MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

Secure extraction of verified effectful F⋆ programs to ML

Cezar-Constantin Andrici

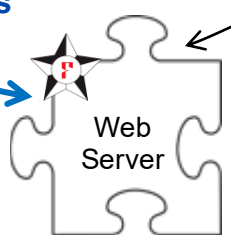
joint work with

Danel Ahman, Ștefan Ciobâcă, **Cătălin Hriţcu**, Ruxandra Icleanu, Guido Martínez, Éric Tanter, Abigail Pribisova, Exequiel Rivas, Théo Winterhalter

April 13, 2026, Aarhus University

Proof-oriented language **F★ offers strong guarantees**

Annotated with
refinement types and
pre- and postconditions

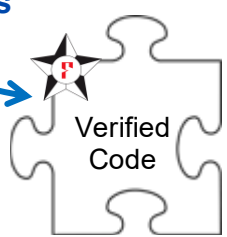


satisfies security property π

(e.g., π is “the web server's private key is not leaked”)

Extract the verified code to run it

Annotated with
refinement types and
pre- and postconditions



Extraction

satisfies security property π

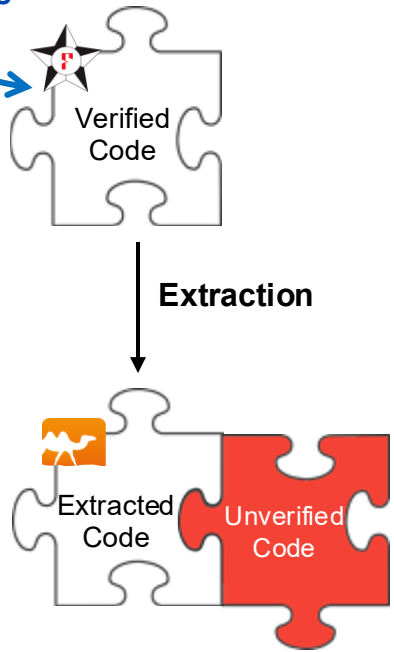
(e.g., π is “the web server's private key is not leaked”)

satisfies security property π

In practice, the extracted verified code gets mixed with unverified code

Annotated with **refinement types** and **pre- and postconditions**

e.g., **EverCrypt,** cryptocraphic provider extracted to C to be integrated in Mozilla Firefox, the Linux kernel, etc.



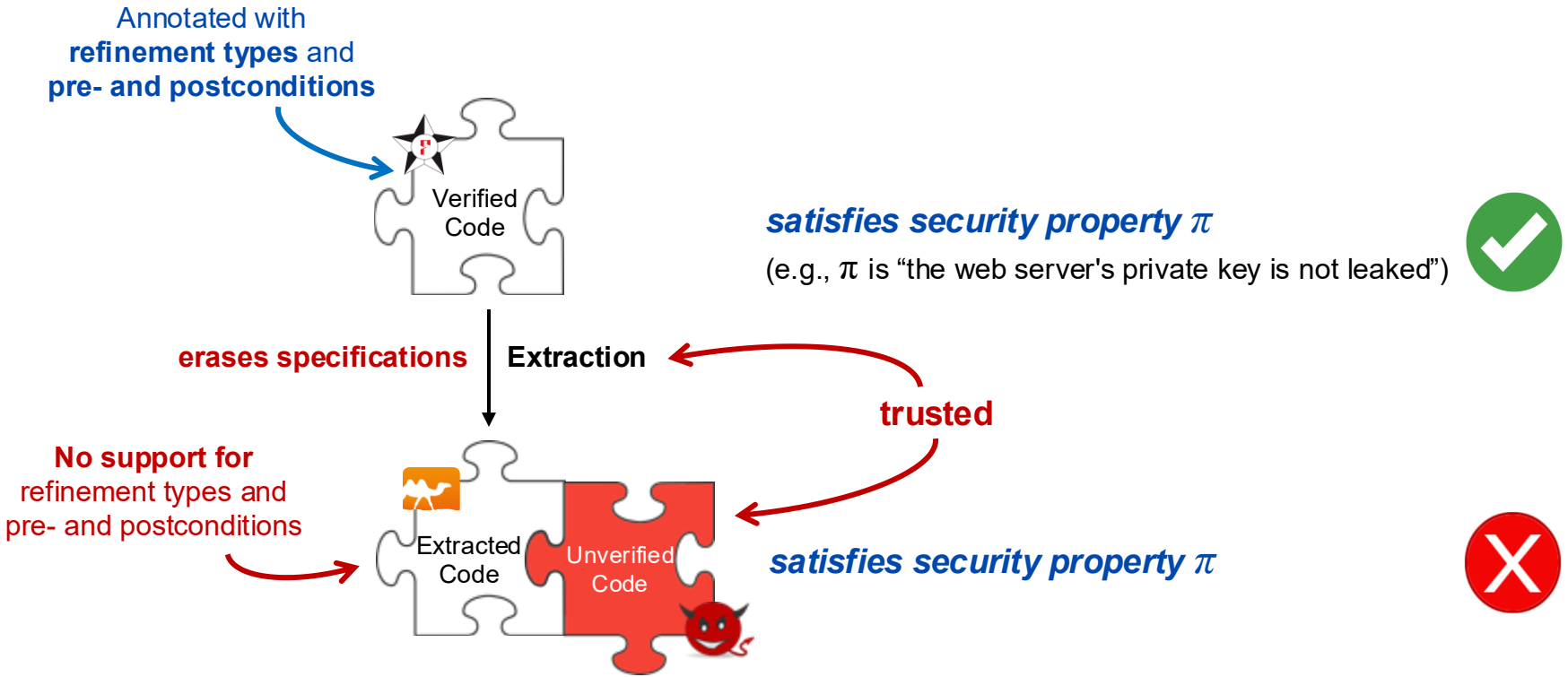
satisfies security property π

(e.g., π is “the web server's private key is not leaked”)

