Information Borrowing

Hazel Torek[1] Hemant Gouni[2] Jonathan Aldrich[2]

IWACO 2026

[1]Clemson University

[2]Carnegie Mellon University

Borrowing for memory

```
fn main() {
  let s = String::from("Hello World!");
  steals(s);
  // s is no longer accessible :(
}

fn steals(s: String) {
  println!("{}", s);
}
```

Borrowing for memory

```
fn main() {
  let s = String::from("Hello World!");
  borrows(&s);
  // s is returned back to caller! :)
}

fn borrows(s: &String) {
  println!("{}", s);
}
```

Borrowing for memory

Borrowing enables memory safety by passing other functions **temporary access** to memory.

Borrowing for memory

Borrowing enables memory safety by passing other functions **temporary access** to memory.

Can this idea be generalized to **information**?

Borrowing for information

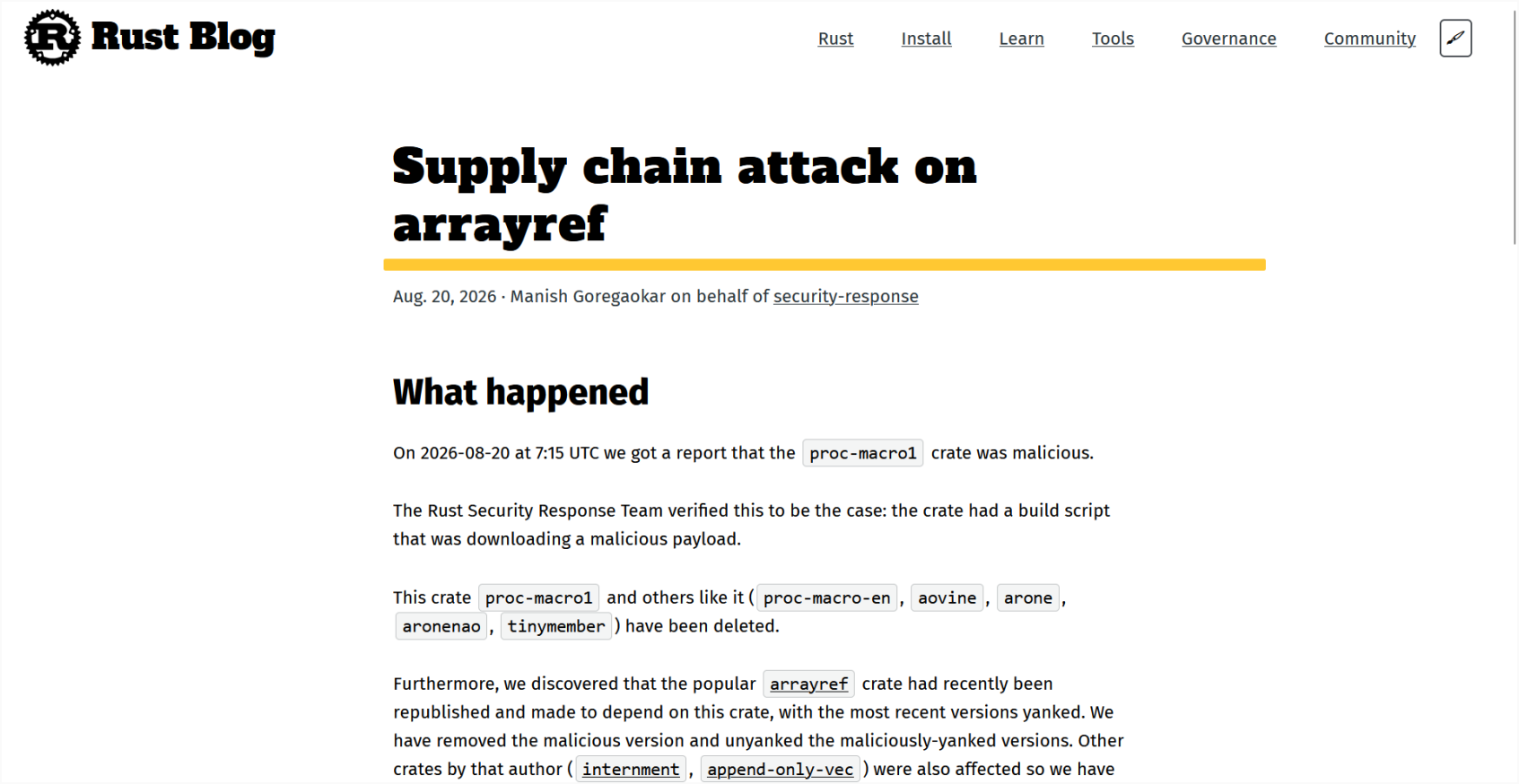
If I pass a piece of **information** to a function, how do I know it will only flow into the return value without getting leaked out through side effects?

