Information Borrowing

Hazel Torek, Hemant Gouni, and Jonathan Aldrich

Motivation

When using **untrusted code**, it is necessary to **establish security boundaries** for interactions with sensitive data. For example, the following program that checks password strength also maliciously leaks the prospective password.

Untrusted Third-Party Library Code

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

🔍 Can't identify possible leakage of sensitive info by looking at types.

🐛 Safety issue: this function leaks the password via side effect!

Identifying such concerns by inspecting the implementation scales poorly. We introduce **information borrowing**, which captures **non-escape** at the type level. It ensures borrowed information is confined until returned.

Pass by Dereference and Pass by Borrow

Borrowed types are written &T, and dereferences are written `*t` to extract the underlying term type T out of an information borrowed term of type &T. Using these constructs, we provide two approaches to fixing our motivating example:

Pass by Dereference

```
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)
```

✨ We can see in the type signature that the parameter is **borrowed**, so it won't be leaked!

🔓 The borrowed variable is **dereferenced**; additional checks on doing so cause send to fail.

Pass by Borrow

🔒 The additional functions borrow their parameters, so the types match and safety is ensured.

✅ The call to send fails because of a type mismatch. The other functions respect the sensitive data!

```
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)
```

The dereferencing approach is more general but limited because it forbids called functions from performing side effects unrelated to the secret data. The borrowing approach provides finer granularity; since called functions also use borrowing, I/O and other side effects cannot leak info.

Elaboration

Information flow type systems are used to prove security properties of programs. We repurpose three key constructs from recent work to enable information borrowing:

1. The **quasi-closed modality** ● represents borrows
2. The **quasi-open modality** ◯ tracks effects
3. **Quantifiers** ∀ create and scope dependency variables.

The following program shows the dereferencing approach elaborated into the underlying IFC language.

Pass by Dereference in IFC

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

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

⚠️ Type error: send has return type ◯[io](unit), which violates the rules for unseal.

The quasi-open modality type ◯[io](unit) represents the I/O side effect of send.

The quantified dependency variable α scopes the information borrow to the body of the function.

The valid function uses the quasi-closed modality types ●[α](string) and ●[α](bool) to protect the parameter from leaking anywhere but the return type.

The unseal operation takes the parameter of type ●[α](string) and binds it as a string variable x, ensuring the body is free of effects which would leak it. This causes the error at the call to send.

Mutable References

The machinery we've presented for information borrowing generalizes to handle mutable references in information flow languages, since writing to a reference is an **effect** tracked via the quasi-open modality. Reading can be controlled via the quasi-closed modality.

- Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2025. Structural Information Flow: A Fresh Look at Types for Non-interference. Proc. ACM Program. Lang. 9, OOPSLA2 (October 2025), 3954–3980.
- Hemant Gouni, Frank Pfenning, and Jonathan Aldrich. 2026. The Duality of Information Flow: Reconciling Robust Downgrading with Non-Interference.