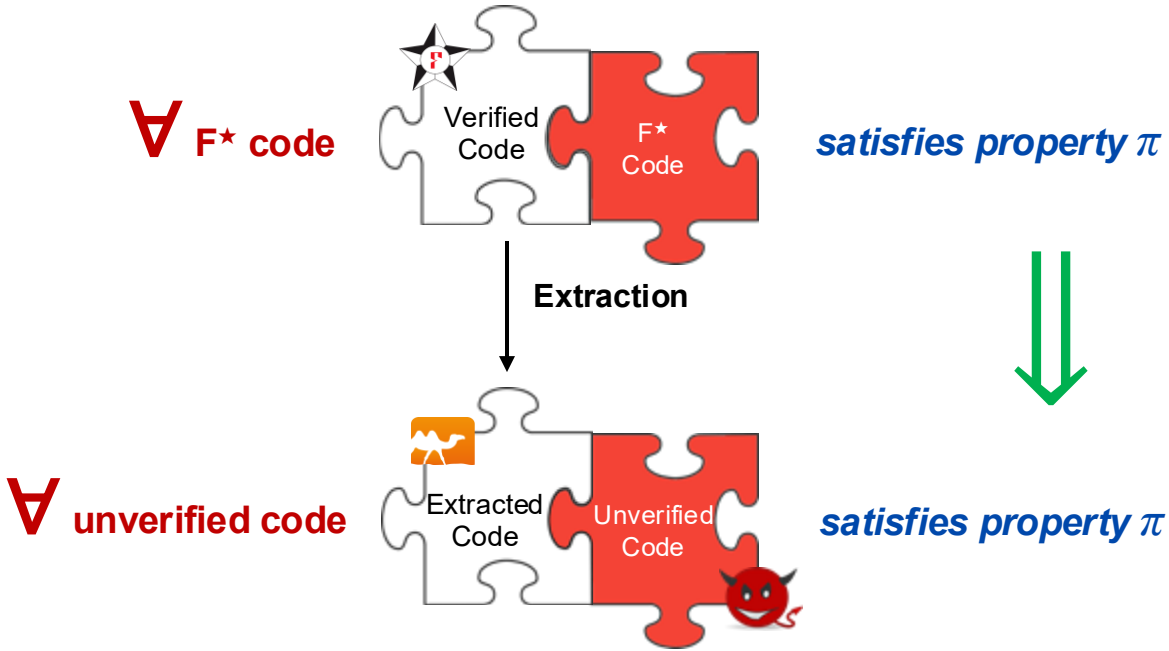
satisfies security property π

?

Mixing verified code with unverified code can be unsound and insecure

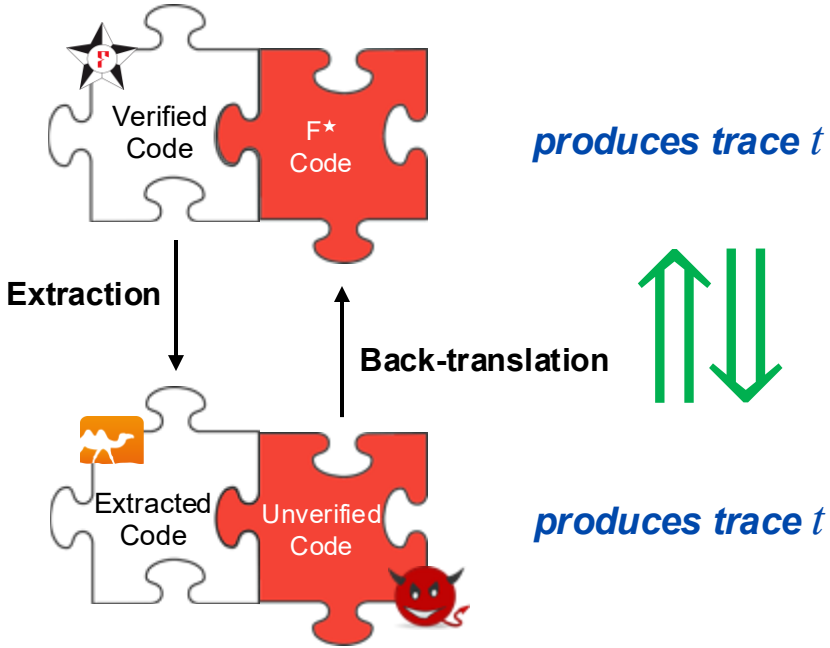


Sound mixing of verified code with unverified code



Secure mixing of verified code with unverified code

∀ unverified code ∃ F⋆ code ∀ traces t



Robust Relational Hyperproperty Preservation

Strongest criterion of Abate et al. (CSF'19)

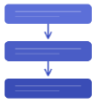
Stronger than full abstraction.
Extraction preserves:

- Observational equivalence
- Noninterference
- Trace properties

Towards a verified secure extraction framework for verified effectful F★ programs to ML

Sound dynamic enforcement of specifications on the interface between verified-unverified code

Proof of secure extraction
Preserves observational equivalence, noninterference and trace properties



Vertically composable
Transitive criteria that allow for extensions to lower level languages

