2013 IEEE Symposium on Security and Privacy

All Your IFCException Are Belong To Us

Cătălin Hriţcu[1], Michael Greenberg[1], Ben Karel[1], Benjamin C. Pierce[1], Greg Morrisett[2]
[1]University of Pennsylvania [2]Harvard University

***Abstract*—Existing designs for fine-grained, dynamic information-flow control assume that it is acceptable to terminate the entire system when an incorrect flow is detected—i.e, they give up availability for the sake of confidentiality and integrity. This is an unrealistic limitation for systems such as long-running servers.**

We identify *public labels* and *delayed exceptions* as crucial ingredients for making information-flow errors recoverable in a sound and usable language, and we propose two new error-handling mechanisms that make all errors recoverable. The first mechanism builds directly on these basic ingredients, using *not-a-values (NaVs)* and data flow to propagate errors. The second mechanism adapts the standard exception model to satisfy the extra constraints arising from information flow control, converting thrown exceptions to delayed ones at certain points. We prove that both mechanisms enjoy the fundamental soundness property of non-interference. Finally, we describe a prototype implementation of a full-scale language with NaVs and report on our experience building robust software components in this setting.

***Keywords*-dynamic information flow control, fine-grained labeling, availability, reliability, error recovery, exception handling, programming-language design, public labels, delayed exceptions, not-a-values, NaVs**

1 Introduction

Information flow control (IFC) [24] is an approach to security that controls how information propagates between the various components of a system, and between the system and the outside world. This is achieved by associating security levels (called *labels*) to entities such as processes, communication channels, files, and data structures, and enforcing the constraint that information derived from secret data does not leak to untrusted processes or to the public network. Conversely, IFC can enforce that untrusted processes or tainted inputs from the network have only carefully mediated influence on high integrity entities such as a database. These guarantees help reduce the trusted computing base, preventing bugs in untrusted code from breaking the confidentiality or integrity properties of the whole system.

Approaches to IFC fall roughly into two groups: *static*, where labels and information-flow checks are built into a type system or other static analysis tool [24, etc.] and *dynamic*, where labels are attached to run-time entities and propagated during execution. Static approaches have the usual advantages of early error detection and low run-time overhead. On the other hand, dynamic techniques are applicable in settings such as scripting languages [6], [8], [15], operating systems [13], [19], [28], and hardware implementations [10], [12] where static checking is problematic. Moreover, while early implementations of dynamic IFC focused on simple forms of taint tracking that did not detect implicit flows (secrets transmitted through the program's control flow), it has recently been shown [4], [25] that more sophisticated dynamic checks can soundly enforce a well-defined, formal policy—termination-insensitive non-interference, our criterion for *sound* IFC. Furthermore, dynamic IFC can be used together with discretionary access control (e.g., clearance [26]) to break up large systems into mutually distrustful components that run with least privilege [10], [13], [19], [26], [28].

Dynamic IFC can work at different levels of granularity. In *fine-grained* dynamic IFC (*FIFC*, for short) [4]–[6], [14], [15], [22], [23], [25], [26], each value—including, in general, the constituent parts of compound values—is protected by its own label, and the result of each computation step is given a label based on the labels of all the data involved. The main advantage of such fine-grained labeling is that it allows individual values to be *de*classified when necessary; this makes it easier to understand what gets declassified and simplifies the code audit process, compared with coarse-grained techniques [13], [19], [21], [28, etc.] where all the data owned by a process has a single label and thus gets classified and declassified together. Our focus in this paper is on (sound) FIFC.

However, current formulations of FIFC[1] suffer from a critical weakness: IFC violations are not recoverable. Instead, they lead to fatal "stop the world" errors in which the entire system is immediately terminated. This makes them unsuitable for some real-world settings—ones where not only confidentiality and integrity but also high *availability* are crucial concerns. To remedy this shortcoming, we need to enrich FIFC with an error-handling mechanism that allows *all* errors (IFC violations and others) to be recoverable, but that does not violate the soundness of information-flow tracking. Showing how this can be done is the main contribution of this paper.

Poison-pill Attacks To illustrate the problems and introduce the main ideas of our solution (§2 gives more details), we start by explaining a new class of availability attacks that are specific to FIFC, which we call *poison-pill attacks*. For this we use a simple idealized example—a server that

[1]One partial exception [27] is discussed in §9.

© 2013, Catalin Hritcu. Under license to IEEE.
DOI 10.1109/SP.2013.10

IEEE computer society

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

receives a pair of numbers, sends the larger one back to the client, and then loops to service the next request:

```
fun process_max (x,y) = if x <= y then y else x

fun rec max_server_loop () =
  send out (process_max (recv in));
  max_server_loop ()
```

The request and the response happen over public inter-process communication channels `in` and `out` respectively, so the pair received by the server is guaranteed to be labeled public, and the server has to produce a public response. However, with fine-grained labeling, data structures can be heterogeneously labeled (i.e., even though a pair is labeled public, its components can still be classified) and channels only check the topmost label.

A malicious or confused client can mount an attack on the max server by sending it a *poison pill*—a pair labeled public containing numbers labeled secret. The server will compare these numbers and try to send the larger of the two back to the client. But since this number is labeled secret, the send performed by the server will fail with a fatal IFC violation.

We would like to protect the max server from such availability attacks. The standard idiom in programming languages is for all errors to lead to *catchable* exceptions; we can then wrap the body of the server in a `try/catch` expression and thereby ensure that it keeps running:

```
fun rec max_server_loop' () =
  try send out (process_max (recv in))
  catch x => log x;
  max_server_loop' ()
```

However, combining catchable exceptions with FIFC can easily lead to unsoundness, since exceptions can leak secrets via labels or via the control flow of the program. In the rest of this section we sketch each of these problems and describe our solutions at a high level, postponing details to §2.

Problem: IFC Exceptions Reveal Information About Labels It is well known in the IFC community [23], [26], [29, etc.] that dynamically varying labels are themselves information channels. For instance, the following simple example encodes the secret bit `h` by varying the label of the final result:

```
if h then ()@high else ()@top
```

In this and the following examples we use label `low` for public data, `high` for secret data, and `top` for top-secret data. We use the term `()@high` to classify unit to label `high`.

In a FIFC language there is usually more than one label channel—e.g., one for labels on values, illustrated above, and a different one for labels on references (used for controlling reads and writes). For each label channel, we can prevent leaking secrets in one of two ways: (1) either by preventing secret information from leaking *into* the label channel [4], [26] or (2) by preventing any information from leaking *out of* the label channel [4], [5], so that, even though there may be secrets in the label channel, there is no way to observe them. In the presence of catchable IFC exceptions, however, the second alternative is not satisfactory. Observing IFC exceptions inherently reveals information about labels, so if one wanted to prevent information from leaking out of the label channel, one would need to impose severe restrictions on the observability of exceptions.[2] The following example encodes the secret `h` using labels as above and then tries to leak it using catchable IFC exceptions:

```
try
  href := (if h then ()@high else ()@top);
  true
catch IFCException => false
```

Here, `href` is a reference cell holding `high` values. Writing to `href` succeeds when `h` is `true` (writing a `high` value to a `high` reference is OK) but raises an exception when `h` is `false` (writing a `top`-secret value to a `high` reference fails). Thus the success or failure of the assignment depends on the label of the value that gets written. Since the reference write is now basically a conditional branching construct, one could prevent the leak by recording that the control flow decision was potentially influenced by secrets encoded in the label of the value that gets written. This would, however, lead to a very restrictive language, in which the information whether an IFC exception has occurred or not is protected by an arbitrarily high label, and where programmers have little control over how data is labeled. For instance, in such a system the max server above would have no way to log that an IFC exception occurred, since this information would be labeled with a label chosen by the poison-pill attacker.

Solution: Sound Public Labels Instead of allowing secrets to flow into the label channel and then trying to hide labels (and thus IFC exceptions), we obtain soundness by making sure that the information in the labels is public in the first place. We can do this by separating the choice of label (which needs to be done in a low context) from the computation of the labeled data (which happens in a high context). To achieve this separation we put the code that branches on secrets inside *brackets* that explicitly specify the label of the result [26]. In the example above the conditional has to be bracketed with `top`, i.e., a label that is more secure than the label of the result of either branch:

```
top[if h then ()@high else ()@top]
```

Regardless of which branch is chosen, the example now evaluates to `()@top`, thus preventing `h` from being leaked to the label channel. Brackets close the label channel, which allows us to make labels and IFC errors publicly observable. Moreover, the soundness of the techniques we propose does not depend on the label annotations on brackets being

[2] This problem only gets worse when one also adds label-based discretionary access control to the language (as we do in Breeze), since then even adding two numbers can cause access control violations and thus reveal information about labels.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

correct. We defer the discussion about brackets and incorrect annotations to §2.2 and §2.3.

Problem: Exceptions Destroy Control Flow Merge Points The standard formulation of catchable exceptions can leak secret information via the control flow of the program. Propagating exceptions adds many new edges to the control flow graph and thus introduces additional exit edges out of basic blocks. Without exceptions there is a unique edge out of a conditional which merges the two branches. Such *control flow merge points* play a crucial role for IFC in general, because they mark the end of a high context (where some secrets have affected the control flow). For instance, brackets are only sound if ending brackets are control flow merge points, but standard exception propagation breaks this invariant. We explain this in more detail in §2.3, but intuitively the problem here is that an expression like

```
try
  ignore high[if h then throw Ex else ()];
  false
catch _ => true
```

throws an exception in a `high` context, but catches it in a `low` context, outside the brackets. The fact that the exception "jumps" out of the brackets allows the secret boolean `h` to be leaked as a `low` boolean.

