be automatic with a few short changes. The efficacy of Police's policy enforcement is discussed in Section 6.2.

Section 5.1 summarizes the layout of a Rails application, for those who might be unfamiliar.

Section 5.2 details the design of Police.

5.1 Overview of Rails Application Architecture

Ruby on Rails is a successful framework for building web applications. It takes a heavy-weight approach, bundling a number of complex features together such as template rendering, routing, and an object-relational mapper (or ORM). The majority of a Rails application is built behind the scenes for the developer. Rails' opinionated and regular design makes extending it to support POLICE straightforward, since all applications made with it share the same overall structure.

5.1.1 Model-View-Controller

Ruby on Rails adopts the Model-View-Controller design pattern for building applications. This design pattern splits the task of building an application with a user interface into three discrete components. For a web application, the breakdown of responsibilities often comes down to:

- The model defines the data structures used by the application.
- The view defines the visual layout of a data structure or visual element.
- The controller receives commands and carries out the desired operation. Often this will involve logic to retrieve data from a model or many models, pass that data into a view, and return the HTML output.

Rails is built up from many packages. *ActiveModel* is Rails' implementation of the Model components, and in a default application it closely works with *ActiveRecord*, the ORM. *ActionView* contains the templating logic for generating views, and *ActionPack* contains the Controller definitions.

5.1.2 Rack Middleware

Rails operates on top of Rack, which is an API that connects web servers to frameworks. It handles incoming request data into the Rails app, and outgoing response

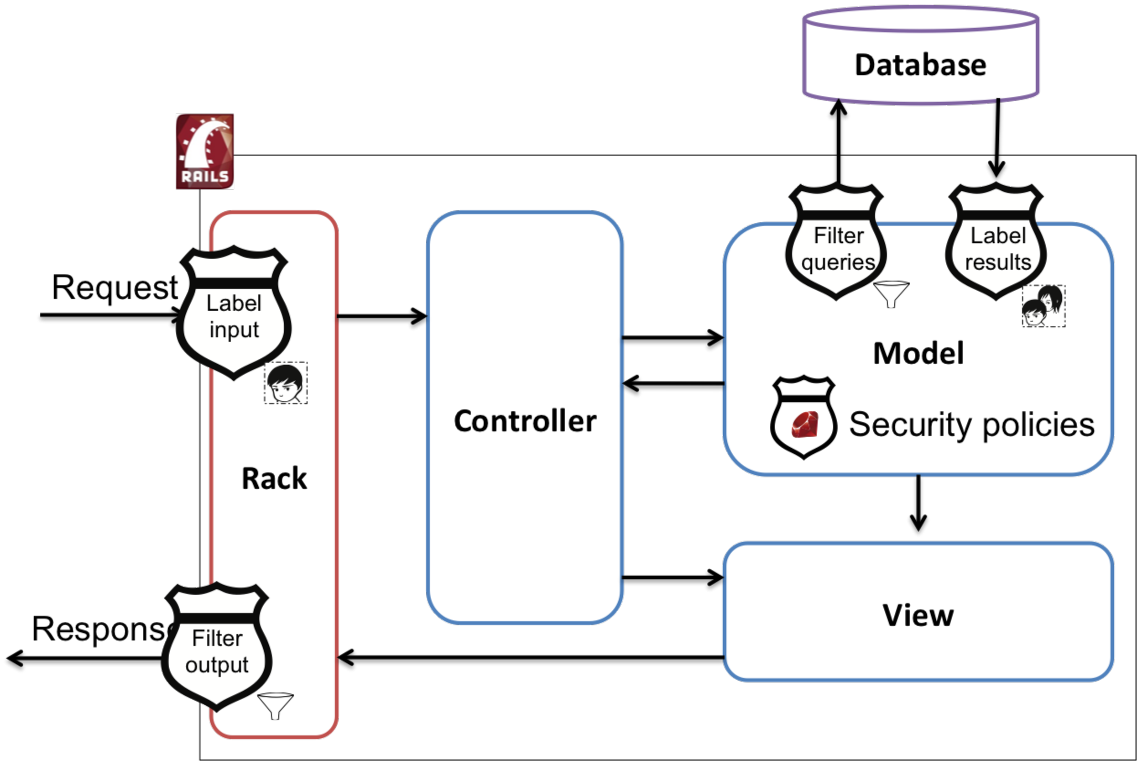

Figure 5-1: An overview of how POLICE fits into a Rails application. The policies are defined on the model, and labeling and filtering is done at the data boundaries, between the application and the network and database. Label propagation ensures that labels flow all throughout the application logic.

data produced by the Rails app. It supports *middleware*, which are applications that hook into this request processing cycle.

Figure 5-1 presents a full picture of the structure of a Rails application. Requests come in through Rack, the controller retrieves data from the models and persistent storage, gives it to the view, and outputs the page through Rack. This design is standard across all Rails applications.

5.2 Extensions to Rails

POLICE consists of three central pieces. The first is a policy domain-specific language to define security and privacy policies for the application's data. The second is filters that label incoming data with those policies and verify the policies of outgoing data. The third is the label propagation logic that allows labels to follow the data flow of the application.