Written and verified in F★

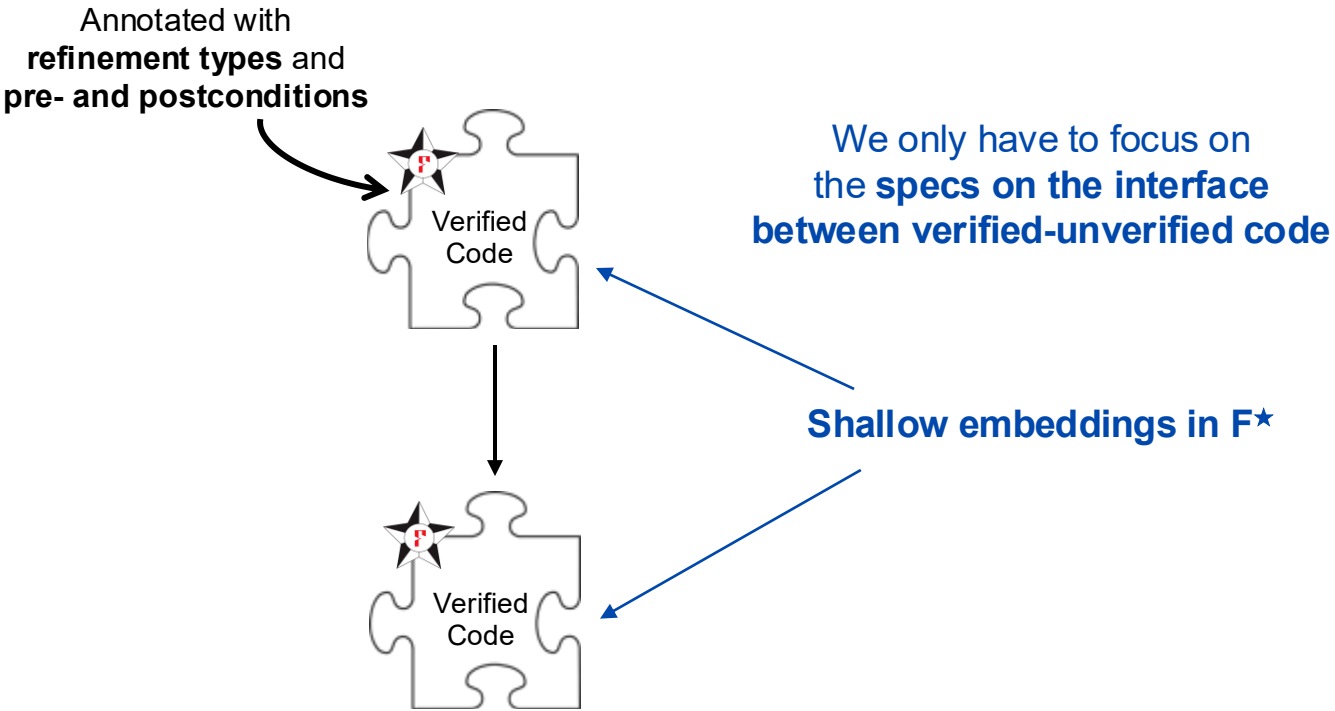
Results towards secure extraction

Secure compilation for verified IO programs
(Andrici et al. POPL'24)

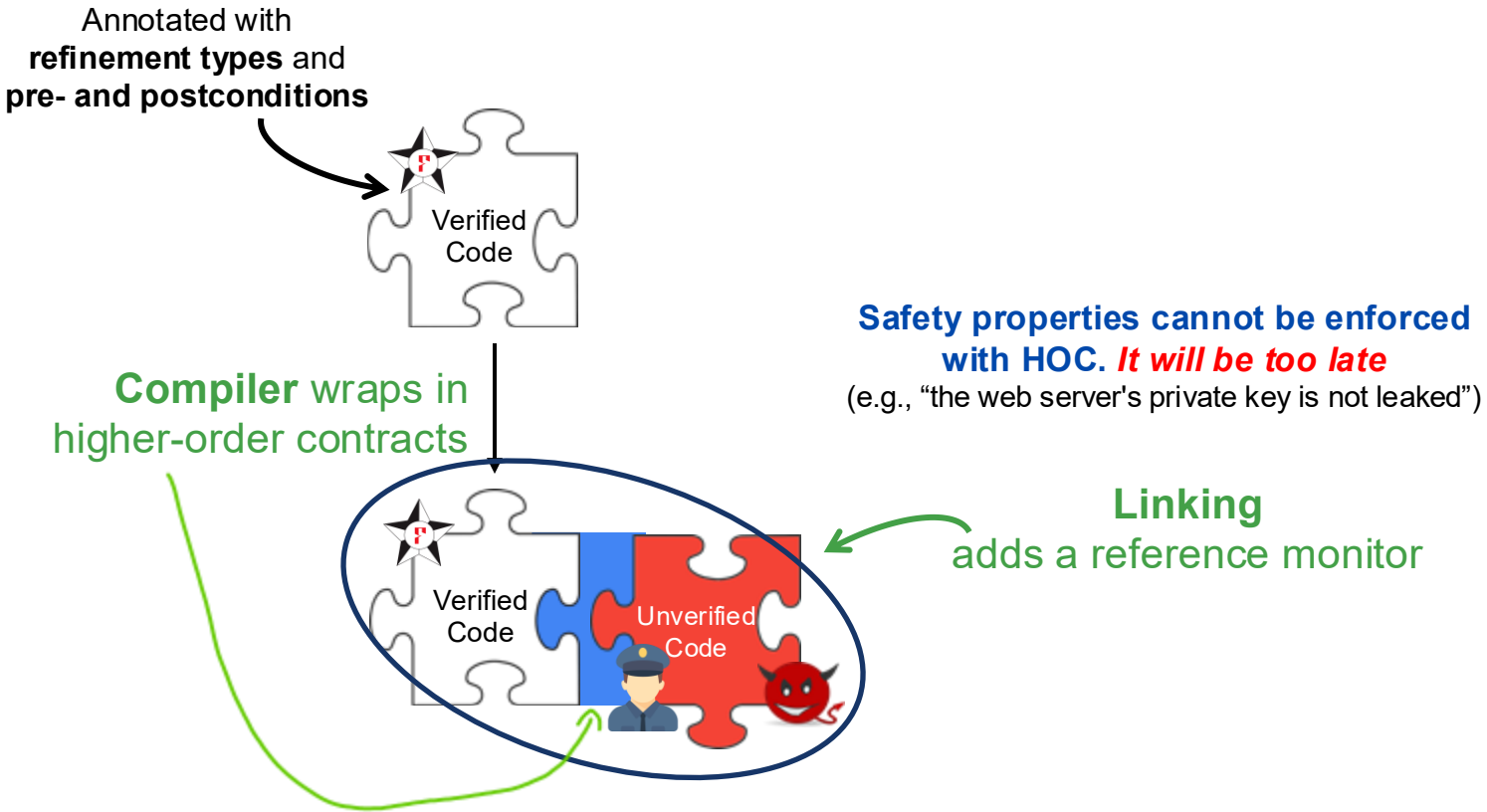
Secure compilation for verified stateful programs
(Andrici et al. ICFP'25)

Secure extraction of IO programs
(Andrici et al. 2026)

Verified secure compilation framework for verified IO programs (Andrici et al. POPL'24)