Solution: Delayed Exceptions To fix this, we need to change the language so that exceptions do *not* jump out of the brackets—i.e., any exception that happens inside a bracket needs to be *delayed* and turned into a *result* of the bracket expression. While such delayed exceptions seem unavoidable given the constraints of our setting (see §2.3), we do have a choice about exactly how they propagate when they are used, for instance: (1) we can simply rethrow the exception (see §5.2 and LIO [27]), or (2) we can change the semantics so that the result of the operation is the delayed exception (see §2.4 and §4). While we investigate both these alternatives in the paper, we find the second one particularly interesting, because it allows us to devise an error handling mechanism based *solely* on delayed exceptions. In this new mechanism, exceptions are propagated only via the data flow of the program. Since this is to a certain extent a generalization of how not-a-numbers (NaNs) [16] propagate, we call such delayed exceptions *not-a-values (NaVs).*

Contributions Our primary contribution is the identification of public labels and delayed exceptions (§2) as the key ingredients for making all errors (IFC violations and all other exceptional conditions) recoverable in a sound and usable language. Additionally, we explore the space of possible designs based on these ingredients and focus on two propagation approaches for delayed exceptions: a simpler one using not-a-values (NaVs) and data flow to propagate exceptions (§2.4 and §4), and a more complex one using standard catchable exceptions that are delayed by ending brackets (§5). We identify the rules that ensure soundness in either case and we formally prove in Coq that both designs enforce error-sensitive, termination-insensitive non-interference [1]. Moreover, we devise translations that encode each error handling mechanism in terms of the other (§6). We also illustrate how each of the mechanisms can be used to protect the simple max server above against poison-pill attacks (§7). Finally, we have designed and implemented a language called Breeze that incorporates the simpler and more novel approach based on NaVs [1]. To gain experience with the design, we have constructed a large library and a number of small but illustrative applications. We report on our experience, identifying practical issues that arise with NaVs and idioms that can be used to work around potential shortcomings (§8).

We discuss related work in §9; we conclude and sketch future directions in §10.

2 Overview

To set the stage for the details of the calculi and their properties, this section gives a technical overview of the underlying ideas. §2.1 is a gentle introduction to the basic mechanisms of FIFC; §2.2 gives more details on public labels and brackets; §2.3 explains why delayed exceptions are unavoidable if we want all errors to be recoverable in a FIFC system; finally, §2.4 explains NaVs.

2.1 A Gentle Introduction to FIFC

In order to track information flow at a very fine level of granularity [4], [5], each value is protected by an individual IFC label representing a security level (e.g., `low`, `high`, or `top`). Security levels are partially ordered: `low` is below `high` (since it is always safe to protect public data as if it was a secret) and `high` is below `top`. The semantics of the language automatically propagates these labels as computation progresses. For instance, the expression `1@low + 2@high` evaluates to `3@high`, thus capturing the dependency of the result on the secret input `2`. Trying to write the secret result to a public reference (i.e., readable by the attacker) is an example of an *explicit flow*; it results in an IFC violation:

```
lref := 1@low + 2@high  // -> IFC violation
```

In most existing FIFC systems [4]–[6], [15], [22], [23], [25], such IFC violations are fatal errors, immediately stopping the execution of the program to prevent secret information from being leaked.

Preventing only explicit flows is not enough to obtain a sound IFC system, though, since the control flow of the program can also leak secret information:

```
lref := false; if h then lref := true
```

In this example, an *implicit flow*[3] is used to copy the secret bit `h` to the public reference `lref`. The standard way of stopping such leaks is a *security context* label, called the

[3] In this paper we use the term *implicit flow* to mean any information leak via the control flow of the program.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

pc label, that dynamically captures the security level of all the values that have influenced the control flow. In the example above, branching on h raises the pc to high, which prevents writing to low references such as lref. In all of this section's examples, the pc starts out as low.

Stopping low side-effects when the pc is high is necessary but not sufficient for stopping implicit flows. Implicit flows can equally well affect purely functional code:

```
if h then true else false
```

Assuming the constants true and false are public even when the pc is high, this code obtains a public copy of the secret h. The restriction on side-effects alone does not prevent this implicit flow, since in most existing FIFC systems [4]–[6], [14], [15], [22], [23], [25] the pc is automatically restored on control flow merge points. This means that, without additional restrictions, the following code would successfully exfiltrate h, since lref is only updated after the two branches of the conditional are merged.

```
lref := (if h then true else false)
```

One way to soundly restore the pc automatically is to let the pc "infect" the resulting value first [4], [5]. Then whatever the conditional returns is at least as secret as h—and therefore cannot be written to lref. However, as we will see in the next section, automatically restoring the pc is not sound when labels are publicly observable.

2.2 Public Labels and Brackets

Since FIFC enforces security dynamically, IFC labels have a run-time representation and are automatically propagated by the FIFC system. It is well known in the IFC community [23], [26], [29, etc.] that dynamically varying labels are themselves information channels. However, many of the existing FIFC systems [4]–[6], [15], [22] do not completely prevent leaking secrets into label channels. Instead, they preserve soundness by preventing (some of) the labels from being publicly observable. In a FIFC system, automatically restoring the pc on control flow merge points allows information to be leaked into the label channel formed by the labels on values. So in a FIFC system with automatic pc restoring allowing any way of publicly observing information about the labels on values would be unsound. For instance, adding a label inspection construct would be unsound: the following simple example would leak the secret bit h.

```
labelOf (if h then ()@high else ()@top) == high
```

This is similar to the purely functional implicit flow example above, but here we are varying the *label* of the result of a conditional based on the secret h—the result *value* is unit on both branches. The labels we use for signaling (high and top secret) are above or the same as the label of h (high), so "infecting" the result of the conditional with the high pc [4], [5], as discussed at the end of §2.1, does not have any effect. If the pc is automatically restored at the end of the conditional, the variation in the labels on values is made public by labelOf, revealing the secret h.

Label inspection is, however, only one way of making labels observable. Making IFC errors recoverable also reveals information about the labels, as we saw in §1. Since we think that trying to restrict information about IFC errors is counterproductive, we obtain soundness by preventing secrets from being leaked *into* the label channels. For the label channel formed by the labels on values, we do this by restoring the pc only manually, using brackets [26]. With brackets the pc is restored after a conditional *only if the label on the result has been chosen in advance, before looking at any secrets*. In a language with brackets the example above is safe, because the pc is not automatically restored at the end of the conditional, so it stays high and stops any low side-effects *even after* the control flow merge point. In this setting, each value is effectively protected both by its explicit label and by the current pc. To restore the pc we must wrap the conditional in a bracket, as in:

```
top[if h then ()@high else ()@top]
```

No matter which branch of the conditional is taken, the result is labeled top, and the pc is restored to the value it had before the bracket started. Brackets are always control flow merge points, so the pc can be safely restored. The label on the bracket is chosen outside the bracket, before it runs, so it cannot depend on any secrets inspected inside. However, the semantics of brackets also needs to ensure that the label on the bracket is high enough to effectively protect the result of the bracketed expression; i.e., brackets are not a declassification construct. For instance, if we were to put a lower label on the bracket, say high, in the example above, then executing the else branch would cause an IFC error, since high is not above top:

```
high[if h then ()@high else ()@top]
```

If availability were of no concern, one could make such failed brackets be fatal errors and obtain a language with public labels that has error-insensitive, termination-insensitive non-interference (indeed, we do as much in our $\lambda^{[\,]}$ calculus in §3). Error insensitivity means that this soundness result ignores computations where brackets are incorrectly labeled.

2.3 Delayed Exceptions

While we want IFC errors to be recoverable, failed brackets cannot throw catchable exceptions: we can only soundly restore the pc at control flow merge points, and throwing exceptions would destroy the merge point at the end of a bracket. The following example would exfiltrate h if failed brackets threw an exception but still restored the pc:

```
lref := false
try
  ignore high[if h then ()@high else ()@top];
  lref := true
catch _ => ()
```


Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

The bracket would succeed when `h` is `true`; the bracket would fail with a catchable exception when `h` is `false` causing `lref := true` to be skipped. At the end of the bracket, the pc returns to `low`, so when `h` is `true` the update to `lref` would be allowed to exfiltrate the secret. As illustrated in §1, a very similar problem occurs if exceptions in the body of the bracket freely jump out of the bracket.

To prevent such behavior, brackets must either produce a value or diverge. There needs to be exactly one control flow edge leaving the bracket; they cannot throw catchable exceptions. Moreover, since we do not want labels to be a possible source of leaks, whatever comes out of the bracket must be labeled with the bracket's label. Nevertheless, in order for the error handling mechanism to be useful in practice, the produced value has to be as informative as possible. In particular, this value should record if the bracket has failed or not. If the bracket has failed, the value should record the cause of the failure if possible. We believe that any workable solution to these design constraints will have to involve *delayed exceptions* in one form or another. Thus, when a bracket fails, the result should be a delayed exception protected with the label specified on the bracket.

2.4 Not-a-Values (NaVs)

One can design a language with both catchable and delayed exceptions (we do that in §5). However, a more radical solution is to get rid of catchable exceptions altogether and to design a new error handling mechanism based solely on delayed exceptions in the form of *not-a-values (NaVs)*. We outline the main ideas of this solution here and study the details in §4 and §8. NaVs are *first-class* replacements for values that are propagated solely via the data flow of the program. Like values, NaVs are labeled. More importantly, NaVs are *pervasive*: (a) all errors produce NaVs that remember the cause (e.g., dividing by zero will produce a different NaV than trying to add a boolean to an int), and (b) all non-parametric operations are NaV-strict (adding an int to a NaV will return the original NaV). However, for parametric operations, which do not inspect their arguments, there is a choice whether to be NaV-strict or to be NaV-lax. There are two questions one has to answer:

1) Should a function applied to a NaV argument fail and return the NaV (NaV-strict) or just bind that argument to the NaV and keep evaluating the function's body (NaV-lax)?
2) Should constructing a data value using NaV arguments produce a NaV (NaV-strict) or simply produce a data structure containing NaVs (NaV-lax)?

NaV-strictness has the advantage of short-cutting error propagation and revealing errors earlier, but it also has several big disadvantages.

1) NaV-strict function applications introduce a new control flow edge: when the argument is a NaV, they jump over the function body. In order to preserve soundness, the pc must be raised by the label of the argument on all NaV-strict function calls.
2) NaV-strict data constructors force the label on data structures to be a summary of everything inside. Every time we NaV-strictly cons onto a list, we must first check that the value we are consing on is not a NaV. The label on the list—and everything we get out of it—will be higher than every cons cell's label.

The $\lambda^{[\,]}_{NaV}$ calculus in §4 gives the answer "NaV-lax" to the two questions above. That is, we make all parametric operations NaV-lax, while allowing NaVs to be "forced" explicitly. In our prototype implementation (described in §8), we allow the programmer to choose the desired behavior explicitly, on a case-by-case basis. While in theory selective NaV-strictness is only a convenience, in practice convenience makes a big difference.

One might wonder what would happen if one were to make all constructs of the language NaV-strict. In such a language NaVs would propagate very similarly to catchable exceptions. However, brackets would be totally useless, since as soon as the bracket would restore the pc, the bracket's context would perform a NaV check on its result, raising the pc even higher than it was right after the bracket ended.