Borrowing for information

If I pass a piece of **information** to a function, how do I know it will only flow into the return value without getting leaked out through side effects?

Why is this something you care about?

Supply chain attacks



Rust Blog

Rust Install Learn Tools Governance Community

Supply chain attack on arrayref

Aug. 20, 2026 · Manish Goregaokar on behalf of security-response

What happened

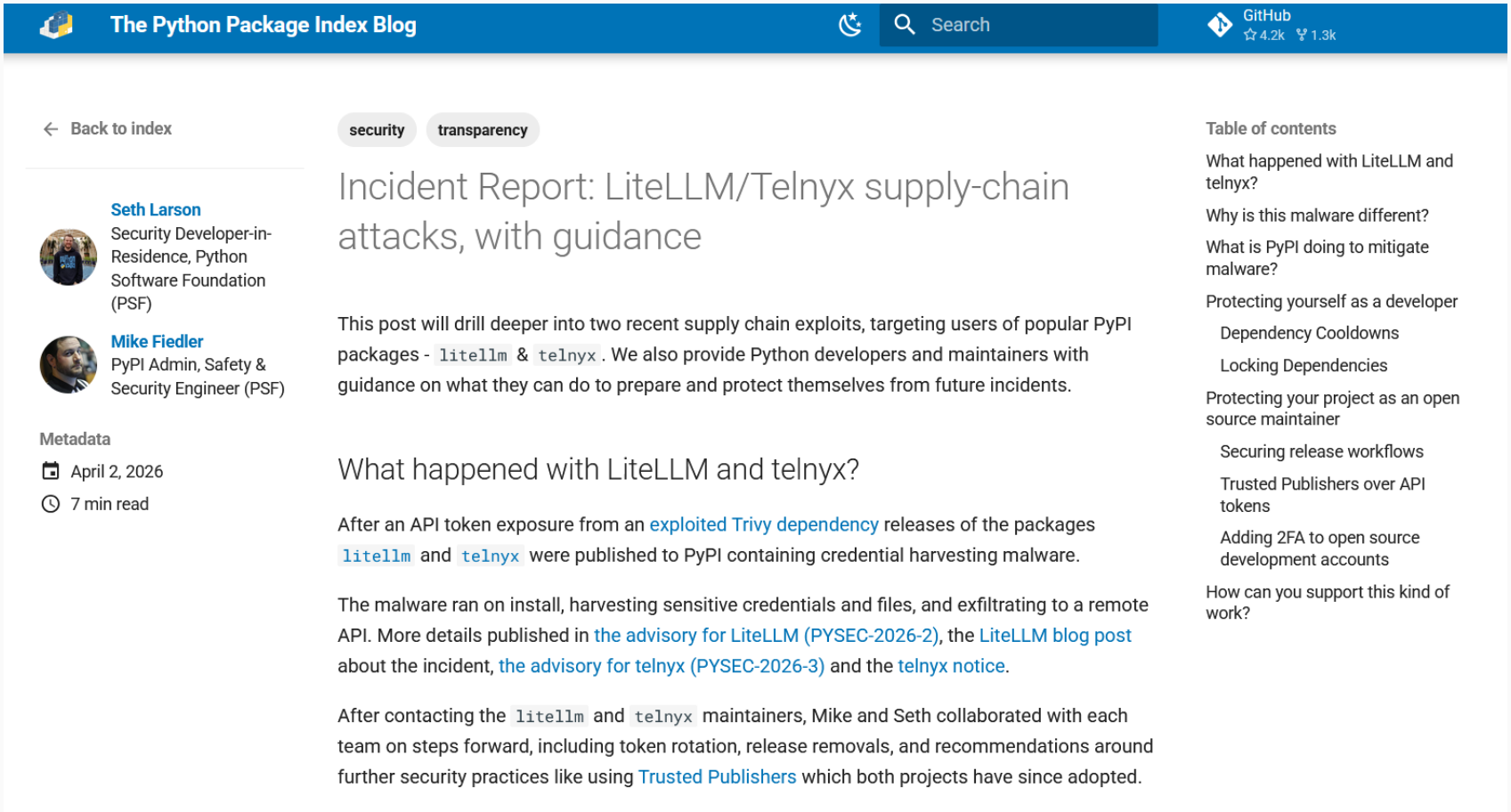
On 2026-08-20 at 7:15 UTC we got a report that the `proc-macro1` crate was malicious.