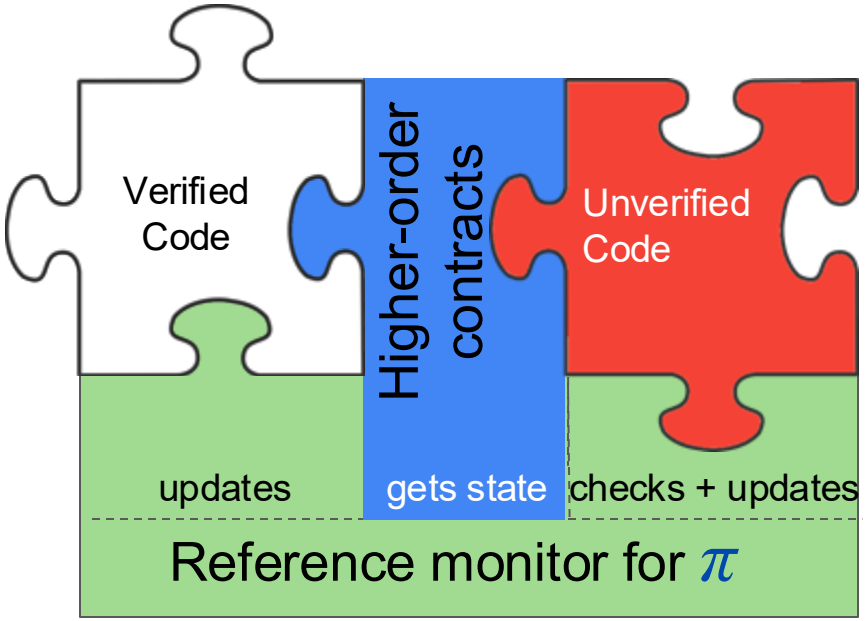
Verified secure compilation framework for verified IO programs (Andrici et al. POPL'24)



Safety properties cannot be enforced with HOC. *It will be too late*
(e.g., “the web server's private key is not leaked”)

Verified to soundly enforce specifications dynamically at the interface level

∀ unverified code



satisfies security property π

Verifying an IO program

```
val lib_parse : file_descr → MIO string
let lib_parse _ = open… "…/passwd"; ...
```

```
let verif_prog (lib_parse) ≑
 MIO unit
  (requires True)
  (ensures (λ lt → ∀ev ∈ lt. EvOpenfile? ev ⟹ ev.fnm == "data.csv"))=
    let fd = openfile "data.csv" in
    let msg = lib_parse fd in
    close fd
```

IO operation returns an error of the monitor

Local Trace:
[EvOpenfile "data.csv" fd; …… ; EvClose fd]

The specs on a strong interface are treated differently

pre-condition

e.g., file descriptor given as argument is open

enforced statically

```
lib_parse:
    fd:_ → MIO msg:string (requires (pre fd))
                          (ensures (post fd msg))
```