5.2.1 ActiveRecord Domain-Specific Language

Domain-specific languages (DSLs) expose simple interfaces to complex logic. Ruby is well suited for defining domain-specific languages, and Rails and its ecosystem has leveraged this functionality significantly. *ActiveRecord*'s data validations [25], used to specify the proper format of a model's data, and CanCan [2], used to define authorization checks on model attributes, both define properties on models through DSLs. The DSL for POLICE is inspired by these libraries, in an effort to make it as simple as possible to use, and familiar to current developers.

POLICE extends *ActiveRecord*, the object-relational mapping layer of Rails with class methods to define policies on models. An individual policy offers either `:read` and `:write` protection over individual attributes of each model. `:read` policies are used to prevent unintended information disclosure, and `:write` policies can prevent unauthorized or inappropriate writes to the database. `:write` policies are sufficient to prevent mass-assignment attacks, which plagued Rails for years before *ActiveRecord* was fully patched [4, 19].

A policy is defined by a set of attributes on the model to protect, the action to protect (either `:read` or `:write`) and an anonymous function that contains the policy check. When invoked, the anonymous function is given a reference to the current user, so that the attribute can enforce access control.

```ruby
class Profile < ActiveRecord::Base
  police :name, :university, :department, :write => (lambda do |this, user|
    this.user == user or user.admin?
  end)
end
```

Listing 5.1: A policy on the `Profile` model that says that the only users allowed to update a particular user's name, university and department information are admins and that user. The variable `this` refers to the instance of the `Profile` being used, and `user` is the user of the current session

The DSL also provides support for nested relationships of models. Rails model relations can quickly become complex, and it can be useful to specify a policy on a nested attribute. Listing 5.2 illustrates this case with models from *Seven*: a homework `Submission` has an owner, which is a `User`, and can have a `HomeworkFile`. However a `HomeworkFile` is simply a blob data type that does not know which `Submission` it belongs to. There should be a policy that no students besides the one who submitted the file should be able to read its contents. This policy should belong to `Submission`, since it is the model that connects a `User` who submitted the file to the contents of the homework file. The contents of the homework file is not a simple attribute of `Submission`, so POLICE allows it to be specified through a list of attributes and be protected in the same way as if it were one of `Submission`'s own direct attributes.

```ruby
class Submission < ActiveRecord::Base
  police [:db_file, :f, :file_contents], :read => (lambda do |this, user|
    user && (this.is_owner?(user) || user.admin?)
  end)
end
```

Listing 5.2: A policy that protects the contents of a `Submission`'s homework file from being read by anyone but the user who submitted the file or an admin.

5.2.2 Filtering Middleware

POLICE's filters sit at the boundaries of a Rails application to label incoming data and prevent the export of policy-protected data. The filters reside in Rack middleware and in hooks to *ActiveRecord*.

POLICE's Rack middleware labels all request data, such as GET parameters, the querystring and form data, with a `UserSupplied` label. The label, a Ruby object, at first contains a `nil` reference to the active user's `User` model, as the middleware does not yet know which user the active session belongs to. POLICE then attaches a lambda method to the `RackEnvironment`, a hash map of Rack's state, that when called, sets the `UserSupplied` label to refer to a given `User`.

In a Rails application, the controller is the only module that knows the current user. Because the middleware labeled request data with a `UserSupplied` label, but did not know the current user, the controller is then responsible for setting the `User` on the labels. This is accomplished through POLICE's module to extend `ApplicationController`, the parent controller to all other controllers. `ApplicationController` executes its code before any custom controller for the application, so POLICE adds in a snippet that sets the `User` of the `UserSupplied` labels to that of the current session by calling the lambda method attached to the `RackEnvironment`. In this way, `UserSupplied` labels can freely propagate after the middleware filter, before the current user is even known, and once the user is known, the labels will all refer to the correct user.

With these steps, all request data will be labeled as coming from the current user once execution reaches the controllers. There are then two important cases that must be handled: writing data to the database, and returning an HTTP response. These are durable actions that must honor the labels of the data they process–failure to do so could lead to absent access control and myriads of problems.

POLICE uses *ActiveRecord*'s `:before_save` and `:before_update` hooks to enforce `:write` policies before writing to the database. POLICE will look through all of the labels attached to the model instance that is about to be saved. If there are any `UserSupplied` labels, it can then extract the `User` associated with the current request. It will then invoke the policy methods specified for the model, passing them the current `User` so that the policies can perform access control checks for the user. If all policies are satisfied at this point, *ActiveRecord* can continue on with its database save or update. If there are policy failures, the operation is rolled back and the application returns an error.

`:read` policies are enforced before returning an HTTP response by using *ActiveRecord*'s `:after_initialize` hook. `:after_initialize` executes after a model is instantiated, and POLICE then labels each of its policy-protected fields with `ReadRestriction` labels with a reference to the model's class. These labels will flow throughout the remainder of the Rails application as it generates a response.

The POLICE Rack middleware will eventually receive the response's contents, and check if any of it contains `ReadRestriction` labels. If it finds any, it looks up which models they came from, and therefore which policies must be enforced before the data exits the application. It can then execute the policy checks, and appropriately continue with or deny the response.

5.3 Label Propagation

Label propagation is the key idea to enforcing `:read` and `:write` policies without explicit checks. Once an object is labeled, all derived values will be too, and eventually if any of them reach a data boundary, filters will verify that the data satisfies the application's policies.