What's in a NaV? Other than the cause of the error (i.e., the error message), each NaV contains two additional values that facilitate debugging: a *stack trace* that pinpoints the origin of the error and a *propagation trace* that records the way the NaV meandered from the place where it was originally created to the place where it was eventually detected. We have proved that providing these debugging aids does not invalidate our soundness results (see §4.2).

3 $\lambda^{[\,]}$: A FIFC Calculus with Public Labels

We begin with the basis of our FIFC calculi, $\lambda^{[\,]}$ (pronounced "lambda bracket"), a simple calculus for fine-grained purely dynamic IFC. In $\lambda^{[\,]}$ all labels are public but errors are still fatal; in §4 and §5 we extend $\lambda^{[\,]}$ with two different error handling mechanisms which make all errors recoverable. For the sake of simplicity, we drop some of the language features that we used in earlier examples (in particular, channels and references) and work with pure core calculi throughout our formal development.

The syntax of $\lambda^{[\,]}$ is in Figure 1; much of it is standard. Information flow aside, $\lambda^{[\,]}$ is a dynamically typed lambda calculus with tagged variants (Inl, Inr, and match), equality on constants ($x == y$), and reflection on type tags (tagOf x). To simplify our evaluation relations, we present $\lambda^{[\,]}$ and its extensions in a syntactically restricted form reminiscent of A-normal form (ANF). In the examples we give throughout the paper we will, however, use standard syntactic sugar.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

Constants

$c \quad ::= \quad () \mid L \mid \mathsf{TFun} \mid \mathsf{TSum} \mid \mathsf{TUnit} \mid \mathsf{TLab} \mid \mathsf{TTag}$

Terms, constructors, and operations

$t \quad ::= \quad x \mid c \mid \mathsf{let}\, x = t \,\mathsf{in}\, t' \mid \lambda x.t \mid x\, y \mid C\, x \mid$
$\mathsf{tagOf}\, x \mid x \otimes y \mid \mathsf{getPc}\,() \mid \underline{x}[t] \mid \mathsf{labelOf}\, x \mid$
$(\mathsf{match}\, x \,\mathsf{with} \mid \mathsf{Inl}\, x_1 \Rightarrow t_1 \mid \mathsf{Inr}\, x_2 \Rightarrow t_2)$

$C \quad ::= \quad \mathsf{Inl} \mid \mathsf{Inr}$

$\otimes \quad ::= \quad == \mid \sqsubseteq \mid \vee$

Values, environments, and atoms

$v \quad ::= \quad c \mid C\, a \mid \langle \rho,\, \lambda x.\, t \rangle$

$\rho \quad ::= \quad x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$

$a \quad ::= \quad (\mathbf{V}\, v)@L$

Figure 1. Syntax of $\lambda^{[\,]}$

In order to track information flow at a very fine level of granularity, $\lambda^{[\,]}$ works with *atoms*: values labeled with a security level. The security levels (or "labels") are drawn from an arbitrary join-semilattice with a bottom element, which is denoted $\perp$ and used for labeling public data (in the examples from §1 and §2 we let $\perp$ = `low`). Unlike in most other FIFC systems [4]–[6], [14], [15], [22], [23], [25], $\lambda^{[\,]}$'s labels are *public* and *first-class*: labelOf performs label inspection, returning an atom's label—as an atom, itself labeled with $\perp$; the operator "$\vee$" computes the join of two labels; and the operator "$\sqsubseteq$" compares two labels according to the semi-lattice's partial order. Additionally, getPc () returns the current security context label, pc, which is the join of all labels of values that have affected control flow. The pc label is necessary for preventing *implicit flows*, which can affect even purely functional code (see §2.1). In $\lambda^{[\,]}$, every value is protected not only by its explicit label, but also by the current pc label. That is, values labeled public are still considered secret when the pc is secret.

A bracket $\underline{x}[t]$ serves two main purposes: it labels the result of evaluating t with the label x (classification); and, after evaluating t, it reverts the pc to its original level before the bracket. The latter is particularly important, since it is unsound for a language with public labels to automatically lower the pc at the end of conditionals or other control flow branches (see §2.2). The only way to restore the pc in $\lambda^{[\,]}$ is manually, using brackets. This is crucial for preventing leaks into "the label channel" (see §2.2), since the label on the final result is chosen in advance, before branching on secrets inside the bracket. Note that the label on the bracket need not be a constant—it can be computed at runtime, for instance using labelOf and joins.

The operational semantics of $\lambda^{[\,]}$ in Figure 2 adds FIFC to a completely standard environmental big-step semantics. We have three kind of values: constants, tagged variants, and closures. Values are heterogeneously labeled: $(\mathbf{V}\,(\mathsf{Inl}\,((\mathbf{V}\,())@\mathsf{high})))@\mathsf{low}$ is a high constant contained inside a public value. The evaluation relation uses an explicit environment ρ, mapping variables to atoms. The environment ρ implements lexical scoping, while the pc is threaded through like a piece of state (rule *BLet*). In $\lambda^{[\,]}$ there are only two kinds of errors: type errors and failed brackets. Neither can be handled—there simply won't be a derivation. An implementation would have to treat these errors as fatal and "stop the world".

Variables are just looked up in the environment (rule *BVar*). The standard introduction rules—*BConst*, *BSum*, and *BAbs*—follow a similar pattern: the introduced value is labeled with the public label, $\perp$. Since in $\lambda^{[\,]}$ labels are public, extracting the label from an atom (rule *BLabelOf*) or from the current pc (rule *BGetPc*) produces a first-class label value that is labeled $\perp$, too.

In rule *BApp* the body of the closure is evaluated under an extended environment and with a pc raised by the closure's label. We have to raise the pc because, due to first-class functions, *what* function we invoke is generally data-dependent. *BMatch*, the rule for pattern matching, also raises the pc: the scrutinee influences control flow. Type tags can also be used as an information channel, so when extracting the tag of an atom, we protect the result with the original atom's label (rule *BTagOf*). The rule for binary operations (*BBOp*) is standard: the condition on *tagsArgs* ensures that the operation is well typed, and the result of the operation is labeled with the join of the labels on the arguments.

Finally, *BBrk* specifies the semantics of brackets. The $L'' \vee pc' \sqsubseteq L \vee (pc \vee L')$ premise ensures that the value returned from the bracket is more protected with the bracket $(L \vee (pc \vee L'))$ than it would have been if we did not use a bracket $(L'' \vee pc')$; i.e., brackets are not a declassification construct. Keeping in mind that each value is protected by the join of its explicit label and the pc, we illustrate this condition below by means of examples.

In the simplest case, brackets merely classify data: the term $\underline{L}[x]$ classifies x to label L. Suppose that $\rho(x) = (\mathbf{V}\, v)@L''$ and that $L'' \sqsubseteq L \vee pc$; then we have $\rho \vdash \underline{L}[x], pc \Downarrow (\mathbf{V}\, v)@L, pc$. The label L is not itself secret, since *BConst* yields $\perp$-labeled atoms. *BVar* does not change the pc, so the pc stays the same throughout the bracket. The final condition on the bracket is $L'' \vee pc \sqsubseteq L \vee (pc \vee \perp)$, which holds because $pc \vee \perp = pc$, $\sqsubseteq$ is reflexive, and we have assumed that $L'' \sqsubseteq L \vee pc$. On the other hand, if $L'' \not\sqsubseteq L \vee pc$ then the condition at the end of the bracket does not hold (i.e., we are trying to declassify using a bracket), so there is no derivation. An implementation would have to cause a fatal error and "stop the world"—there is no safe way to continue running the program.

In addition to classifying data, brackets are the only way to lower the pc in $\lambda^{[\,]}$. For example, $t = \underline{\mathsf{high}}[\underline{\mathsf{high}}[\lambda x.x]\,(\lambda x.x)]$ starts a high bracket in which it classifies $\lambda x.x$ to high and then applies it to an unclassified $\lambda x.x$. We have the following derivation, starting and ending with a $\perp$ pc: $(\rho \vdash t, \perp \Downarrow (\mathbf{V}\,\langle\emptyset,\, \lambda x.\, x\rangle)@\mathsf{high}, \perp)$.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

$$\frac{\rho(x) = a}{\rho \vdash x, pc \Downarrow a, pc} \; (BVar)$$

$$\frac{\rho \vdash t, pc \Downarrow a, pc' \quad (\rho, x \mapsto a) \vdash t', pc' \Downarrow a', pc''}{\rho \vdash \mathsf{let}\, x = t \,\mathsf{in}\, t', pc \Downarrow a', pc''} \; (BLet)$$

$$\frac{}{\rho \vdash c, pc \Downarrow (\mathbf{V}\, c)@\bot, pc} \; (BConst)$$

$$\frac{\rho(x) = a}{\rho \vdash C\, x, pc \Downarrow (\mathbf{V}\,(C\, a))@\bot, pc} \; (BSum)$$

$$\frac{}{\rho \vdash (\lambda x.t), pc \Downarrow (\mathbf{V}\,\langle \rho,\, \lambda x.\, t\rangle)@\bot, pc} \; (BAbs)$$

$$\frac{\rho(x) = (\mathbf{V}\, v)@L}{\rho \vdash \mathsf{labelOf}\, x, pc \Downarrow (\mathbf{V}\, L)@\bot, pc} \; (BLabelOf)$$

$$\frac{}{\rho \vdash \mathsf{getPc}\,(), pc \Downarrow (\mathbf{V}\, pc)@\bot, pc} \; (BGetPc)$$

$$\frac{\begin{array}{l}\rho(x_1) = (\mathbf{V}\,\langle \rho',\, \lambda x.\, t\rangle)@L \quad \rho(x_2) = a \\ (\rho', x \mapsto a) \vdash t, (pc \vee L) \Downarrow a', pc'\end{array}}{\rho \vdash (x_1\, x_2), pc \Downarrow a', pc'} \; (BApp)$$

$$\frac{\begin{array}{l}\rho(x) = (\mathbf{V}\,(C\, a))@L \\ (\rho, y \mapsto a) \vdash t_C, pc \vee L \Downarrow a', pc'\end{array}}{\begin{array}{l}\rho \vdash \mathsf{match}\; x \;\mathsf{with} \quad\;\; , pc \Downarrow a', pc' \\ \quad\; \mid \mathsf{Inl}\; y \;\Rightarrow\; t_{\mathsf{Inl}} \\ \quad\; \mid \mathsf{Inr}\; y \;\Rightarrow\; t_{\mathsf{Inr}}\end{array}} \; (BMatch)$$

$$\frac{\rho(x) = (\mathbf{V}\, v)@L}{\rho \vdash (\mathsf{tagOf}\, x), pc \Downarrow (\mathbf{V}\,(\mathit{tagOf}\, v))@L, pc} \; (BTagOf)$$

$$\frac{\begin{array}{l}\rho(x_1) = (\mathbf{V}\, v_1)@L_1 \quad \rho(x_2) = (\mathbf{V}\, v_2)@L_2 \\ \{\mathit{tagOf}\, v_1, \mathit{tagOf}\, v_2\} \subseteq (\mathit{tagsArgs}\, \otimes) \quad v \triangleq v_1 \otimes v_2\end{array}}{\rho \vdash (x_1 \otimes x_2), pc \Downarrow (\mathbf{V}\, v)@(L_1 \vee L_2), pc} \; (BBOp)$$

$$\frac{\begin{array}{l}\rho(x) = (\mathbf{V}\, L)@L' \quad \rho \vdash t, (pc \vee L') \Downarrow (\mathbf{V}\, v)@L'', pc' \\ L'' \vee pc' \sqsubseteq L \vee (pc \vee L')\end{array}}{\rho \vdash \underline{x}[t], pc \Downarrow (\mathbf{V}\, v)@L, (pc \vee L')} \; (BBrk)$$