post-condition about the result

e.g., returned value was read from fd

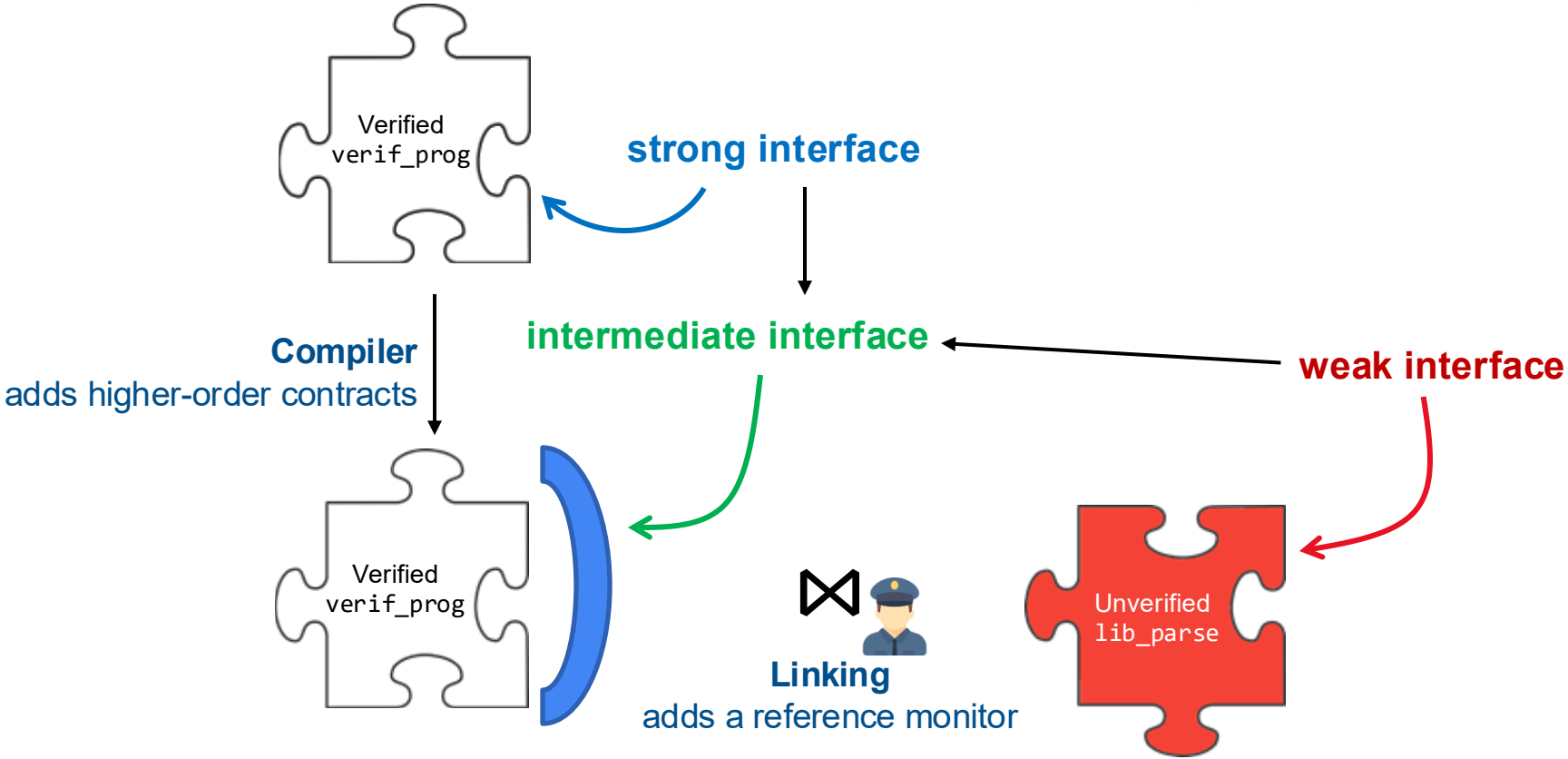
enforced by HOC

π= **post-condition about IO behavior**

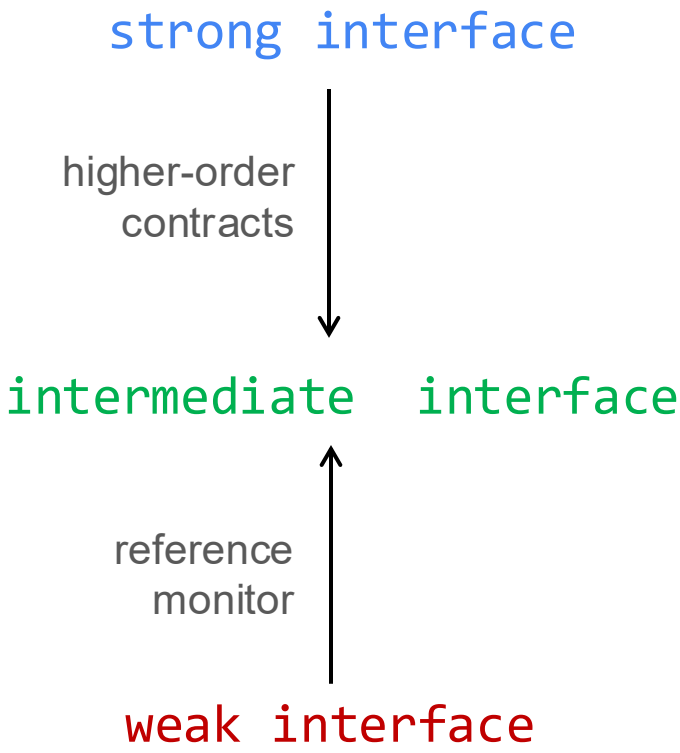
e.g., *does not open any files*

enforced by reference monitor

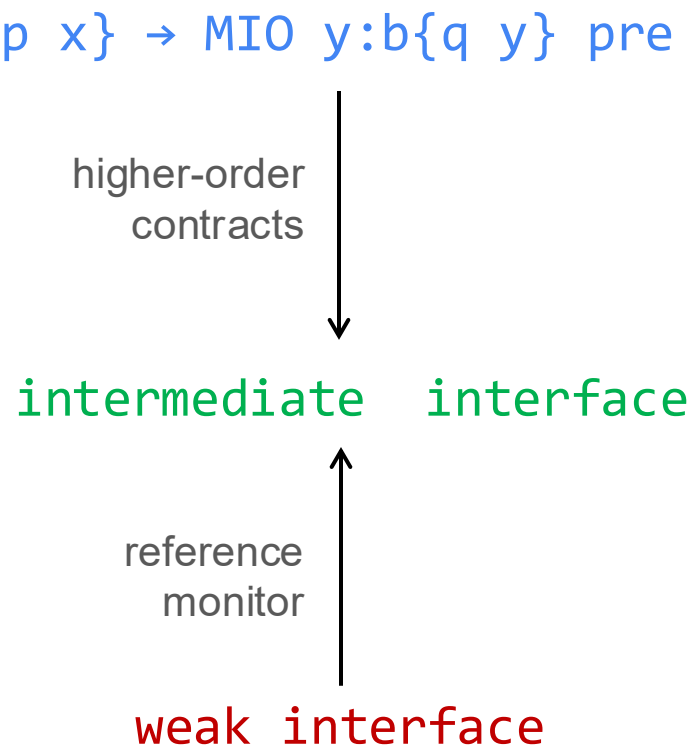
Enforcing the refinements and pre- and postconditions on the interface in two steps