With POLICE, when an object is labeled, it is given a security context. The security context transparently propagates labels and itself to new objects, so that labels follow data flow. The propagation rules used by the security context are the same as those for tainting, so few modifications had to be made between `Police`'s security context and the taint checking security context.

POLICE defines new convenience methods on `Object`, such as `:label_with`, `:labeled?` and `:has_label?`. In usage, the security contexts are invisible to a user of POLICE.

The security contexts provide the label propagation, but never need to be directly acted upon. This helps separate the logic of labels and policy control from the mechanism to propagate them between objects.

Chapter 6

Evaluation

6.1 Performance Benchmarks

We benchmarked the performance of our security context-based taint tracking and data flow assertions with *Seven*. For taint tracking, we wrote a *tainting middleware*, that tainted all user-supplied values in the incoming HTTP request. For the data flow assertions, we installed POLICE into our application and wrote several policies for the application's data models.

For each test, we used ApacheBench to send 10,000 requests to the application, and recorded the average throughput and latency. The benchmarks were run on an Intel i7-3720QM with 8GB memory.

6.1.1 Requests

We evaluated our application against the following requests:

- A GET to the login page of *Seven*, which contains a simple login form without any particular data policies.
- A GET to the home page after logging in. The home page contains a newsfeed of recent homework uploads, posted assignments and who posted them, and upcoming deadlines, all of which is data that must be loaded dynamically from

the database and then interpolated into HTML templates. This makes it a computationally intensive process, in contrast to loading the login page.

- A POST to change a user's profile information. This includes request data from the user, which must propagate taint or labels if available.

- A GET to a user-owned homework file, which after passing access control checks, leads to a direct download. With Police, this tests the overhead of the *ActiveRecord* labeling and filtering for `:read` restrictions.

6.1.2 Police Policies

When testing Police, we wrote three policies for the application to protect a student's data:

- Restricting uploaded homework files to only be readable by the student or an administrator.

- Restricting an assignment's grading to only be readable by the student or an administrator.

- Protecting a user's profile information to only be changeable by that user.

6.1.3 Results

	RBX	RBX-SC	RBX-SC Native Taint	RBX-SC SC Taint	RBX-SC Police
Login Page	15.61	11.12	11.83	11.19	11.27
Submission	23.63	17.76	17.73	18.46	9.33
Profile	5.73	5.33	5.93	5.79	4.20
Home	2.48	1.77	1.88	1.89	1.59

Table 6.1: Requests per second for each type of request sent to *Seven*, averaged over 10,000 requests.

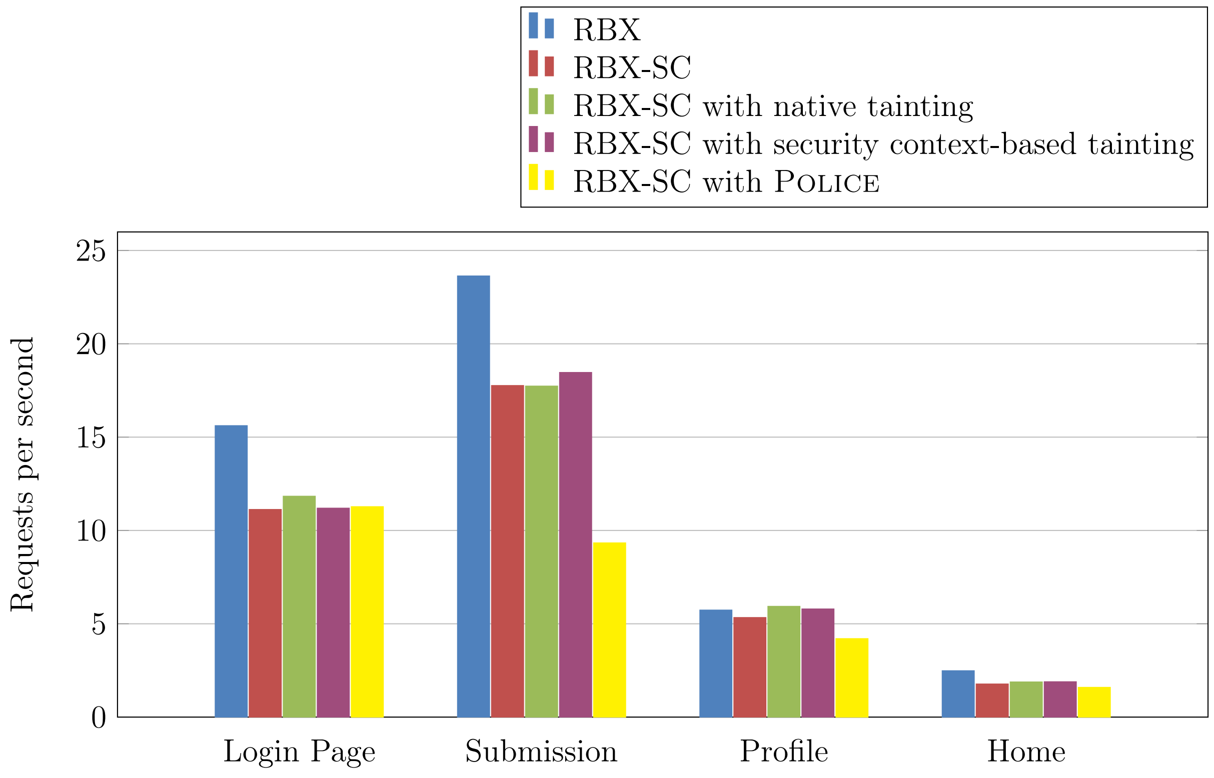

Figure 6-1: Visually comparing the throughput of requests from Table 6.1.