Where

$$\begin{array}{rcl} \mathit{tagsArgs}\ (\sqsubseteq) = \mathit{tagsArgs}\ (\vee) & = & \{\mathsf{TLab}\} \\ \mathit{tagsArgs}\ (==) & = & \{\mathsf{TUnit}, \mathsf{TLab}, \mathsf{TTag}\} \end{array}$$

Figure 2. Evaluation relation for $\lambda^{[\,]}$

The *BApp* sub-derivation for the bracket body finishes with $pc' = (\bot \vee \mathsf{high}) = \mathsf{high}$ and returns the atom $(\mathbf{V}\,\langle \rho,\, \lambda x.\, x\rangle)@\bot$. The condition at the end of the bracket is $\bot \vee \mathsf{high} \sqsubseteq \mathsf{high} \vee (\bot \vee \bot)$, which clearly holds. It is therefore sound to lower the pc to $\bot \vee \bot = \bot$ and relabel the closure as $(\mathbf{V}\,\langle \rho,\, \lambda x.\, x\rangle)@\mathsf{high}$. That is, the outer bracket has moved taint from the pc to the resulting value.

The two examples above illustrate the most common usage scenarios for brackets. There is another interesting use case: brackets can also be used for moving taint from values to the pc. For example, this usage of brackets can make a heterogeneously labeled data structure into one classified by a single outer label; pulling the inner label in $(\mathbf{V}\,(\mathsf{Inl}\,(\mathbf{V}\,())@\mathsf{high}))@\mathsf{high}$ out, yielding $(\mathbf{V}\,(\mathsf{Inl}\,(\mathbf{V}\,())@\mathsf{low}))@\mathsf{high}$. Or, suppose $\rho(x) = (\mathbf{V}\, v)@\mathsf{high}$ and the pc is high. We can use $\underline{\mathsf{low}}[x]$ to obtain $(\mathbf{V}\, v)@\mathsf{low}$.

Atom equivalence
$(\mathbf{V}\, v_1)@L' \equiv_L (\mathbf{V}\, v_2)@L' \iff L' \sqsubseteq L \implies v_1 \equiv_L v_2$

Value equivalence
$c \equiv_L c$
$C\, a_1 \equiv_L C\, a_2 \iff a_1 \equiv_L a_2$
$\langle \rho_1,\, \lambda x.\, t\rangle \equiv_L \langle \rho_2,\, \lambda x.\, t\rangle \iff \rho_1 \equiv_L \rho_2$

Environment equivalence
$\emptyset \equiv_L \emptyset$
$\rho_1, x \mapsto a_1 \equiv_L \rho_2, x \mapsto a_2 \iff \rho_1 \equiv_L \rho_2 \wedge a_1 \equiv_L a_2$

Figure 3. Equivalence below a given label L

This is not a declassification, since the high pc already protects v. In this example the condition at the end of the bracket is $\mathsf{high} \vee \mathsf{high} \sqsubseteq \mathsf{low} \vee (\mathsf{high} \vee \mathsf{low})$, which holds because (a) $\mathsf{low} \vee \mathsf{high} = \mathsf{high}$, (b) $\mathsf{high} \vee \mathsf{high} = \mathsf{high}$, and (c) $\sqsubseteq$ is reflexive. Here the label on the result of the bracket is *above* the label on the bracket. Joining the post-bracket pc on the right-hand-side of the condition in *BBrk* gives us the flexibility to permit this sound usage of brackets.

Non-interference $\lambda^{[\,]}$ enjoys *non-interference*: for every computation, the high parts of the input do not affect the low parts of the output. In $\lambda^{[\,]}$ the environment and the initial pc constitute the input, while the resulting atom and final pc constitute the output. The non-interference proof is fairly standard [4]. First, in Figure 3 we define a family of label-indexed equivalences $\equiv_L$ on atoms, values, and environments. Each equivalence distinguishes two classes: low things are labeled below L, and high things are not labeled below L. In each equivalence, low things must correspond closely, while high things need not. Atom equivalence is the crux of the $\equiv_L$ equivalence; the equivalences on values and environments are structural. Labels are public, so the labels on atoms are treated as low data: equivalent atoms have the same label. Low atoms labeled below L must have equivalent values, while high atoms need not.

Theorem 1 (Non-interference for $\lambda^{[\,]}$)**.** *Given a label L, a term t, environments ρ_1 and ρ_2, and a starting* pc *label pc, if: (1) $\rho_1 \equiv_L \rho_2$, (2) $\rho_1 \vdash t, pc \Downarrow a_1, pc'_1$, (3) $\rho_2 \vdash t, pc \Downarrow a_2, pc'_2$, and (4) $pc'_1 \sqsubseteq L$ or $pc'_2 \sqsubseteq L$ then $pc'_1 = pc'_2$ and $a_1 \equiv_L a_2$.*

Proof. By induction on (2), using the fact that the pc increases monotonically. We have proved this in Coq. ☐

Premise (4) of Theorem 1 is necessary because atoms are protected by both their labels and the pc label—if the computation finishes with a pc that is not below L, then there are no low parts of the output, and non-interference is immediately satisfied.

Non-interference in $\lambda^{[\,]}$ is *error insensitive* and *termination insensitive*. That is, since errors and divergence are represented by absence of a derivation, Theorem 1 says nothing in case of errors or divergence. Finally, we do not

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

Exceptions and constants

ε ::= $\mathsf{EBrk} \mid \mathsf{EType} \mid \ldots$

c ::= $\ldots \mid \mathsf{TExcp} \mid \varepsilon$

Terms

t ::= $\ldots \mid \mathsf{toSum}\, x \mid \mathsf{mkNaV}\, x$

Boxes and atoms

b ::= $\mathbf{V}\, v \mid \mathbf{D}\, \varepsilon$

a ::= $b@L$

Figure 4. Syntax changes and extensions from $\lambda^{[\,]}$ to $\lambda^{[\,]}_{NaV}$

have a declassification construct in $\lambda^{[\,]}$, and if we added one, our non-interference results would hold only for programs that do not declassify. Addressing declassification is an interesting topic for future work; our hope is that we can adapt and reuse results from the static IFC setting [2, etc.].

4 Not-a-Values Formally

4.1 $\lambda^{[\,]}_{NaV}$: Calculus with NaVs

In $\lambda^{[\,]}_{NaV}$, we extend $\lambda^{[\,]}$ with NaV-based error handling. The extensions to the syntax are in Figure 4. We introduce *exception names* like EType and EBrk as constants; like the other constants they are both values and terms. Every atom in $\lambda^{[\,]}_{NaV}$ is a labeled *box*, where a box contains either a value ($\mathbf{V}\, v$) or a NaV (a delayed exception denoted $\mathbf{D}\, \varepsilon$). Type and bracket errors produce NaVs automatically. Programmers can create their own NaVs using the mkNaV operation, which turns an exception name into a NaV. Once created, NaVs propagate automatically: e.g., trying to call a NaV like a function yields the NaV. Since $\lambda^{[\,]}_{NaV}$ lacks exceptional control flow, there is no "catch" mechanism per se: instead, the toSum operation is used to check whether or not a given atom is a NaV. For the moment, we omit the stack and propagation traces that are also contained in NaVs—we will, however, consider them in §4.2 below.

Evaluating a $\lambda^{[\,]}_{NaV}$ program yields one of two possible outcomes: either it loops forever, or it terminates with an atom. In particular, $\lambda^{[\,]}_{NaV}$ has no fatal errors. We have proved in Coq that the big-step semantics of $\lambda^{[\,]}_{NaV}$ is equivalent to a small-step semantics satisfying strong progress[4].

The evaluation rules of $\lambda^{[\,]}_{NaV}$ are largely similar to $\lambda^{[\,]}$. We give only the most interesting rules in Figure 5. Rules ending in E signal errors, using a helper function *prEx* to propagate exceptions: when given a value, *prEx* returns EType to signal a type error; when given a NaV, *prEx* propagates it. For example, rule *NAppE* returns an EType NaV when the value in the function position is not a closure: $\rho \vdash (\mathsf{Inl}\, L)\,(\lambda x.x), pc \Downarrow (\mathbf{D}\, \mathsf{EType})@\bot, pc$. It propagates NaVs from the function position:

$$\rho \vdash (\mathsf{mkNaV}\,(\varepsilon))\,(\mathsf{Inr}\, L), pc \Downarrow (\mathbf{D}\, \varepsilon)@\bot, pc.$$

[4] Strong progress is, however, just a rough cut at formalizing robust error handling—a calculus can have progress without providing robust handling of errors. For example, it might loop forever on errors, only catch errors at the top-level, or always hide error messages.