The Rust Security Response Team verified this to be the case: the crate had a build script that was downloading a malicious payload.

This crate `proc-macro1` and others like it (`proc-macro-en` , `aovine` , `arone` , `aronenao` , `tinymember`) have been deleted.

Furthermore, we discovered that the popular `arrayref` crate had recently been republished and made to depend on this crate, with the most recent versions yanked. We have removed the malicious version and unyanked the maliciously-yanked versions. Other crates by that author (`internment` , `append-only-vec`) were also affected so we have

Supply chain attacks



The Python Package Index Blog

Search

GitHub ☆ 4.2k ⑂ 1.3k

← Back to index

Seth Larson
Security Developer-in-Residence, Python Software Foundation (PSF)

Mike Fiedler
PyPI Admin, Safety & Security Engineer (PSF)

Metadata

April 2, 2026

7 min read

security transparency

Incident Report: LiteLLM/Telnyx supply-chain attacks, with guidance

This post will drill deeper into two recent supply chain exploits, targeting users of popular PyPI packages - `litellm` & `telnyx`. We also provide Python developers and maintainers with guidance on what they can do to prepare and protect themselves from future incidents.

What happened with LiteLLM and telnyx?

After an API token exposure from an exploited Trivy dependency releases of the packages `litellm` and `telnyx` were published to PyPI containing credential harvesting malware.

The malware ran on install, harvesting sensitive credentials and files, and exfiltrating to a remote API. More details published in the advisory for LiteLLM (PYSEC-2026-2), the LiteLLM blog post about the incident, the advisory for telnyx (PYSEC-2026-3) and the telnyx notice.

After contacting the `litellm` and `telnyx` maintainers, Mike and Seth collaborated with each team on steps forward, including token rotation, release removals, and recommendations around further security practices like using Trusted Publishers which both projects have since adopted.

Table of contents

- What happened with LiteLLM and telnyx?
- Why is this malware different?
- What is PyPI doing to mitigate malware?
- Protecting yourself as a developer
  - Dependency Cooldowns
  - Locking Dependencies
- Protecting your project as an open source maintainer
  - Securing release workflows
  - Trusted Publishers over API tokens
  - Adding 2FA to open source development accounts
- How can you support this kind of work?



Platform Resources Customers Partners Company

Massive PyPI Supply Chain Attack Harvests Cloud Credentials via Python Startup Hooks



The Orca Research Pod | Jun 08, 2026

A coordinated supply chain attack targeting PyPI has compromised 26 packages (37 malicious wheel files) across bioinformatics, graph ML, deep-learning, and developer tooling ecosystems. Dubbed the “Hades Campaign” and attributed to the Miasma/Shai-Hulud threat lineage, the attack exploits a legitimate Python feature to execute malicious code at interpreter startup, aggressively harvesting cloud credentials and exfiltrating tokens from compromised environments. Due to the breadth of affected packages and the severity of credential exposure, immediate package removal and credential rotation are required.

About the Vulnerability: Hades Campaign

The attack leverages Python .pth files, a mechanism that allows import statements to execute automatically when the Python interpreter initializes. Compromised package releases ship a *-setup.pth file containing an

Motivation for information borrowing

How do we use types to specify the **non-escape** of sensitive data from **untrusted** third-party functions? How do we ensure that sensitive parameters flow only to the return value and not elsewhere?

Motivation for information borrowing

How do we use types to specify the **non-escape** of sensitive data from **untrusted** third-party functions? How do we ensure that sensitive parameters flow only to the return value and not elsewhere?

- We can manually inspect the implementations, but this clearly doesn't scale well.

Motivation for information borrowing

How do we use types to specify the **non-escape** of sensitive data from **untrusted** third-party functions? How do we ensure that sensitive parameters flow only to the return value and not elsewhere?