To establish a baseline, we first measured unmodified Rubinius v2.5.0 and RBX-SC v2.5.0, our modified version of Rubinius 2.5.0 with security contexts, across these tests without any tainting middleware or POLICE policies. From Table 6.1, we can see that RBX-SC has a performance drop between 28.7% on the login page load to 6.9% on the profile update. The primary slowdown is due to RBX-SC's Ruby-level hook into string interpolation, rather than using the Rubinius VM's optimized string builder methods. Without this hook, the overhead of RBX-SC on code without security contexts is a single if-statement check per method call.

Using RBX-SC, we then measured the performance cost of tainting using Rubinius's built-in tainting, versus tainting via security contexts. These numbers are quite close, and bode well for security context-based flow control schemes. In fact, the test for accessing a homework file was faster on the security context tainting than native, which hints that the noise of parsing a request and querying the database might overshadow any overhead in using security contexts for tainting. This is further evident when looking at the results of plain RBX-SC, which has results all within the same tight range and is actually a bit slower on average. Tainting natively or with security contexts seems to have near negligible impact on Rails app performance.

However, with POLICE we can see that the numbers do dip on all tests aside from the login page, which had no policies attached to its data. The home page throughput declines 10% compared to plain RBX-SC, as some of the homepage's content has read restrictions that must be checked. Updating profile information decreases in throughput by 27.5%, since each of the POST request parameters must be labeled, and then policy checked before committing to the database. Perhaps most surprising is the decrease of 47.5% throughput for accessing a read protected homework file. This request involves no template rendering, which has a high CPU burden, which might have masked the overhead of label propagation in other requests. It is worth noting that POLICE'S performance on the login page suggests negligible slowdown for pages that do not handle policy-protected data.

These numbers are all within the realm of reason for a security-conscious application. They furthermore add negligible memory usage: we used security contexts as singleton objects, so their memory overhead was a single Ruby object. It is also worth noting that RBX-SC has not been optimized, and there is significant low-hanging fruit to further improve its performance. In particular, when the VM executes the *pre-hooks* and *post-hooks*, it first checks if the hook methods exist, and then executes the method call, which in itself requires another identical lookup. This makes five calls in total: each hook, lookups for each hook, and the original method call. Caching the lookups is an easy win for performance. Another area for improvement is the Ruby-level string interpolation method, which has not been fine-tuned for performance. String interpolation is a ubiquitous operation whose performance strongly affects the performance of the application as a whole, as seen in the difference in baselines of RBX and RBX-SC.

6.2 Policy Protection

To measure the efficacy of POLICE'S policies to enforce read and write restrictions, we removed the explicit access control checks throughout *Seven* and instead attached policies to the models. We then attempted to exploit the absence of access control checks to manipulate data owned by other users from a user account that was not privileged to do so. These included attempts to:

- Update another user's profile information
- Read a protected user's profile information
- Directly download the generated PDF homework file of another user
- Load the homework grade report of another user
- Intentionally mix in read-protected information into the user's newsfeed

All of these attempts were prevented by POLICE, which was able to track the provenance of sensitive data into the database write or page output and cause the application to abort the request when a policy would have been violated. POLICE's success across these requests was warming to see, though not altogether a surprise–the design is modeled after RESIN, which has shown that such a system is conceivable and powerful. POLICE's novelty lies in its implementation, built on the security context, that allowed such the entirety of the scheme to come to life as a Ruby script, rather than fragmenting the work across virtual machine modifications and web application changes.

6.3 Limitations

One shortcoming of information flow control in Ruby is that `Fixnum` (`Integer`) and `Boolean` values are internally represented with immutable singletons. They therefore do not support tainting, the thinking being that they cannot be created by an external source, so if those objects exist within the runtime then they are implicitly sanitized.

This design prevents us from attaching a security context to `Fixnum`s and `Boolean`s, so we are unable to track their data flow.

This does pose some limitations on the potential policies that Police can define. For instance, a `User`'s grades for an assignment are stored as `Integer` values. Ideally, these grades should be `:read` policy protected from being accidentally or maliciously revealed to another non-staff user, but because of Ruby's limitations, the values cannot be labeled with `ReadRestrictions` so this policy cannot be attached to the grade numbers themselves.

If required, this can be worked around by storing all sensitive values at `String` types, but that introduces complexity and can require restructuring how data is stored, which hampers the plug-and-play goal of Police.

6.4 Programmer Effort

Building security context-based taint checking took under 200 lines of code, and passes all relevant RubySpec tests.

Police is under 600 lines of code for both the label propagation logic as well as hooks into Ruby on Rails. The majority of the security context propagation code is largely borrowed from the taint tracking reimplementation. Police is bundled into a Ruby Gem, and was added into the Rails application with two lines of code. A typical policy is written in a familiar DSL, and was often only 1-3 lines of code. The full source code for both applications is listed in Appendix A.