$$\frac{\rho(x_1) = b@L \quad \mathit{tagOf}\; b \neq \mathsf{TFun}}{\rho \vdash (x_1\, x_2), pc \Downarrow (\mathbf{D}\,(\mathit{prEx}\; b))@\bot, pc \vee L}\ (\mathit{NAppE})$$

$$\frac{\rho(x) = b@L \quad \mathit{tagOf}\; b \neq \mathsf{TSum}}{\rho \vdash \mathsf{match}\ x\ \mathsf{with}\ \ldots, pc \Downarrow (\mathbf{D}\,(\mathit{prEx}\; b))@\bot, pc \vee L}\ (\mathit{NMatchE})$$

$$\frac{\rho(x) = (\mathbf{D}\,\varepsilon)@L}{\rho \vdash (\mathsf{tagOf}\; x), pc \Downarrow (\mathbf{D}\,\varepsilon)@L, pc}\ (\mathit{NTagOfE})$$

$$\frac{\begin{array}{l}\rho(x_1) = b_1@L_1 \quad \rho(x_2) = b_2@L_2 \\ \mathit{tagOf}\; b_1 \notin (\mathit{tagsArgs}\; \otimes)\end{array}}{\rho \vdash (x_1 \otimes x_2), pc \Downarrow (\mathbf{D}\,(\mathit{prEx}\; b_1))@(L_1 \vee L_2), pc}\ (\mathit{NBOpE1})$$

$$\frac{\begin{array}{l}\rho(x_1) = b_1@L_1 \quad \rho(x_2) = b_2@L_2 \\ \mathit{tagOf}\; b_1 \in (\mathit{tagsArgs}\; \otimes) \quad \mathit{tagOf}\; b_2 \notin (\mathit{tagsArgs}\; \otimes)\end{array}}{\rho \vdash (x_1 \otimes x_2), pc \Downarrow (\mathbf{D}\,(\mathit{prEx}\; b_2))@(L_1 \vee L_2), pc}\ (\mathit{NBOpE2})$$

$$\frac{\begin{array}{l}\rho(x) = (\mathbf{V}\; L)@L' \quad \rho \vdash t, (pc \vee L') \Downarrow b@L'', pc' \\ L'' \vee pc' \sqsubseteq L \vee (pc \vee L')\end{array}}{\rho \vdash \underline{x}[t], pc \Downarrow b@L, (pc \vee L')}\ (\mathit{NBrk})$$

$$\frac{\begin{array}{l}\rho(x) = (\mathbf{V}\; L)@L' \quad \rho \vdash t, (pc \vee L') \Downarrow b@L'', pc' \\ L'' \vee pc' \not\sqsubseteq L \vee (pc \vee L')\end{array}}{\rho \vdash \underline{x}[t], pc \Downarrow (\mathbf{D}\, \mathsf{EBrk})@L, (pc \vee L')}\ (\mathit{NBrkEBrk})$$

$$\frac{\rho(x) = b@L' \quad \mathit{tagOf}\; b \neq \mathsf{TLab}}{\rho \vdash \underline{x}[t], pc \Downarrow (\mathbf{D}\,(\mathit{prEx}\; b))@\bot, (pc \vee L')}\ (\mathit{NBrkE})$$

$$\frac{\rho(x) = b@L}{\rho \vdash \mathsf{labelOf}\; x, pc \Downarrow (\mathbf{V}\; L)@\bot, pc}\ (\mathit{NLabelOf})$$

$$\frac{\rho(x) = (\mathbf{V}\,\varepsilon)@L}{\rho \vdash \mathsf{mkNaV}\; x, pc \Downarrow (\mathbf{D}\,\varepsilon)@L, pc}\ (\mathit{NMkNaV})$$

$$\frac{\rho(x) = b@L \quad \mathit{tagOf}\; b \neq \mathsf{TExcp}}{\rho \vdash \mathsf{mkNaV}\; x, pc \Downarrow (\mathbf{D}\,(\mathit{prEx}\; b))@L, pc}\ (\mathit{NMkNaVE})$$

$$\frac{\rho(x) = (\mathbf{V}\; v)@L}{\rho \vdash \mathsf{toSum}\; x, pc \Downarrow (\mathbf{V}\,(\mathsf{Inl}\,((\mathbf{V}\; v)@\bot)))@L, pc}\ (\mathit{NToSumV})$$

$$\frac{\rho(x) = (\mathbf{D}\,\varepsilon)@L}{\rho \vdash \mathsf{toSum}\; x, pc \Downarrow (\mathbf{V}\,(\mathsf{Inr}\,(\mathbf{V}\,\varepsilon)@\bot))@L, pc}\ (\mathit{NToSumD})$$

Where

$$\begin{array}{rcl}\mathit{tagsArgs}\;(==) & = & \{\mathsf{TUnit}, \mathsf{TLab}, \mathsf{TTag}, \mathsf{TExcp}\} \\ \mathit{prEx}\,(\mathbf{V}\; v) & = & \mathsf{EType} \\ \mathit{prEx}\,(\mathbf{D}\,\varepsilon) & = & \varepsilon\end{array}$$

Note: Rules *NVar*, *NConst*, *NLet*, *NAbs*, *NApp*, *NBOp*, *NSum*, *NMatch*, *NTagOf*, *NGetPc* are the same as in $\lambda^{[\,]}$.

Figure 5. Evaluation relation for $\lambda^{[\,]}_{NaV}$

Rule *NAppE* raises the pc by the label on the function position, just like *NApp*. NaV-propagation rules treat the pc just like their success-path counterparts; NaVs do not introduce new control flow edges, so the pc raises just as it does in $\lambda^{[\,]}$. This is one of the advantages of NaVs over the more traditional mechanism based on catchable exceptions from §5, which has to raise the pc more often to account for exceptional control flow.

Rule *NBrkEBrk* applies when the body of a bracket yields

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

Atom equivalence
$b_1@L' \equiv_L b_2@L' \iff L' \sqsubseteq L \implies b_1 \equiv_L b_2$

Box and value equivalence
$\mathbf{V}\, c \equiv_L \mathbf{V}\, c$
$\mathbf{V}\,(C\; a_1) \equiv_L \mathbf{V}\,(C\; a_2) \iff a_1 \equiv_L a_2$
$\mathbf{V}\,\langle \rho_1,\, \lambda x.\, t\rangle \equiv_L \mathbf{V}\,\langle \rho_2,\, \lambda x.\, t\rangle \iff \rho_1 \equiv_L \rho_2$
$\mathbf{D}\,\varepsilon_1 \equiv_L \mathbf{D}\,\varepsilon_2 \iff \varepsilon_1 = \varepsilon_2$

Figure 6. $\lambda^{[\,]}_{NaV}$'s atom, box, and value equivalence (below label L)

Exceptions and constants
$\varepsilon ::= \mathsf{EBrk} \mid \mathsf{EType} \mid \ldots$
$c ::= \ldots \mid \mathsf{TExcp} \mid \varepsilon$

Terms
$t ::= \ldots \mid \mathsf{throw}\, x \mid \mathsf{try}\, t\, \mathsf{catch}\, x \Rightarrow t'$

Results
$res ::= a \mid \mathbf{T}\,\varepsilon$

Figure 7. Syntax extensions from $\lambda^{[\,]}$ to $\lambda^{[\,]}_{throw}$

a value that is labeled too high or a pc that is too high. (the precise condition is the same as in §3). The result of evaluating the bracket is discarded, and replaced with an EBrk NaV labeled with the label of the bracket. For example:

$$\rho \vdash \underline{\mathsf{low}}[\underline{\mathsf{high}}[\lambda x.x]], pc \Downarrow (\mathbf{D}\,\mathsf{EBrk})@\mathsf{low}, pc.$$

Since NaVs flow like data, throwing away the result of the bracket can hide errors:

$$\rho \vdash \underline{\mathsf{low}}[\underline{\mathsf{high}}[\mathsf{mkNaV}\,(\varepsilon)]], pc \Downarrow (\mathbf{D}\,\mathsf{EBrk})@\bot, pc$$

The NaV generated by the bracket completely hides the high-labeled NaV that the programmer constructed. This is crucial for soundness.

Finally, running $\mathsf{toSum}\, x$ will yield a Inl-tagged value if x holds a value (rule *NToSumV*); it will yield a Inr-tagged exception constant if x holds a NaV (rule *NToSumD*). In either case, the label is moved from x to the tag.

Non-interference for $\lambda^{[\,]}_{NaV}$

The equivalence $\equiv_L$ changes slightly to account for NaVs as shown in Figure 6. Otherwise, the definitions of environment equivalence and non-interference remain the same as in §3.

Theorem 2 (Non-interference for $\lambda^{[\,]}_{NaV}$). *Given a label L, a term t, environments ρ_1 and ρ_2, and a starting* pc *label pc, if: 1) $\rho_1 \equiv_L \rho_2$, 2) $\rho_1 \vdash t, pc \Downarrow a_1, pc'_1$, 3) $\rho_2 \vdash t, pc \Downarrow a_2, pc'_2$, and 4) $pc'_1 \sqsubseteq L$ or $pc'_2 \sqsubseteq L$ then $pc'_1 = pc'_2$ and $a_1 \equiv_L a_2$.*

Non-interference in $\lambda^{[\,]}_{NaV}$ is termination insensitive, just like $\lambda^{[\,]}$. But unlike $\lambda^{[\,]}$, the non-interference theorem in $\lambda^{[\,]}_{NaV}$ is *error sensitive*. Since $\lambda^{[\,]}_{NaV}$ has evaluation rules for all potential errors, programs that have type or bracket errors (or other, user-defined exceptions) still enjoy non-interference. This is, as far as we are aware, the first error-sensitive non-interference proof in the purely dynamic IFC setting where there are no fatal errors whatsoever.

4.2 Adding Stack and Propagation Traces to $\lambda^{[\,]}_{NaV}$

To more closely model our implementation, we have extended $\lambda^{[\,]}_{NaV}$ so that NaVs also carry stack and propagation traces and we have reproved non-interference and progress in Coq. This extension is fairly straightforward. We instrument the semantics to keep the current stack trace alongside the environment. For this we add a new kind of constant loc, drawn from a set of program locations. A new expression $\mathsf{trace}^{loc}\; t$ indicates that when executing t, we should push the location loc onto the current stack trace; when t returns, we pop loc off the stack trace. Whenever a NaV is created, it stores a copy of the then-current stack trace. When a traced computation returns a NaV, we add loc to the NaV's propagation trace. Finally, given a NaV, toSum returns a triple holding the exception name, the stack trace, and the propagation trace. Our formalization in Coq also extends $\lambda^{[\,]}_{NaV}$ with pairs, which make it particularly easy to encode the stack trace as a language value—a list of locations, where cons cells are pairs. These lists are labeled $\bot$ throughout.

It might seem surprising that the stack and propagation traces are not protected using IFC labels. In a language without declassification each NaV is protected enough (by its explicit label and by the pc) so that no information can be leaked via the traces inside. We expect that if we added declassification we would need to explicitly label each new cons cell we add to the traces with the then-current pc label. However, for the simple calculus we consider here this explicit labeling is not necessary. We have proved in Coq that this extension of $\lambda^{[\,]}_{NaV}$ is non-interfering.