- We can manually inspect the implementations, but this clearly doesn't scale well.
- We can use effect systems to find where side effects are happening, but this doesn't tell us *what* data is being leaked.

Password validator

We have a candidate for a password, but we want to check whether it is sufficiently strong using this function from a third-party library.

```
val valid : string -> bool
let valid = fun pw ->
  len pw <= 10 && hasSpecialChar pw
```

```
import valid
. . .
```

Password validator

We have a candidate for a password, but we want to check whether it is sufficiently strong using this function from a third-party library.

```
val valid : string -> bool
let valid = fun pw ->  ⟵ third-party library
  len pw <= 10 && hasSpecialChar pw
```

```
import valid
. . .
```

Password validator

We have a candidate for a password, but we want to check whether it is sufficiently strong using this function from a third-party library.

```
val valid : string -> bool
let valid = fun pw ->  ⟵ third-party library
  send pw;
  len pw <= 10 && hasSpecialChar pw
```

```
import valid
...
```

Revisiting the password validator

```
val len : string -> int
val hasSpecialChar : string -> bool

val valid : string -> bool
let valid = fun pw ->
  len pw >= 10 && hasSpecialChar pw
```

Revisiting the password validator

```
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> bool  ⟵ info-borrowed type
let valid = fun pw ->
  len *pw >= 10 && hasSpecialChar *pw
```

Revisiting the password validator

```
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> bool
let valid = fun pw ->
  len *pw >= 10 && hasSpecialChar *pw
      ↑ dereferencing info-borrowed values into strings
```

Revisiting the password validator

To maintain the rules of the information borrow, a dereferenced value cannot be passed to a function that performs IO.

```
val send : string -> [io] unit
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> bool
let valid = fun pw ->
  send *pw;
  len *pw >= 10 && hasSpecialChar *pw
```

Revisiting the password validator

To maintain the rules of the information borrow, a dereferenced value cannot be passed to a function that performs IO.

```
val send : string -> [io] unit  ⟵ annotated effects
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> bool
let valid = fun pw ->
  send *pw;
  len *pw >= 10 && hasSpecialChar *pw
```

Revisiting the password validator

To maintain the rules of the information borrow, a dereferenced value cannot be passed to a function that performs IO.

```
val send : string -> [io] unit
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> bool
let valid = fun pw ->
  send *pw;
  len *pw >= 10 && hasSpecialChar *pw
```

Revisiting the password validator

To maintain the rules of the information borrow, a dereferenced value cannot be passed to a function that performs IO.

```
val send : string -> [io] unit
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> bool
let valid = fun pw ->
  send *pw; ⚠ ⟵ effect could leak borrowed value
  len *pw >= 10 && hasSpecialChar *pw
```

Revisiting the password validator

But functions can perform IO that doesn't leak borrowed parameters!

```
val send : string -> [io] unit
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> [io] bool
let valid = fun pw ->
  print "innocent log message"
  len *pw >= 10 && hasSpecialChar *pw
```

Revisiting the password validator

But functions can perform IO that doesn't leak borrowed parameters!

```
val send : string -> [io] unit
val len : string -> int
val hasSpecialChar : string -> bool

val valid : &string -> [io] bool
let valid = fun pw ->
  print "innocent log message"  ⟵ does allow IO here!
  len *pw >= 10 && hasSpecialChar *pw
```

How does it work?

Information flow

Information flow type systems are used for reasoning about **non-interference**, which means that the type system has to faithfully track information.

Two modalities for information flow

From parametric information flow [1], we use the **open modality** $\bigcirc_\varphi(-)$ to track effects and the **closed modality** $\bullet_\varphi(-)$ to protect information.

We extend parametric IFC with effects.