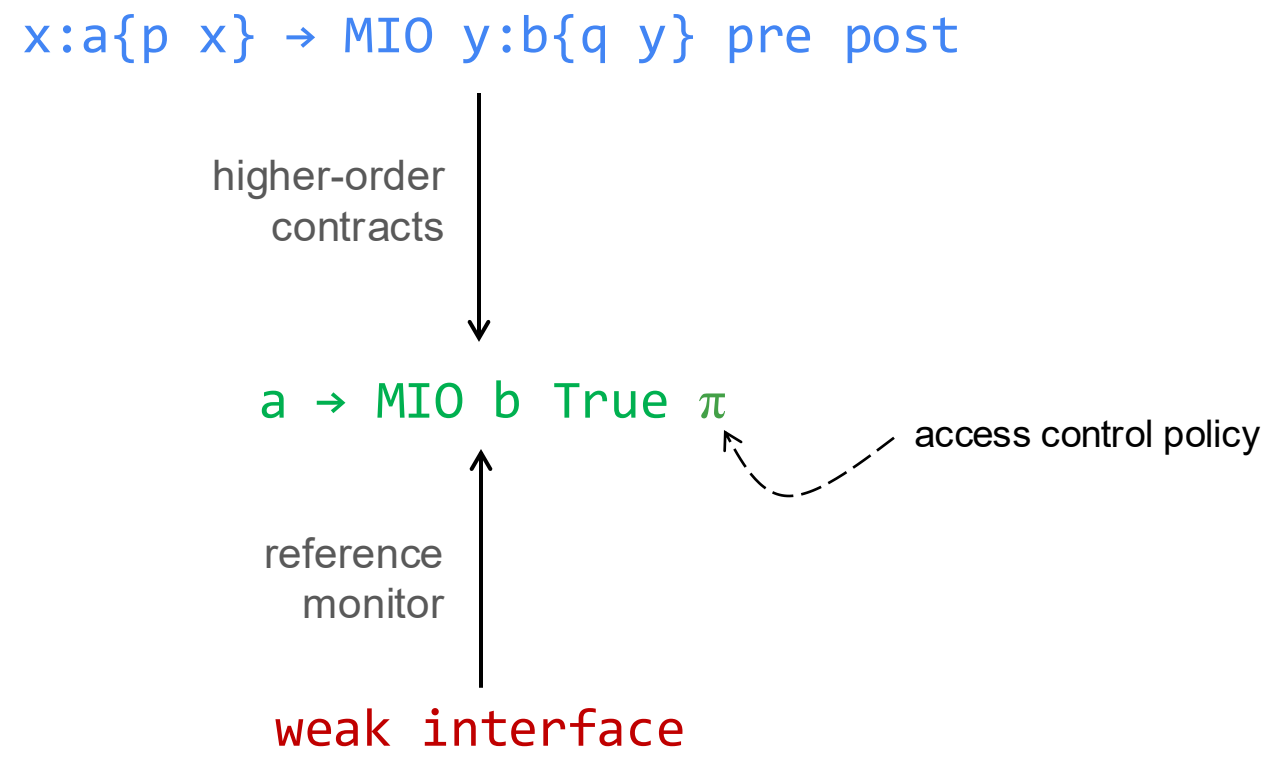
Translations between interfaces



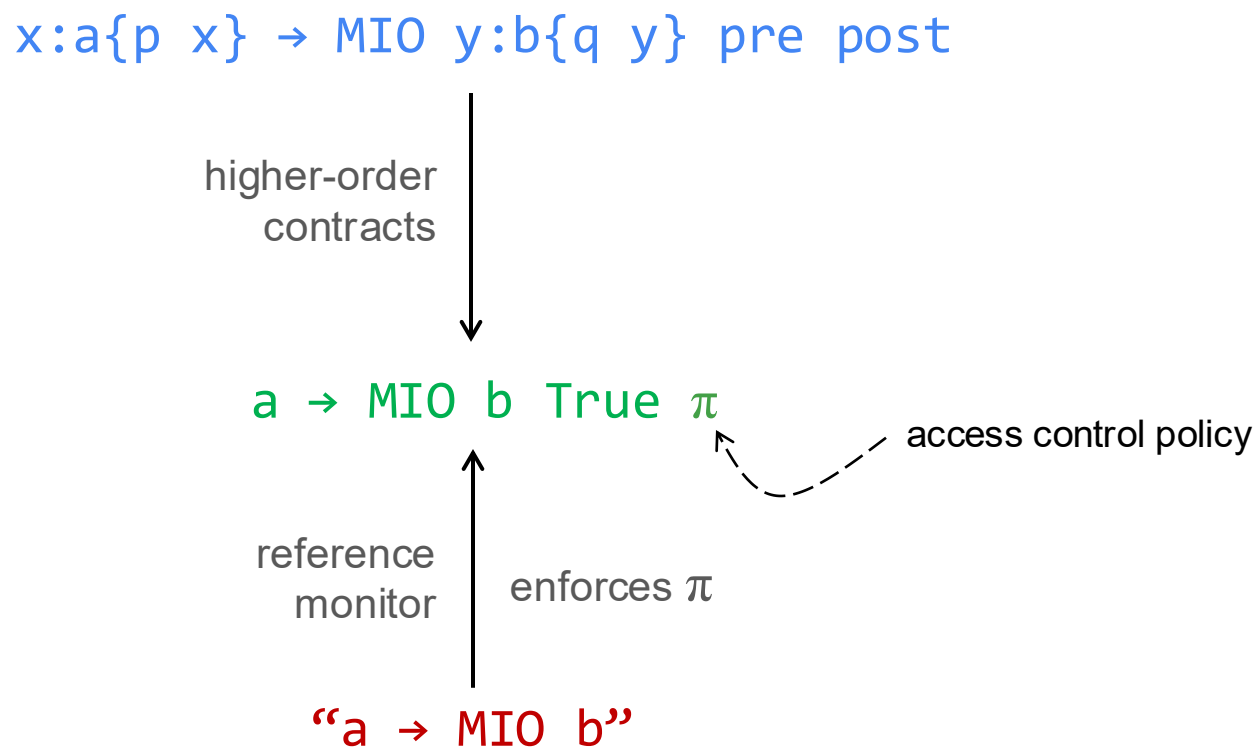
Translations between interfaces



Translations between interfaces



Translations between interfaces



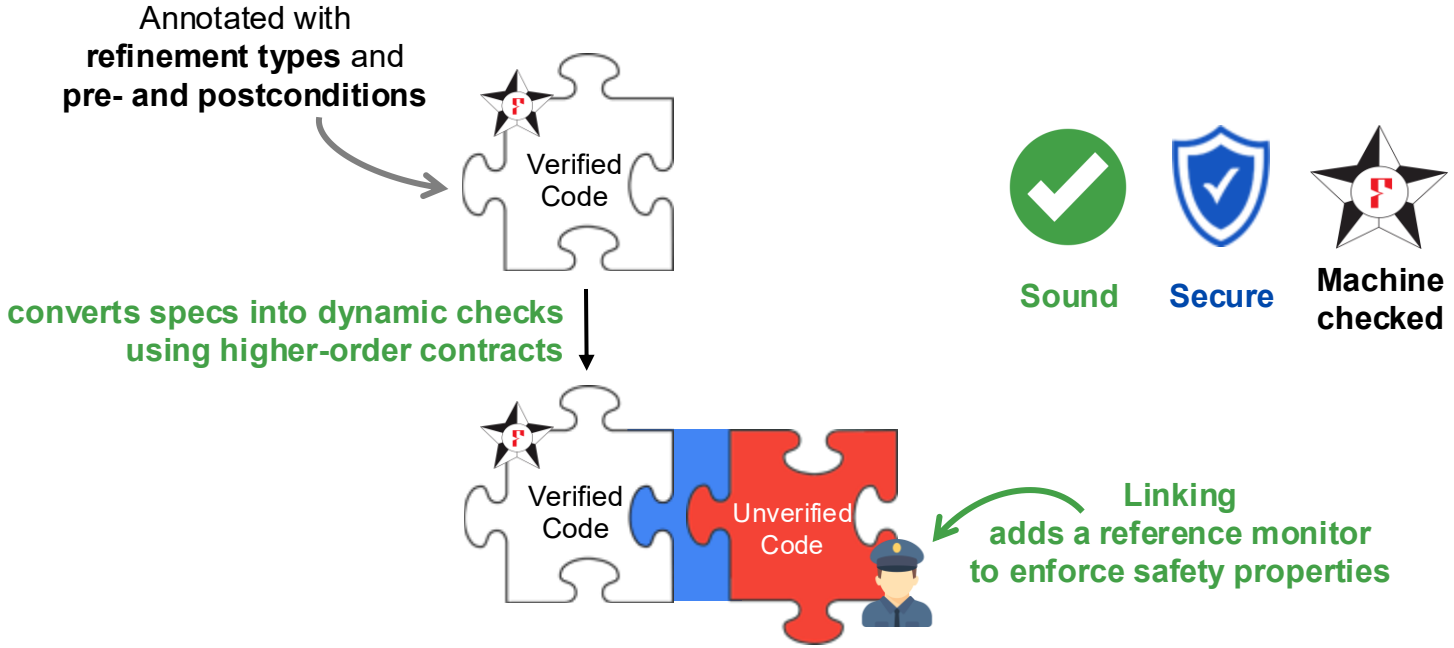
Proof of secure compilation

compiler satisfies Robust Relational Hyperproperty Preservation (RrHP)

Usually very hard to prove, but our proof is by construction:

- Shallow embeddings of the source and target language
- Specialized design of the higher-order contracts
- Follows from the following informal syntactic equality:
  - Compilation + Target Linking = Back-translation + Source Linking

Verified secure compilation framework for verified IO programs (Andrici et al. POPL'24)



Results towards secure extraction

Secure compilation for verified IO programs
(Andrici et al. POPL'24)

Secure compilation for verified stateful programs
(Andrici et al. ICFP'25)

Secure extraction of IO programs
(Andrici et al. 2026)

How to securely compile stateful programs?