5 Catchable Exceptions

5.1 $\lambda^{[\,]}_{throw}$: Calculus Where Brackets Delay Exceptions

Our third calculus, $\lambda^{[\,]}_{throw}$, demonstrates an alternative design that eschews delayed exceptions where possible, resulting in a language that has a more traditional treatment of exceptions and control flow. However, as noted in §1 and §2.3, we cannot soundly allow exceptions to propagate outside of brackets. Thus, in $\lambda^{[\,]}_{throw}$, brackets catch and delay all exceptions. The syntax extensions compared to $\lambda^{[\,]}$ are presented in Figure 7. We add two new term forms: $\mathsf{throw}\, x$, which raises an exception x, and a standard try/catch construct: $\mathsf{try}\, t\, \mathsf{catch}\, x \Rightarrow t'$. In $\lambda^{[\,]}_{throw}$ delayed exceptions are only produced by brackets. To keep the calculus simple, we have brackets return tagged values: Inl means success and Inr means failure. Although they are represented as tagged variants, values of the form $\mathsf{Inr}\,(\mathbf{V}\,\varepsilon)@\bot$ are a simple form of delayed exceptions. In §5.2 we propose a more complex calculus that lifts this simplification by adding primitive delayed exceptions to $\lambda^{[\,]}_{throw}$.

The evaluation relation for $\lambda^{[\,]}_{throw}$ differs slightly from $\lambda^{[\,]}$

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

and $\lambda^{[\,]}_{NaV}$: evaluation produces a *result* rather than an atom. Results are either atoms or uncaught exceptions $\mathbf{T}\,\varepsilon$. We can relate $\lambda^{[\,]}_{NaV}$ and $\lambda^{[\,]}_{throw}$'s approach to error handling by thinking about the set of elements produced by evaluation: let V, E, and L denote the sets of values, errors, and labels, respectively. Evaluation in $\lambda^{[\,]}_{NaV}$ yields an atom and a pc label, in the set $((V + E) \times L) \times L$. Evaluation in $\lambda^{[\,]}_{throw}$ yields a result and a pc label, in the set $((V \times L) + E) \times L$. In both $\lambda^{[\,]}_{NaV}$ and $\lambda^{[\,]}_{throw}$, everything is protected by the pc. Both values and errors get their own labels in $\lambda^{[\,]}_{NaV}$; in $\lambda^{[\,]}_{throw}$, errors do not get their own label—they are protected only by the pc. Since in $\lambda^{[\,]}_{throw}$ errors are propagated via the control flow of the program the pc needs to raise more often than in $\lambda^{[\,]}$ and $\lambda^{[\,]}_{NaV}$.

The semantics of brackets is the most interesting part of $\lambda^{[\,]}_{throw}$. When the label check at the end of a bracket fails (since the value and/or pc are labeled too high) the result of the bracket is an appropriately labeled bracket error, i.e., a delayed exception:

$$\rho \vdash \underline{\mathsf{low}}[\underline{\mathsf{high}}[\lambda x.x]], pc \Downarrow (\mathbf{V}\,(\mathsf{Inr}\,(\mathbf{V}\,\mathsf{EBrk})@\bot))@\mathsf{low}, pc$$

We have already established that it would be unsound for brackets to throw exceptions—brackets must instead catch and delay all exceptions. There are two cases for this: If the exception caught by the bracket is thrown when the pc is low enough, we can reveal the source of the failure—after raising the exception's label to the bracket's label:

$$\rho \vdash \underline{\mathsf{high}}[\mathsf{throw}\,(\varepsilon)], pc \Downarrow (\mathbf{V}\,(\mathsf{Inr}\,(\mathbf{V}\,\varepsilon)@\bot))@\mathsf{high}, pc$$

But if an exception is thrown with a too high pc, then it would be unsound to reveal the exact failure that occurred. In this case, we hide the precise cause of the failure and return an EBrk. For example, if $t = (\lambda x.\mathsf{throw}\,(\varepsilon))$ then:

$$\rho \vdash \underline{\mathsf{low}}[\underline{\mathsf{high}}[t]\,(\lambda x.x)], pc \Downarrow (\mathbf{V}\,(\mathsf{Inr}\,(\mathbf{V}\,\mathsf{EBrk})@\bot))@\mathsf{low}, pc$$

The following example shows that it is necessary to hide the exceptions thrown with a too high pc:

```
match low[if h then throw Ex else ()] with
 | Inl _ => ()
 | Inr Ex => lref := true
 | Inr EBrk => lref := false
```

This kind of error-hiding also occurs in $\lambda^{[\,]}_{NaV}$, though it is less obvious. Since $\lambda^{[\,]}_{NaV}$ does not have exceptional control flow, rule *NBrkEBrk* hides any bracket result that is labeled too high, whether it is a value or a NaV.

The semantics of `try/catch` is also interesting. First, since the exception handler itself can raise exceptions we cannot soundly restore the pc at the end of a `catch` block. In $\lambda^{[\,]}_{throw}$ only ending brackets are guaranteed to be control flow join points, so brackets are the only construct to restore the pc. Second, in $\lambda^{[\,]}_{throw}$ the pc does not raise before executing the exception handler—this is an important difference compared to the LIO exception handling mechanism [27] that is further discussed in §9. Due to space constraints, we omit the formal details of $\lambda^{[\,]}_{throw}$ (see [1]). Like for the other calculi in this paper, we have proved in Coq that $\lambda^{[\,]}_{throw}$ satisfies non-interference. Just like for $\lambda^{[\,]}_{NaV}$, non-interference for $\lambda^{[\,]}_{throw}$ is error sensitive and termination insensitive.

5.2 Adding Primitive Delayed Exceptions to $\lambda^{[\,]}_{throw}$

The brackets in $\lambda^{[\,]}_{throw}$ caught exceptions and, for simplicity, produced labeled tagged variants: $(\mathbf{V}\,(\mathsf{Inl}\,a))@L$ for success and $(\mathbf{V}\,(\mathsf{Inr}\,(\mathbf{V}\,\varepsilon)@\bot))@L$ for failure. With a bit more work, we can make delayed exceptions primitive, as in $\lambda^{[\,]}_{NaV}$. We have devised another calculus we call $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$, in which evaluation produces results like in $\lambda^{[\,]}_{throw}$, but atoms contain boxes like in $\lambda^{[\,]}_{NaV}$—i.e., $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ evaluation produces elements in the set $(((V+E)\times L)+E)\times L$. Brackets must still catch exceptions, but the various bracket rules from $\lambda^{[\,]}_{throw}$ now return atoms instead of tagged values.

Finally, there were many choices to be made about how to produce and propagate exceptions in $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$. Like in $\lambda^{[\,]}_{throw}$, we chose that type errors cause catchable exceptions (not delayed exceptions like in $\lambda^{[\,]}_{NaV}$). Additionally, the user can raise her own catchable exceptions using throw. Delayed exceptions are only produced by brackets and later propagate as follows: For parametric operations, which do not inspect their arguments, we chose to be lax with respect to delayed exceptions (like in $\lambda^{[\,]}_{NaV}$). So, in $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$, calling a function with a delayed exception argument will succeed and bind the formal argument to the delayed exception. On the other hand, non-parametric operations need to fail when one of their arguments is a delayed exception. In $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ we chose to fail by rethrowing the delayed exception. This is different than $\lambda^{[\,]}_{NaV}$ where we were making the result *be* the delayed exception. This is, however, quite similar to the exception mechanism very recently proposed for LIO [27] (see §9 for a precise comparison). Due to space constraints, we omit the details of $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ (they are available in an online appendix [1]). Like for the other calculi in this paper, we have proved in Coq that $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ satisfies non-interference.

6 Encodings

We have devised global translations between four of the five calculi presented above (see Figure 8), and used Coq extraction together with extensive random testing with QuickCheck [9] to validate that these translations are semantics preserving. Based on this evidence, we conjecture that $\lambda^{[\,]}_{NaV}$, $\lambda^{[\,]}_{throw}$, and $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ can all encode each other, which is an indication that the error handling mechanism based on NaVs and the ones based on catchable exceptions have similar expressive power. The fact that we can faithfully encode $\lambda^{[\,]}_{NaV}$ into $\lambda^{[\,]}_{throw}$ might be a bit surprising, since in $\lambda^{[\,]}_{throw}$ the pc raises more often that in $\lambda^{[\,]}_{NaV}$. However, we use brackets to bring the pc back down to the same level

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

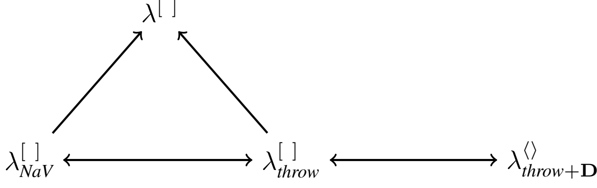

Figure 8. Encodings

it would have been in $\lambda^{[\,]}_{NaV}$, and our translation can always generate the expression needed to compute the label on each of these brackets. Moreover, we conjecture that $\lambda^{[\,]}_{NaV}$, $\lambda^{[\,]}_{throw}$, and $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ can all be encoded in $\lambda^{[\,]}$, but, because of brackets, these encodings are more complicated than the standard "error monad" encodings. We hope to prove these conjectures formally in the future. The fact that these encodings are whole-program translations restricts their practical applicability. They are, however, interesting for studying the theoretical properties of the calculi and for transferring non-interference results instead of reproving them for each calculus, as recently done for state by Austin et al. [7].

7 Bulletproofing the Max Server

We can now return to the simple max server from §1 and show how to protect it against poison-pill attacks using the two mechanisms we have described—NaVs and catchable exceptions. For illustrating NaVs we will work in an extension of $\lambda^{[\,]}_{NaV}$, letting all parametric operations be NaV-lax, with the exception of the sequencing operator (semicolon). The original `max_server_loop` executes as follows when receiving a poison pill `(1@high,2@high)@low`. Branching on the result of `(1@high) <= (2@high)` raises the `pc` to `high` to account for potential implicit flows and the result of `process_max` is `2@high`, which the server attempts to send over the `low` channel. The `send` fails and returns a "send error" NaV labeled `low`; then the whole sequence returns the same NaV. The tail call to `max_server_loop` no longer happens, effectively killing the server.