Open modality

The open modality tracks effects.

```
val log : string -> [io] unit
let log s = fun s ->
  print s
```



```
val log : string -> ◯io(unit)
let log s = fun s ->
  print s
```

Open modality

The open modality tracks effects.

```
val log : string -> [io] unit
let log s = fun s ->
  print s
```



```
val log : string -> ○io(unit)
let log s = fun s ->
  print s
```

Closed modality

The closed modality protects information.

```
val pw : ●secret(string)
let pw = seal "Password123"

val n : int
let n = len pw
```

Closed modality

The closed modality protects information.

```
val pw : ●secret(string)
let pw = seal "Password123"

val n : int
let n = len pw  ⚠  ⟵ not sealed at secret level!
```

Closed modality

The closed modality protects information.

```
val pw : ●secret(string)
let pw = seal "Password123"

val secure_n : ●secret(int)
let secure_n = unseal pw in x.
  seal len(x)
```

Closed modality

The closed modality protects information.

```
val pw : ●secret(string)
let pw = seal "Password123"

val secure_n : ●secret(int)
let secure_n = unseal pw in x.
  seal len(x)
```

Quantifiers

We use quantifiers to introduce and **scope** new labels, which we can instantiate at a function's call site.

`val id :` $\forall\alpha.$ $\bullet_\alpha$`(string) -> `$\bullet_\alpha$`(string)`
`let id = fun x -> x`

`(id (seal "Password123!"))[secret]`
`(id "Hello, World!")[`$\varnothing$`]`

Quantifiers

We use quantifiers to introduce and **scope** new labels, which we can instantiate at a function's call site.

`val id :` $\forall\alpha.$ $\bullet_\alpha$`(string) -> `$\bullet_\alpha$`(string)`
`let id = fun x -> x`

`(id (seal "Password123!"))[secret]` $\longleftarrow$ $\bullet_{\texttt{secret}}$`(string)`
`(id "Hello, World!")[`$\varnothing$`]`

Quantifiers

We use quantifiers to introduce and **scope** new labels, which we can instantiate at a function's call site.

```
val id : ∀α. ●α(string) -> ●α(string)
let id = fun x -> x

(id (seal "Password123!"))[secret]
(id "Hello, World!")[∅]  ⟵ ●∅(string) ≅ string
```

Elaboration at function definition

State sealing checks that no effects that conflict with the borrow are taking place.

```
val valid : &string -> bool
let valid = fun pw ->
  len *pw >= 10 && hasSpecialChar *pw
```



```
val valid : ∀α. ●α(string) -> ●α(bool)
let valid = fun pw ->
  unseal pw in x .
    seal ((len x) >= 10 && hasSpecialChar x)
```

Elaboration at function definition

State sealing checks that no effects that conflict with the borrow are taking place.

```
val valid : &string -> bool
let valid = fun pw ->
  len *pw >= 10 && hasSpecialChar *pw
```



```
val valid : ∀α. ●α(string) -> ●α(bool)
let valid = fun pw ->
  unseal pw in x .
    seal ((len x) >= 10 && hasSpecialChar x)
```

Elaboration at function definition

State sealing checks that no effects that conflict with the borrow are taking place.

```
val valid : &string -> bool
let valid = fun pw ->
  len *pw >= 10 && hasSpecialChar *pw
```



```
val valid : ∀α. ●α(string) -> ●α(bool)
let valid = fun pw ->
  unseal pw in x .
    seal ((len x) >= 10 && hasSpecialChar x)
```

Elaboration at function definition

State sealing checks that no effects that conflict with the borrow are taking place.

```
val valid : &string -> bool
let valid = fun pw ->
  len *pw >= 10 && hasSpecialChar *pw
```



```
val valid : ∀α. ●α(string) -> ●α(bool)
let valid = fun pw ->
  unseal pw in x .
    seal ((len x) >= 10 && hasSpecialChar x)
```

Elaboration at function definition

State sealing checks that no effects that conflict with the borrow are taking place.

```
val valid : &string -> bool
let valid = fun pw ->
  len *pw >= 10 && hasSpecialChar *pw
```



```
val valid : ∀α. ●α(string) -> ●α(bool)
let valid = fun pw ->
  unseal pw in x .
    seal ((len x) >= 10 && hasSpecialChar x)
```

Elaboration at call site

Since α is universally quantified, when we call `valid` we can instantiate $\alpha = \varnothing$ so the returned value is of type `bool`!

```
let pw : [secret] string = seal "Password123"
valid pw
valid "Hello, World!"
```

$$\Downarrow$$

```
let pw : ●secret(string) = seal "Password123"
(valid pw)[secret]
(valid "Hello, World!")[∅]
```