While identifying all of the proper hooks to fully support propagation through data flow is not necessarily trivial, for most schemes that follow the native tainting propagation rules, it only must be done once. Police was able to use all of the same propagation rules as taint tracking, but with the addition of moving labels between objects, which is a simple change.

Furthermore, it cannot be overstated how powerful it is to have an entire scheme documented within a single class. Part of the complexity of identifying taint tracking's propagation rules lay in its lack of centralized definition–it had hitherto existed as a scattering of statements all throughout the virtual machine and standard library. Consolidating the logic into a single place in the codebase dramatically increases the ease of comprehending and developing the scheme, lowering the overall programming effort required.

Chapter 7

Future Work and Conclusions

We have demonstrated security contexts to be a viable structure for building flow control schemes. We have specifically focused on security-related applications, but this is not all security contexts can do! IFC is a powerful technique that can benefit a wide variety of applications, many of which can be built with security contexts.

7.1 Further Applications

Security contexts open up a large number of powerful flow control applications. Some possible avenues to explore:

- Entropy tracking of probabilistic values–randomly generated values can be attached with a security context. The context can define how the entropy of the value changes based on the operations it undergoes throughout the application.
- Digital-rights management–protected content can be attached with a security context that manages fine-grain policies, e.g. marking a song with a policy that asserts it cannot be played more than five times. Trishul [21], a modified JVM with label propagation explored this as one of its core applications.
- Classification / Declassification policies–a traditional use of flow control, to mark data with different tiers of classification and prevent the leakage of sensitive data to unprivileged users, or similar lattice-based policy schemes [10].

- Dynamically building models of objects and methods–track the types and ranges of values into and out of methods in production. This could be extended to automatically generate tests for methods once enough data is gathered.

- Data-oriented profiling–understanding the specific flow of a piece of data throughout a large application and its effect on other values. This could perhaps be used to track observability.

- Character-level tainting–this is a fine-grain approach to have separate labels on individual characters within a `String`, and not just on the `String` in entirety. Security contexts could demarcate the index boundaries of certain labels for characters in a `String`, and then hook into `String` methods to create the merge rules.

The nature of the hook methods also offer a number of smaller, perhaps less exciting opportunities:

- Automatic argument fixing–*pre-hooks* can be used to automatically correct or modify arguments going in to a method.

- Profiling–understanding how often methods are called during execution. Rubinius inserts profiling methods directly into its virtual machine, but security contexts could avoid this requirement.

- Invariants/Type Checking–*pre-hooks* and *post-hooks* can be used to assert invariant properties of the arguments and return values. For the truly devious, this could be expanded to a more rigorous type checking system.

7.2 Future Work

The security context offers all of the features needed to implement IFC, but its API may not satisfy all developers. Exploring further interfaces, using the security context as a base, could be fruitful. Police used security contexts behind the scenes so that a developer only ever had to use Police's DSL to specify policies, and never had to touch the security contexts directly. This is a good first step, and the code of the security context is the next step for simplification and abstraction.

Of course, one of the biggest barriers to adoption of information flow control is that most languages cannot support them. Security contexts were designed to demonstrate that a simple structure is all that is needed, but unfortunately, the security context is not yet a part of any mainstream language. Adoption of the security context or comparable design into a widely used language would go a long way towards bringing flow control to every day applications. The ECMAScript 6 Proxy, whose design inspired the security context, will ideally bring such functionality to JavaScript. We have not spent adequate time with the Proxy to fully understand its capabilities and limitations, but it could potentially bring flow control to JavaScript-based applications, which would be a potent tool for the web.

7.3 Conclusion

Researchers interested in information flow control have to keep reinventing the wheel when it comes to implementing their ideas. Most languages lack the features necessary to natively build runtime flow control schemes, which stunts the development of new and exciting ideas. But building flow control systems should not be so challenging– with the right language features, it can and should be simple and accessible.

The security context was designed to fill in this gap. It provides all of the functions needed to develop information flow control ideas. Its *pre-hooks* and *post-hooks* give it runtime access to an object's data flow, and it is exists as user-level code, not as complex logic scattered throughout the language virtual machine. This greatly lowers the barrier to the flow control design process, and expedites the development and testing cycle.

We have shown the security context to be effective in building two schemes: taint tracking and data flow assertions for Rails applications. Ruby's taint tracking was fully emulated as a single security context class, and the data flow assertions library is similarly straightforward. The consolidation of flow control logic into single classes is an enormous improvement over the existing systems that scatter logic throughout the language, sometimes in over 100 locations. The performance of these schemes running on a production application was reasonable, particularly for applications handling sensitive data.