A first step in fixing the server is to make sure the tail call always executes, regardless of what happens with the `send`. We replace the NaV-strict semicolon with a NaV-lax let, branch on whether the result of the send is a success or a failure, and log the error, in case of failure. No matter what, we do the tail call.

```
fun rec max_server_loop_n1 () =
  let res = send out (process_max (recv in))
  match toSum res with
   | Inl () => max_server_loop_n1()
   | Inr x => send log x; max_server_loop_n1 ()
```

This is not sufficient, however, for protecting the server. The `pc` of the server raises when comparing the secret numbers in the poison pill, but never goes back down, preventing the server from answering future requests. In this case the server does not crash and keeps processing requests, but the high `pc` prevents it from ever sending an answer back. The solution to this problem is simple: wrap the `recv`, the call to `process_max`, and the `send` into a bracket that restores the `pc` back to its original `low` state.

```
fun rec max_server_loop_n2 () =
  let res = low[send out (process_max (recv in))]
  match toSum res with
   | Inl () => max_server_loop_n2 ()
   | Inr x => send log x; max_server_loop_n2 ()
```

Since the send always returns a `low` result (either a unit or a NaV) the bracket can be annotated with `low`, which means that matching its result below does not change the `pc`. This variant of the server is immune to poison pills.

Protecting the max server with catchable exceptions is quite similar. Wrapping the body of the server loop in a `try/catch`, as done in `max_server_loop'` (see §1), is not enough to protect the server, because, as in `max_server_loop_n1` above, the `pc` is never restored after the comparison, preventing the server from answering future requests. The solution is again to use a bracket, this time instead of a `try/catch` block.

```
fun rec max_server_loop_t1 () =
  let res = low[send out (process_max (recv in))]
  (match res with
    | Inl () => ()
    | Inr x => send log res);
  max_server_loop_t1 ()
```

The bracket also catches all exceptions, but additionally it restores the `pc` to its original `low` state after each request, no matter how high it got while processing the request. If something fails while processing the request, the error is delayed by the bracket and labeled `low`, so the server can write it to a public log. While this implementation is immune to poison pills, the server never answers requests that cause failures, which causes those clients to block. If we want to make the server always respond to requests, we need to take the `send` out of the bracket. Since the label on the bracket is `low` the send cannot cause an IFC violation.

```
fun rec max_server_loop_t2 () =
  let res = low[process_max (recv in)]
  (match res with
    | Inl  m => send out m
    | Inr x => send log res; send out "error");
  max_server_loop_t2 ()
```

8 Implementation and Experience

We have implemented NaVs in a new dynamically typed functional language called Breeze, with purely dynamic FIFC, declassification, concurrency and channel-based communication in the style of Concurrent ML, and higher-order dynamic contracts and coercions (annotations that look like contracts but can alter inputs). To help prevent untrusted code from leaking secrets via covert channels, Breeze also includes a mechanism for discretionary access control called *clearance* [26]—a label that acts as an upper bound on the

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

pc. Declassification (i.e., lowering labels on values or the pc) and raising the clearance are distinct privileged operations.

The Highs & Lows of NaVs

We ported the language's existing standard library and test suite, consisting of 8336 lines of code originally designed for "stop the world" errors in an earlier version of Breeze. We also singled out applications demanding robustness, including a heterogeneously labeled key-value store and a web server. Our initial experience exposed both positive and negative aspects of NaVs. The web server turned out to be particularly easy to protect—the server simply checks whether the serialized response it is about to send to the browser is a NaV, and if so, sends an error page instead.

We expected to stumble over cases where it would be difficult to predict a bracket's label, but in practice this was not an issue for application-level code. In general, we found that NaVs mitigate some pitfalls of traditional exceptions, while adding a few new ones. One shortcoming common to both mechanisms is that error paths in the code are difficult to exercise exhaustively, especially errors involving IFC labels. Another is the ease with which code review can overlook missing error-handling code. Property-based random testing [9] with a focus on label coverage is an interesting direction for future work.

Early debugging impediments in the implementation involved imprecise or insufficiently detailed error messages. One specific source of pain was that NaVs generated from failed contracts did not clearly note which contract had failed. Accurate provenance for such errors makes a world of difference when debugging.

Mixing NaVs and Imperative Code

The Breeze standard library implements reference cells using (labeled) channels, with the invariant that the channel backing a ref cell contains exactly one value. The reference assignment operation is the only place where this invariant is temporarily broken. Prior to implementing NaVs, putting a mislabeled value into the channel resulted in a fatal error that terminated the whole program. With the introduction of NaVs, this previously fatal error was silently swallowed, leaving the ref cell in an inconsistent (empty) state. When other code eventually tried to read from the empty channel, it failed with a cryptic error that made no mention of the NaV generated by the channel send.

This example illustrates two issues exposed by the NaV scheme. First is the danger of ignoring NaVs: it is natural to ignore the result of an operation that returns a unit value, but doing so in the NaV world can result in dropping errors on the floor. (One can make the same mistake with exceptions, but an empty `catch` block is more obviously suspicious than simply discarding a value, at least with Breeze's current ML-like syntax.) Thus, programmers must take extra care not to accidentally ignore potential NaVs in imperative code. The second issue is the need to protect stateful invariants. While a NaV records the history of its origin and propagation, it only propagates via dataflow. So if a discarded NaV results in stateful invariants being broken, this will manifest as a failure that does not cite the culpable NaV. In this particular case, ignoring the channel-send NaV led to a low-level interpreter deadlock.

Managing NaV Propagation

Reference cells provide a concise example of how imperative code must deal with NaVs to avoid or restore broken invariants. But the need to reason about NaVs also applies to purely functional code, as discussed in §2.4.

Two variants of a finite map abstraction illustrated different subtleties about writing code—even pure code—in the presence of FIFC and NaVs. We started with a straightforward implementation using an (unordered) list of key-value pairs. This implementation was written before NaVs existed. When running on homogeneously labeled data, it worked smoothly. However, keys with unexpectedly high labels would trigger the generation of NaVs, which could corrupt the spine of the list. Thus corrupted, a putative "finite map" value would behave innocuously under some operations; for example, insertion into such a map would succeed. However, membership queries would sometimes fail, *depending on the insertion order of the keys*. This violation of the finite map abstraction stemmed from a failure to account for additional invariants required in a language with NaVs and FIFC. Simply tightening the finite map's interface to strictly enforce the contract `{x|x==x}` constituted a partial fix. This ensures that keys must not be NaVs, and must be comparable using `==`. However, a complete fix must also add a label bound to that contract, which in turn requires modifying the interface to the finite map abstraction itself. We do not yet know how common such invasive changes are when bulletproofing existing code.

In this and other examples, we have found it is usually better to fail early, by marking function arguments as NaV-strict, than to run a function when the assumptions it was written under may not hold. Beyond function arguments, it is also useful to reason about constraining the set of possible function return values. We might imagine a contract which says "this value cannot be a NaV," but what happens when that contract fails? All errors are signaled via NaVs, but producing another NaV is obviously unhelpful.

We dealt with this issue in the implementation of a map with heterogeneously labeled keys. Clients of this library look up values in the map by providing a comparison predicate on keys, which can be imbued with *authority* to inspect data labeled as off-limits to the map library itself. A correct comparison function will always return booleans, never NaVs. To be robust, the map library cannot simply trust the client—it must enforce this invariant.

Our solution is to wrap the client's function with a

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

coercion—effectively a contract that might alter its input—that replaces NaVs with a default value. The coercion states a NaV-management *policy* in a centralized and declarative manner à la contracts. When client code passes a comparison function to the heterogeneously labeled map, the library specifies the function's intended behavior using the concrete syntax `Any=>Any=>(Bool 'ReplacingNAVsWith' false)`. If the comparison function is not permitted to read a map entry's key, the map library will treat the client's resulting information-flow error as a response of "this isn't the right key"—precisely the behavior we want.

The Continuum From Lax To Strict, In Practice

As presented, $\lambda^{[\,]}_{NaV}$ is NaV-lax—that is, the only control flow arising from NaVs is due to pattern matches explicitly written by the programmer. However, as discussed above, unrestricted propagation of NaVs can lead to subtle bugs, and manually writing out every explicit check would be cumbersome. We extended our contract system with coercions that make functions NaV-strict: programmers can choose where NaV strictness happens (with its associated pc raises), rather than setting a language-wide policy.

We have found in practice that two different informal reasoning principles apply when making NaV-strictness/laxness explicit. In a generally NaV-lax landscape, we add strictness in order to preclude NaVs from appearing—typically to enforce application-specific invariants. In contrast, when strictness is the default, we add laxness annotations in places where we do not know or control what the label of a value will be. Whenever a function deals with a polymorphic value, the arrow for that argument should be lax. Usually the programmer's aim is not to allow NaVs per se, but rather to avoid the pc raises which accompany strictness checks.

NaNs and Null Pointer Exceptions

The idea of NaVs obviously invites comparison to null pointers and IEEE 754 floating-point NaN values [16]. Names aside, NaVs are actually only superficially similar to NaNs. First, NaNs are restricted to a single type—double-precision floating point numbers—whereas NaVs are injected into every value type of the language. As a result, NaVs propagate freely and are not subject to premature coercion. For instance, when we compare a NaV to an integer, the result is a NaV, whereas with IEEE, the result is constrained to be a (poorly chosen) boolean. The other crucial difference is that a NaN carries little *provenance* about its origins or propagation history.

Null pointer exceptions carry stack traces, but of a different nature than those carried by NaVs. The exception's stack trace records the point at which the null pointer was erroneously dereferenced, which is often far from where the culpable null pointer was generated. In contrast, NaVs carry a stack trace pinpointing the location in the code where the NaV was generated and a trace of its subsequent journey; we have found these debugging aids very useful in practice.

NaVs do have some downsides when compared to traditional exceptions. One is the added worry of properly managing strictness, both for data structures and function arguments. Another is the NaV hiding phenomenon presented in §4.1. Finally, while NaVs avoid issues of premature coercion, they also require more pervasive reasoning about where they might be generated, precisely because NaVs are not limited to any particular type of value.

9 Related Work

The work that is most closely related to ours is LIO [26], a recent dynamic IFC library for Haskell. LIO is the first FIFC language with public labels and a construct called `toLabeled` that is very similar to brackets. Stefan et al. [27] have very recently extended the core LIO calculus with catchable exceptions that get delayed (and labeled) by brackets and reactivated when unlabeled. This independently discovered exception mechanism is quite similar to our $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ calculus (see §5.2). We additionally provide a systematic exploration of the entire solution space and thoroughly investigate a more radical design based on NaVs. The subtle but important differences between LIO exceptions and $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ shed further light on the design space. While LIO brackets also delay exceptions, they do not hide the error message of exceptions thrown in too high contexts. Instead the throw-time pc is remembered inside delayed exceptions. The throw-time pc is not used when exceptions are reactivated, but when they are caught by a `try`/`catch` block. In the exception handler, the pc is raised by the throw time pc of the caught exception. For this to be sound, delayed exceptions cannot be inspected— they must be reactivated and then caught. That is, the throw-time pc inside a delayed exception is not a public label in LIO.