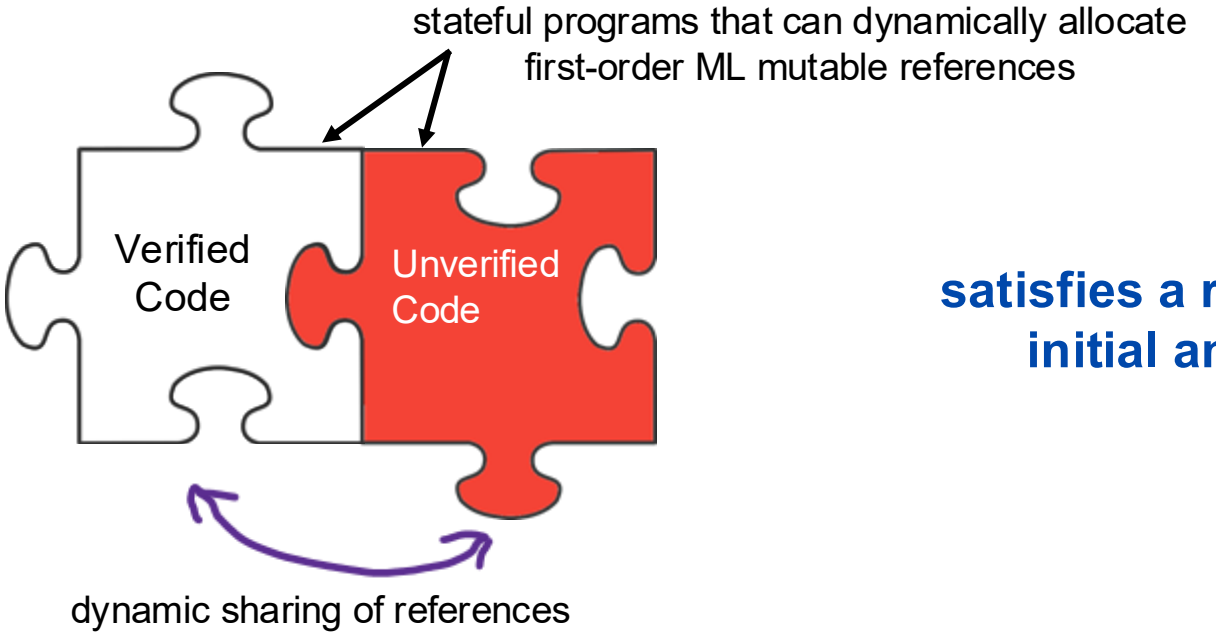
(Andrici et al. ICFP'25)

To do state, we have to do two things differently:

1. Instead of reasoning using traces,
   we want to reason in terms of **initial** and **final heaps**.

2. Dynamic checks are expensive. Can we avoid them for stateful programs?

Can we avoid monitoring state by keeping track of shared references?



satisfies a relation between initial and final state

It is tricky to track which references are shared

```
let prog (unverified_lib : ref (ref int) → unit → unit) =
  let secret : ref int = alloc 42 in
  let r : ref (ref int) = alloc (alloc 0) in
  let cb = unverified_lib r in
  r := alloc 1;
  cb (); // what references get modified here?
  assert (!secret == 42)
```

Heap

secret ↦ 42?

r' ↦ ?

r ↦ ?

r'' ↦ ?

Sharing is *transitive* and *permanent*

Looking at what references get directly passed is not enough

Overaproximating the shared references using labels

Labeling mechanism that is encoded in F⋆ and computationally irrelevant:

- Fresh references are labeled as `private`.
- Dynamic operation to label a reference as `shareable`.

Rules:

- Once `shareable`, forever `shareable`.
- `Shareable` points only to `shareable`.
- Only `shareable` references can be passed between verified-unverified code.

Tracking shared references using a labeling mechanism

Exta pre- and post-conditions:

- accepts and returns only shareable references
- **modifies only shareable references**

```
let prog (unverified_lib : ref (ref int) → unit → unit) =
  let secret : ref int = alloc 42 in
  let r : ref (ref int) = alloc (alloc 0) in
  label_shareable (!r); label_shareable r;
  let cb = unverified_lib r in
  let r'' = alloc 1 in label_shareable r''; r := r'';
  cb ();
  assert (!secret == 42)
```

Extra pre-condition:
If r is shareable,
then r'' has to be shareable.

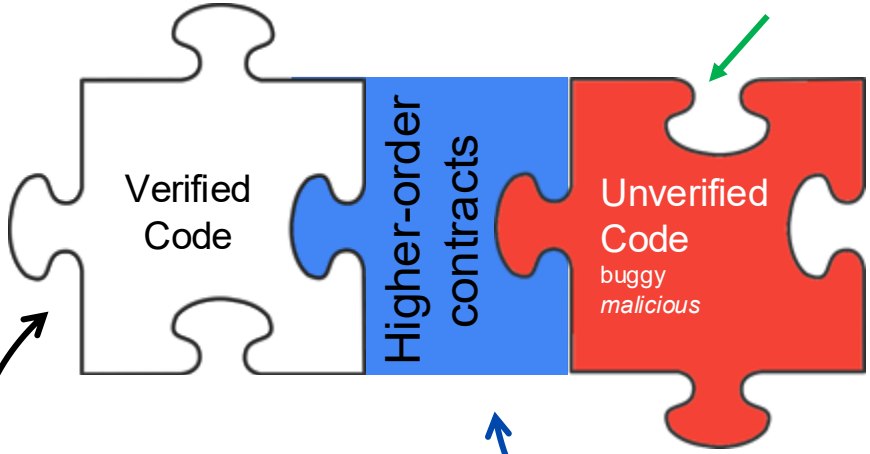
Heap

```
secret ↦ 42
r' ↦ ?
r ↦ ?
r'' ↦ ?

ghost lmap↦{
  secret=private,
  r'=shareable,
  r=shareable,
  r''=shareable,
}
```

Labeling mechanism to track which references are shared

∀ unverified code.