Appendix A

Listings

```
0000: push_literal      "world!"
0002: string_dup
0003: send_stack        :taint, 0
0006: set_local         0    # w
0008: pop
0009: push_literal      "Hello "
0011: push_local        0    # w
0013: allow_private
0014: meta_to_s         :to_s
0016: string_build      2
0018: pop
0019: push_true
0020: ret
```

Listing A.1: Bytecode generated by stock Rubinius compiler for 3.1. The `string_build` function is defined in the virtual machine and cannot be hooked from Ruby.

```
0000: push_literal    "world!"
0002: string_dup
0003: send_stack      :taint, 0
0006: set_local       0    # w
0008: pop
0009: push_const_fast :String
0011: push_literal    "Hello "
0013: push_local      0    # w
0015: allow_private
0016: meta_to_s       :to_s
0018: send_stack      :interpolate_join, 2
0021: pop
0022: push_true
0023: ret
```

Listing A.2: Bytecode generated by the patched Rubinius compiler for 3.1. `interpolate_join` is a Ruby method, which can be hooked.

```ruby
class TaintContext
  attr_accessor :tainted

  @@simple_methods =
  [# Rails, SafeBuffer
   "concat",
   "safe_concat",
   "initialize_copy",

   # Object
   "clone",
   "dup",
   "to_f",
   "to_a",
   "to_s",
   "to_str",

   # String
   "b", "byteslice", "capitalize", "center",
   "chomp", "chop", "crypt", "delete",
   "downcase", "dump", "element_set", "encode",
   "gsub", "insert", "ljust", "lstrip",
   "modulo", "multiply", "plus", "prepend",
   "reverse", "rjust", "rstrip", "squeeze",
   "strip", "sub", "succ", "next",
   "swapcase", "tr", "tr_s", "transform",
   "upcase",

   "find_character",

   # Regexp
```

```
 "match",
 "match_start",
 "search_from",
 "last_match"]

@@multiparam_methods = ["split"]

@@operator_methods = ["multiply", # *
                      "divide",  # /
                      "plus",    # +
                      "minus",   # -
                      "modulo",  # %
                      "not",     # !
                      "gt",      # >
                      "lt",      # <
                      "gte",     # >=
                      "lte",     # <=
                      "backtick", # `
                      "invert",  # ~
                      "not_equals", # !=
                      "similar", # ===
                      "match",   # =~
                      "comparison", # <=>
                      "lshift",  # <<
                      "rshift",  # >>
                      "index",   # []
                      "element_assignment", # []=
                      "bitwise_and", # &
                      "bitwise_or", # |
                      "bitwise_xor", # ^
                      "exponent",   # **
                      "uplus",      # +@
```

```
                         "uminus"]     # -@

  def initialize(tainted)
   puts "Initializing the Tainting Security Context"
    @tainted = tainted

    if @tainted
      define_singleton_method("after_slice") do |obj, arg, method_args|
        case method_args[0]
        when String
          if method_args[0].tainted? and not arg.nil?
            arg.taint
          end
        else
          arg.taint
        end

        return arg
      end

      @@simple_methods.each do |meth|
        define_singleton_method("after_#{meth}") do |obj, arg, method_args|
          if obj.is_a? Array
            if obj.empty?
              return arg
            end
          end

          # Range should not pass on taint to its to_s, unless the
          # begin or ending strings of it are tainted. The Range object
          # itself shouldn't pass it on.
          if meth == "to_s"
            if obj.is_a? Range and not arg.tainted?
              return arg
            end

            if obj.is_a? Hash and obj.empty?
              return arg
            end
          end

          arg.taint
        end
      end

      @@multiparam_methods.each do |meth|
        define_singleton_method("after_#{meth}") do |obj, args, method_args|
          unless obj.is_a? Enumerable
            if args.is_a? Enumerable
              args.each do |arg|
                arg.taint
              end
            else
              args.taint
            end
          end

          return args
        end
      end

      @@operator_methods.each do |meth|
        define_singleton_method("after_op__#{meth}") do |obj, args,
            method_args|
          if not obj.is_a? Enumerable
            if args.is_a? Enumerable
              args.each do |arg|
                arg.taint
              end
            else
              args.taint
            end
          end

          return args
        end
      end
    end
  end

  def infect(host, source)
    host.taint
  end
  end

  module Kernel
    def taint
      if is_a? TrueClass or is_a? FalseClass or is_a? NilClass
      return self
      end

      if tainted? or frozen? or nil?
      return self
      end

      self.secure_context = SecurityManager::TaintedContext
    self
  end

  def tainted?
    if not secure_context?
      return false
    end

    self.secure_context == SecurityManager::TaintedContext
  end

  def untaint
    if is_a? TrueClass or is_a? FalseClass or is_a? NilClass
      return self
    end

    if frozen?
      return self
    end

    self.secure_context = nil
    self
  end

  module SecurityManager
    TaintedContext = TaintContext.new(tainted=true)
  end
end

class String
  alias_method :old_modulo, :%

```

```ruby
  def %(*args)
    ret = old_modulo *args

    unless %w(%e %E %f %g %G).include? self
      if self.eql? '%p'
        args.each do |arg|
          Rubinius::Type.infect ret, arg.inspect
        end
      else
        args.each do |arg|
          Rubinius::Type.infect ret, arg
        end
      end
    end

    ret
  end
end

class Array
  alias_method :old_pack, :pack

  def pack(directives)
    ret = old_pack directives

    self.each do |a|
      Rubinius::Type.infect ret, a
    end

    Rubinius::Type.infect ret, directives

    ret
  end
end

module Rubinius
  module Type
    class << self
      alias_method :old_infect, :infect

      def infect(host, source)
        if source and source.respond_to? :secure_context? and
            source.secure_context?
          source.secure_context.infect host, source
        end

        old_infect(host, source)
      end
    end
  end
end
```