Elaboration at call site

Since α is universally quantified, when we call `valid` we can instantiate $\alpha = \varnothing$ so the returned value is of type `bool`!

```
let pw : [secret] string = seal "Password123"
valid pw
valid "Hello, World!"
```



```
let pw : ●secret(string) = seal "Password123"
(valid pw)[secret]  ⟵ ●secret(string)
(valid "Hello, World!")[∅]
```

Elaboration at call site

Since α is universally quantified, when we call `valid` we can instantiate $\alpha = \varnothing$ so the returned value is of type `bool`!

```
let pw : [secret] string = seal "Password123"
valid pw
valid "Hello, World!"
```

$$\Downarrow$$

```
let pw : ●secret(string) = seal "Password123"
(valid pw)[secret]
(valid "Hello, World!")[∅]  ⟵ ●∅(string) ≅ string
```

Advantages of information borrowing

- **Simple specifications**: don't need to create new labels in your system-wide specification to prevent leakage

Advantages of information borrowing

- **Simple specifications**: don't need to create new labels in your system-wide specification to prevent leakage
- **Automatic declassification**: data is automatically declassified to the level provided at input, rather than needing to manually perform an unsafe operation

Another application: mutable references in IFC

Information borrowing = mutable references?

Information borrowing also gives us a strategy for handling references in parametric IFC!

Information borrowing = mutable references?

Information borrowing also gives us a strategy for handling references in parametric IFC!

- **Reading** requires accessing a piece of information, which we can represent using the closed modality.

Information borrowing = mutable references?

Information borrowing also gives us a strategy for handling references in parametric IFC!

- **Reading** requires accessing a piece of information, which we can represent using the closed modality.
- **Writing** to a reference induces an effect, which we can represent using the open modality.

Mutable references example

Our approach to mutable references uses the same **state sealing** mechanism enabling information borrowing!

```
val logFile : ref string

val processData : string -> ●secret(string) -> unit
let processData = fun a b ->
  write logFile a
  write logFile b
```

Mutable references example

Our approach to mutable references uses the same **state sealing** mechanism enabling information borrowing!

```
val logFile : ref string

val processData : string -> ●secret(string) -> unit
let processData = fun a b ->
  write logFile a
  write logFile b
```

Mutable references example

Our approach to mutable references uses the same **state sealing** mechanism enabling information borrowing!

```
val logFile : ref string

val processData : string -> ●secret(string) -> unit
let processData = fun a b ->
  write logFile a
  write logFile b
```

Mutable references example

Our approach to mutable references uses the same **state sealing** mechanism enabling information borrowing!

```
val logFile : ref string

val processData : string -> ●secret(string) -> unit
let processData = fun a b ->
  write logFile a
  write logFile b
```

Mutable references example

Our approach to mutable references uses the same **state sealing** mechanism enabling information borrowing!

```
val logFile : ref string

val processData : string -> ●secret(string) -> unit
let processData = fun a b ->
  write logFile a
  write logFile b ⚠ ⟵ type mismatch
```

Mutable references example

Our approach to mutable references uses the same **state sealing** mechanism enabling information borrowing!

```
val logFile : ref string

val processData : string -> ●secret(string) -> unit
let processData = fun a b ->
  write logFile a
  unseal b in x.
   write logFile x
```

Mutable references example

Our approach to mutable references uses the same **state sealing** mechanism enabling information borrowing!

```
val logFile : ref string

val processData : string -> ●secret(string) -> unit
let processData = fun a b ->
  write logFile a
  unseal b in x.
   write logFile x ⚠ ⟵ effect conflicts with unsealing
```

Existing approaches to state in IFC

Existing approaches to mutable state in IFC use a **program counter** [2], but it's not clear how to cleanly combine this with features like modular declassification or extensions like information borrowing.

Conclusion

Contributions

- New way to specify non-escape with IFC
- Mutable references for parametric IFC

Future Work

- Prove and mechanize non-interference
- Other ways to use IFC?

ctorek@clemson.edu

I am applying for Ph.D. programs this cycle!


Hemant Gouni, Frank Pfenning, and Jonathan Aldrich.
The Duality of Information Flow: Reconciling Robust Downgrading with Non-Interference, July 2026.
arXiv:2607.17445 [cs.PL].


Andrew K. Hirsch and Ethan Cecchetti.
Giving semantics to program-counter labels via secure effects.
Proceedings of the ACM on Programming Languages, 5(POPL):1–29, January 2021.