LIO also features clearance—a label that acts as an upper bound on the pc. In order to ensure that the clearance bounds the pc, `catch` blocks do not catch exceptions that would bring the pc higher than the clearance. Strictly speaking, only maximally privileged code (clearance `top`) can catch all exceptions in LIO. In order to prevent poison pills in code that cannot catch all exceptions, a low clearance can be set for a `try` block (e.g., the body of the server loop) in order to control the pc of all exceptions thrown in that block. This ensures that all exceptions that might occur can also be caught with the privilege of the code—including exceptions caused by (inadvertent) attempts to exceed the clearance. By contrast, the exception handling mechanism we propose in §5 does not rely on clearance in order to control how high the pc goes on a `catch` block—in $\lambda^{[\,]}_{throw}$ and $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ the pc does not raise at all in a `catch` block. The downside is that, in order to preserve soundness, brackets sometimes hide error messages, replacing them with `EBrk`. However, in Breeze running brackets with a low clearance prevents the

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

pc from ever getting high enough to require error message hiding. While our choices subtly differ from Stefan et al. [27], clearance helps both approaches.

One can imagine further bridging the gap between $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ and LIO exceptions by making failed brackets in $\lambda^{\langle\rangle}_{throw+\mathbf{D}}$ remember the cause of the error together with the pc at the time the error occurred in an opaque package. For instance, a `low` bracket that fails because of an EType exception thrown in a `high` context could return (EBrk $\mathsf{EType}_{\mathsf{high}}$)@low. To preserve soundness, the $\mathsf{EType}_{\mathsf{high}}$ part can only be inspected with a special construct that raises the pc to high before returning EType—otherwise, the label on the inner exception would leak.

Other than LIO, the only FIFC system with an error handling mechanism was recently proposed by Hedin and Sabelfeld [15] in the context of a JavaScript core language with objects, higher-order functions, catchable exceptions, and dynamic code evaluation. They use special *upgrade instructions* [5] to gain precision (flow sensitivity), and introduce a similar upgrade mechanism for their new exception security label. This leads to an exception handling mechanism that is very different from what we propose in §5. In their setting labels are not publicly observable, and exceptions are never delayed, but IFC violations are fatal errors. Moreover, in order to preserve soundness, their system has to treat exceptions raised in high security contexts as fatal IFC violations, as well. Such limitations seem unavoidable when retrofitting FIFC to an existing programming language with exceptions (barring invasive changes to the semantics). We avoid such limitations by exploring new language designs that safely combine reliable error handling and FIFC.

Like our annotations on brackets, the upgrade instructions of Hedin and Sabelfeld do not need to be correct in order to achieve non-interference. This means that unsound techniques like random testing and symbolic execution can be used to infer such upgrade instructions, as recently proposed by Birgisson et al. [8]. Bracket annotations and upgrade instructions are in this respect very different from the *oracles* used by the early FIFC systems [14], [22], [23], since the soundness of these early systems crucially depended on the soundness of a static analysis tool providing information about the branches not taken.

The proof technique used to formally show non-interference for the calculi in this paper was devised by Austin and Flanagan [4]. They were amongst the first [4], [25] to discover that non-interference can be enforced by a conservative purely-dynamic mechanism, without resorting to oracles. Russo and Sabelfeld [23] have studied the trade-offs between static and dynamic IFC, especially in terms of flow-sensitivity: allowing the label of mutable references to change on updates opens up a label channel. As usual, preventing leaks can be done by imposing additional restrictions, which either prevent secret information from leaking into this label channel (e.g., no-sensitive-upgrade [4]) or from leaking out of this label channel (e.g., permissive upgrades [5]). Since Breeze has no legacy constraints, we could easily avoid the flow-sensitivity problem for references completely: we require the label on all references (and channels) to be fixed at creation time.

Error handling is problematic beyond the FIFC setting. In the static IFC setting, exceptions are a significant source of imprecision [3], [20]. In order to keep their typing rules for arrays precise, Deng and Smith [11] replace throwable exceptions with default values (out-of-bounds array reads yield 0) and silent failures (out-of-bounds array writes are simply skipped). King et al. [18] report that exceptions are responsible for the overwhelming majority of false alarms in JLift. It will be interesting to see if NaVs are an acceptable error handling mechanism for new languages with static IFC.

10 Conclusion and Future Work

In this paper we show that FIFC does not have to punt on availability. We propose the first error handling mechanisms that are sound in this setting, while allowing all errors to be recoverable, even IFC violations. Although quite different at the surface, the main ingredients of the two mechanisms we propose are the same: public labels and delayed exceptions. We show formally that these two ingredients are sufficient for making all errors recoverable—and we believe that they are also necessary for achieving this in a sound and usable system. Our practical experience with NaVs suggests that the issues introduced by delayed exceptions can be tricky, but are not insurmountable. We have proposed mitigations for most of the difficulties we have encountered with NaVs, and they seem to help in practice. Fundamentally, identifying useful invariants and writing good error recovery code is hard even without the additional constraints imposed by sound FIFC—we do not claim that adding FIFC into the mix will magically make error handling easy. What we propose here are mechanisms that make recovering from all errors *possible*, even in a FIFC setting.

Future work Our current practical experience with NaVs is limited to running Breeze code in an interpreter. We are, however, working on two compilers for Breeze, one targeting a conventional architecture, and another targeting a novel architecture with hardware support for FIFC and NaVs [10], [12]. In both cases NaVs promise to simplify compilation compared to catchable exceptions, especially if the compiler has the freedom to produce imprecise error messages, stack and propagation traces [17].

Acknowledgments This work arose in the context of the SAFE project [10], [12]. We are grateful to B. Montagu and R. Pollack for their help with the formal proofs and for contributing to useful discussions. We thank A. Chudnov, A. DeHon, J. Hsu, S. Iyer, A. Russo, H. Shrobe,

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.

E. Silkensen, D. Stefan, and D. Wittenberg and the anonymous reviewers for their insightful comments. We also thank M. Hicks, D. Hedin, E. Kohler, G. Malecha, D. Mazières, and O. Shivers for interesting discussions. This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This work has also been supported by the National Science Foundation under grant 0915671 *Contracts for Precise Types*.

References

[1] All your IFCException are belong to us. Online appendix, Coq formalization, Breeze interpreter, libraries, and sample applications, are all available at http://www.crash-safe.org/node/23.

[2] A. Askarov and A. C. Myers. A semantic framework for declassification and endorsement. In *19th European Symposium on Programming (ESOP)*, March 2010.

[3] A. Askarov and A. Sabelfeld. Catch me if you can: permissive yet secure error handling. In *Workshop on Programming Languages and Analysis for Security*, PLAS. 2009.

[4] T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *Workshop on Programming Languages and Analysis for Security*, PLAS. 2009.

[5] T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th Workshop on Programming Languages and Analysis for Security*, PLAS. 2010.

[6] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th Symposium on Principles of Programming Languages*, POPL, 2012.

[7] T. H. Austin, C. Flanagan, and M. Abadi. A functional view of imperative information flow. To appear in 10th ASIAN Symposium on Programming Languages and Systems (APLAS 2012), Springer, December 2012.

[8] A. Birgisson, D. Hedin, and A. Sabelfeld. Boosting the permissiveness of dynamic information-flow tracking by testing. In *Proceedings of 17th European Symposium on Research in Computer Security*, ESORICS. 2012.

[9] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP. 2000.

[10] A. DeHon, B. Karel, T. F. Knight, Jr., G. Malecha, B. Montagu, R. Morisset, G. Morrisett, B. C. Pierce, R. Pollack, S. Ray, O. Shivers, J. M. Smith, and G. Sullivan. Preliminary design of the SAFE platform. In *6th Workshop on Programming Languages and Operating Systems*, PLOS, October 2011.

[11] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *17th IEEE Computer Security Foundations Workshop*. 2004.

[12] U. Dhawan, A. Kwon, E. Kadric, C. Hriţcu, B. C. Pierce, J. M. Smith, G. Malecha, G. Morrisett, T. F. Knight, Jr., A. Sutherland, T. Hawkins, A. Zyxnfryx, D. Wittenberg, P. Trei, S. Ray, G. Sullivan, and A. DeHon. Hardware support for safety interlocks and introspection. In *SASO Workshop on Adaptive Host and Network Security*, September 2012.

[13] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazières, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *20th Symposium on Operating Systems Principles*, SOSP. 2005.

[14] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *20th Computer Security Foundations Symposium*, CSF. 2007.

[15] D. Hedin and A. Sabelfeld. Information-flow security for a core of JavaScript. In *25th IEEE Computer Security Foundations Symposium*, CSF. 2012.

[16] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*, July 1985. ANSI/IEEE Std 754-1985.

[17] S. L. P. Jones, A. Reid, F. Henderson, C. A. R. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI. 1999.

[18] D. King, B. Hicks, M. Hicks, and T. Jaeger. Implicit flows: Can't live with 'em, can't live without 'em. In *4th International Conference on Information Systems Security*, ICISS, 2008.

[19] M. N. Krohn, A. Yip, M. Z. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *21st Symposium on Operating Systems Principles*, SOSP. October 2007.

[20] G. Malecha and S. Chong. A more precise security type system for dynamic security tests. In *5th Workshop on Programming Languages and Analysis for Security*, PLAS. 2010.

[21] T. Murray, D. Matichuk, M. Brassil, P. Gammie, T. Bourke, S. Seefried, C. Lewis, X. Gao, and G. Klein. sel4: from general purpose to a proof of information flow enforcement. In *34th IEEE Symposium on Security and Privacy*. 2013. To appear.

[22] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *Proceedings of the Symposium on Security and Privacy*, SP. 2007.

[23] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the 23rd Computer Security Foundations Symposium*, CSF. 2010.

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

[25] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *Ershov Memorial Conference*. 2009.

[26] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in Haskell. In *Proceedings of the 4th Symposium on Haskell*. 2011.

[27] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible Dynamic Information Flow Control in the Presence of Exceptions. *ArXiv e-print 1207.1457*, July 2012.

[28] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. *Communications of the ACM*, 54(11):93–101, 2011.

[29] L. Zheng and A. C. Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84, 2007.

Authorized licensed use limited to: IEEE Xplore. Downloaded on September 27,2024 at 02:26:36 UTC from IEEE Xplore. Restrictions apply.