Listing A.3: Taint checking fully re-written using security contexts

```ruby

module Police
  class Middleware
    def call(env)
      user_supplied_label = Police::DataFlow::UserSupplied.new
      req = Rack::Request.new(env)

     # Label all of the parameters from the user's request
      req.params.each do |k, v|
        if v.is_a? Hash
          v = label_hash(v, user_supplied_label)
          req.update_param k, v
        else
          req.update_param k, v.label_with(user_supplied_label)
        end
      end

      ['QUERY_STRING', 'REQUEST_URI', 'ORIGINAL_FULLPATH',
      'rack.request.query_string', 'rack.request.form_vars'].each do
          |user_supplied_data|
        env[user_supplied_data].label_with user_supplied_label
      end

      env['police.from_user_label'] = user_supplied_label

      env['police.set_user'] = lambda do |user|
        user_supplied_label.payload = user
      end

      status, headers, response = @app.call(env)

    if response.is_a? Rack::BodyProxy
      response.each do |v|
        v.labels.each do |label|
          if label.is_a? Police::DataFlow::ReadRestriction
            origin = label.payload
            origin.enforce_read_restrictions
                env['police.from_user_label'].payload
          end
        end
      end
    end
    return status, headers, response
  end
 end
end
```

Listing A.4: POLICE's middleware that labels incoming values with `UserSupplied` labels, and checks response output for `ReadRestriction` labels

```ruby
module Police
  module Controller
    extend ActiveSupport::Concern

    included do
      before_action :set_user
    end

    def set_user
      request.env['police.set_user'].call current_user
    end
  end
end
```

Listing A.5: POLICE extension to `ApplicationController` to set the `User` of the current session on all `UserSupplied` labels

```ruby
module Police
  class PoliceError < StandardError
  end

  module Model
    module DSL
      extend ActiveSupport::Concern

      included do
        extend ActiveModel::Naming
        extend ActiveModel::Callbacks
        extend ActiveModel::Translation

        after_initialize :start_dataflow, if: Proc.new {
            self.class.police_policies }
        before_save :check_dataflow_save, if: Proc.new {
            self.class.police_policies }
        before_update :check_dataflow_update, if: Proc.new {
            self.class.police_policies }
      end

      module ClassMethods
        attr_accessor :police_policies

        def police(*protected_fields, action_hash)
          if action_hash.keys.any? { |action| not [:read, :write].include?
              action }
            raise PoliceError, "cannot create Police policy for action
                #{action}"
          end

          policy = Policy.new protected_fields, action_hash

          @police_policies ||= []
          @police_policies << policy
        end
      end

      def enforce_read_restrictions(user)
        check_dataflow(:read, user)
      end

      def label_for_action(action)
        case action
        when :read
          Police::DataFlow::ReadRestriction.new self
        else
          raise PoliceError, "no known label for action #{action}"
        end
      end

      def attach_policy(policy)
        policy.protected_fields.each do |field|
          if policy.protected_actions.include? :read
            attach_label field, label_for_action(:read)
          end
        end
      end

      def enforce_policy(policy, action, user=nil)
        return true if not policy.protects_action? action
        results = []

```

```ruby
      if user.nil?
        policy.protected_fields.each do |field|
          field_with_labels = self

          if field.kind_of? Enumerable
            field.each do |subfield|
              field_with_labels = field_with_labels.send(subfield)
            end
          else
            field_with_labels = send(field)
          end

          field_with_labels.labels.each do |label|
            user = label.payload if label.is_a?
                Police::DataFlow::UserSupplied
            break
          end

          break if user
        end
      end

      results << policy.action_hash[action].call(self, user)
      results.all? { |r| r == true }
    end

    def start_dataflow
      if self.class.police_policies
        self.class.police_policies.each do |policy|
          attach_policy policy
        end
      end
    end

    def check_dataflow(action, user=nil)
      self.class.police_policies.each do |policy|
        if not enforce_policy policy, action, user
          raise PoliceError, "object fails policy #{policy} for action
              #{action}, user #{user}"
        end
      end
    end

    def check_dataflow_save
      check_dataflow(:write)
    end

    def check_dataflow_update
      check_dataflow(:write)
    end

    # Attaches a label to a field that will propagate if needed
    # Attaching a label also attaches a security context that will
    # provide the necessary data flow
    def attach_label(field, label)
      if field.kind_of? Enumerable
        to_label_object = self

        field.each do |subfield|
          if to_label_object
            to_label_object = to_label_object.send(subfield)
          else
            return
          end
          end

          to_label_object.label_with label
        else
          send(field).label_with label
        end
      end
    end
  end
end

ActiveRecord::Base.send :include, Police::Model::DSL
puts "Included Police into ActiveRecord"
```

Listing A.6: POLICE extension to *ActiveRecord* to check `:write` policies before a write, and to attach `ReadRestriction` labels to retrieved data

```ruby
class Object
  @labels = Set.new

  def labels
    @labels ||= Set.new
    @labels
  end

  def label_with(label)
    return self if frozen? or nil?

    if is_a? TrueClass or is_a? FalseClass or is_a? NilClass
      return self
    end

    return self if has_label? label

    if not secure_context?
      self.secure_context = Police::DataFlow::SecureContextSingleton
    end

    @labels ||= Set.new
    @labels.add label

    self
  end

  def propagate_labels(other)
    @labels.each { |label|
      label.propagate other } if labeled?
  end
```

```ruby

def has_label?(label)
  @labels ||= Set.new if not @labels
  @labels.include? label
end

def has_labels?(*labels_list)
  labels_list.all? { |l| has_label? l }
end

def labeled?
  return false if not @labels

  not @labels.empty?
end

# Can pass in :all to clear all labels
def remove_label(label)
  if is_a? TrueClass or is_a? FalseClass or is_a? NilClass
    return self
  end

  return self if frozen?

  if label == :all
    @labels = Set.new
  else
    @labels.delete? label
  end

  self.secure_context = nil if not labeled?

```

```
    self
  end

  def no_label_to_s
    nolabel = dup
    nolabel.remove_label :all

    nolabel
  end
end
```

Listing A.7: POLICE's additions to `Object` to add in labels as a common property of Ruby objects

Bibliography

[1] Austin Seipp Adam Bergmark, Ricky Elrod. Haskell language. `https://www.haskell.org/`. Accessed: 2015-04-23.

[2] Ryan B. Cancan: Authorization gem for ruby on rails. `https://github.com/ryanb/cancan`. Accessed: 2015-04-08.

[3] Jean Bacon, David Eyers, TFJ-M Pasquier, Jatinder Singh, Ioannis Papagiannis, and Peter Pietzuch. Information flow control for secure cloud computing. *Network and Service Management, IEEE Transactions on*, 11(1):76–89, 2014.

[4] GitHub Blog. Public key security vulnerability and mitigation. `https://github.com/blog/1068-public-key-security-vulnerability-and-mitigation`. Accessed: 2015-04-08.

[5] Jonathan Burket, Patrick Mutchler, Michael Weaver, Muzzammil Zaveri, and David Evans. Guardrails: a data-centric web application security framework. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 1–1. USENIX Association, 2011.

[6] Avik Chaudhuri and Jeffrey S Foster. Symbolic security analysis of ruby-on-rails web applications. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 585–594. ACM, 2010.

[7] Winnie Cheng, Dan RK Ports, David A Schultz, Victoria Popic, Aaron Blankstein, James A Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *USENIX Annual Technical Conference*, pages 139–151, 2012.

[8] Erika Chin and David Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM workshop on Secure web services*, pages 3–12. ACM, 2009.

[9] Benjamin Davis and Hao Chen. Dbtaint: cross-application information flow tracking via databases. *Proc. of WebApps*, 10, 2010.

[10] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[11] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[12] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. *ACM SIGOPS Operating Systems Review*, 42(4):301–313, 2008.

[13] Andrey Ermolinskiy, Sachin Katti, Scott Shenker, L Fowler, and Murphy McCauley. Towards practical taint tracking. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-92*, 2010.

[14] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *OSDI*, pages 47–60, 2012.

[15] William GJ Halfond, Alessandro Orso, and Pete Manolios. Wasp: Protecting web applications using positive tainting and syntax-aware evaluation. *Software Engineering, IEEE Transactions on*, 34(1):65–81, 2008.

[16] Petr Hosek, Matteo Migliavacca, Ioannis Papagiannis, David M Eyers, David Evans, Brian Shand, Jean Bacon, and Peter Pietzuch. Safeweb: A middleware for securing ruby-based web applications. In *Proceedings of the 12th International Middleware Conference*, pages 480–499. International Federation for Information Processing, 2011.

[17] Butler W Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[18] Benjamin Livshits and Stephen Chong. Towards fully automatic placement of security sanitizers and declassifiers. *ACM SIGPLAN Notices*, 48(1):385–398, 2013.

[19] Patrick McKenzie. Weapons of mass assignment. *Commun. ACM*, 54(5):54–59, 2011.

[20] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[21] Srijith Nair. Remote policy enforcement using java virtual machine. 2010.

[22] Srijith K Nair, Patrick ND Simpson, Bruno Crispo, and Andrew S Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

[23] Mozilla Developer Network. Proxy. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy`. Accessed: 2015-04-07.

[24] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically hardening web applications using precise tainting. *Security and Privacy in the Age of Ubiquitous Computing*, pages 295–307, 2005.

[25] Ruby on Rails. Active record validations. `http://guides.rubyonrails.org/active_record_validations.html`. Accessed: 2015-04-14.

[26] Ioannis D Papagiannis. Practical and efficient runtime taint tracking. 2013.

[27] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[28] Brian Shirai. Rubyspec: The standard you trust. `http://rubyspec.org/`. Accessed: 2015-04-13.

[29] Deian Stefan, Edward Z Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazieres. Protecting users by confining javascript with cowl. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[30] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. In *ACM SIGPLAN Notices*, volume 47, pages 137–148. ACM, 2012.

[31] FJ-M Pasquier Thomas, Jean Bacon, and Brian Shand. Flowr: Aspect oriented programming for information flow control in ruby.

[32] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

[33] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *NDSS*, 2007.

[34] Jonathan Burket Patrick Mutchler Michael Weaver and Muzzammil Zaveri David Evans. Guardrails: A data-centric web application security framework. In *2nd USENIX Conference on Web Application Development*, page 1, 2011.

[35] D Wichers. Owasp top 10. *Online at https://www. owasp. org/index. php/Category: OWASP_ Top_ Ten_ Project*, 2013.

[36] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 291–304. ACM, 2009.

[37] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278. USENIX Association, 2006.