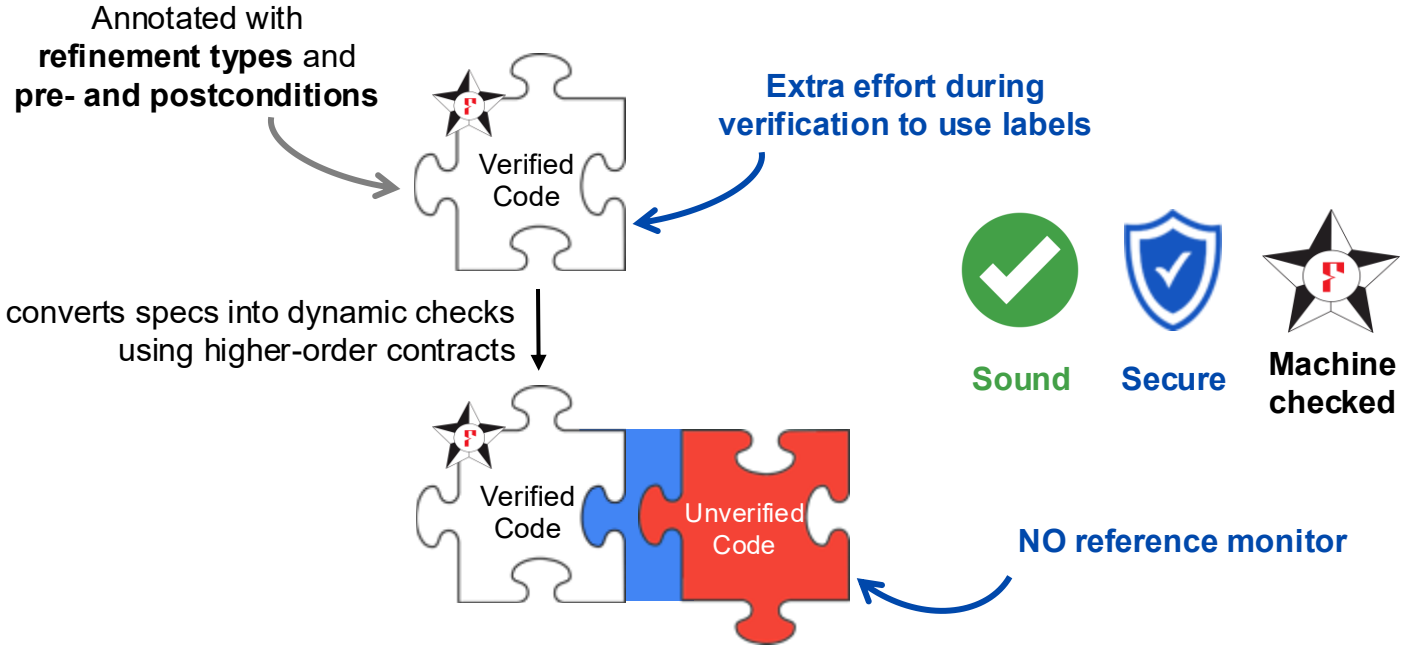
Statically proved that **modifies ONLY `shareable` references** is a property of unverified code

satisfies a relation between initial and final state

When doing proofs, **dynamically labels** references as **`shareable`**

HOC enforce assumptions about `shareable` *references*

A verified secure compilation framework for verified stateful programs (Andrici et al. ICFP'25)



Results towards secure extraction

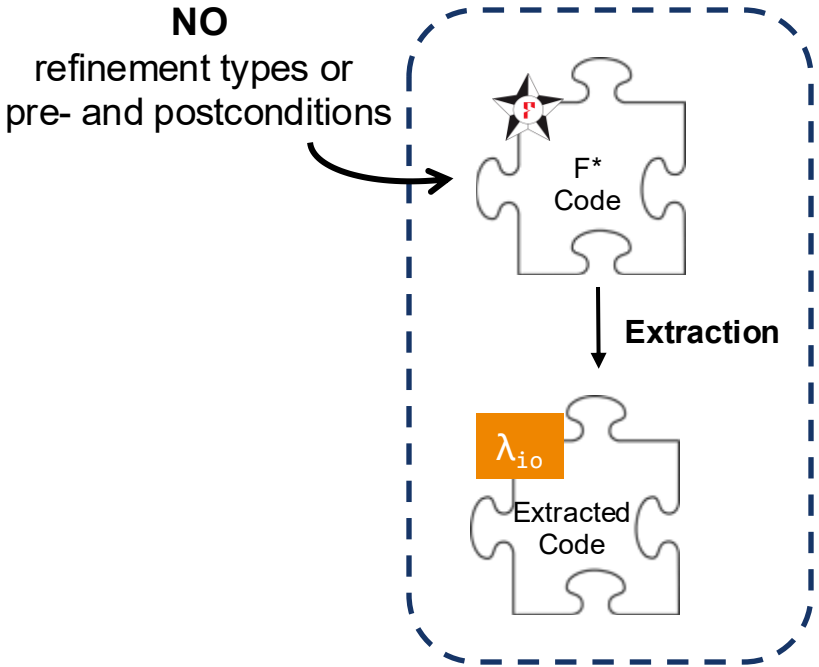
Secure compilation for verified IO programs
(Andrici et al. POPL'24)

Secure compilation for verified stateful programs
(Andrici et al. ICFP'25)

Secure extraction of IO programs
(Andrici et al. 2026)

Secure extraction of IO programs

(Andrici et al. 2026)



Challenge:
how to **define** and **verify** extraction in F*?

Verifying extraction is challenging

```
let prog lib : io unit =
 let msg = read () in
 let res = lib msg in
 write res
```

Shallow embedding

Extraction is a metaprogram

Verifying extraction is challenging:

involves **verifying quotation**—an open challenge in *all proof assistants*

```
let extracted_prog : exp =
 ELambda
  ELet ERead
  ELet (EApp (EVar 1) (EVar 0))
  EEClose (EVar 0)
```

Deep embedding

Existing models of extraction use translation validation

Shallow embedding

```
let prog lib : io unit =
 let msg = read () in
 let res = lib msg in
 write res
```

Metaprogram

Deep embedding

```
let extracted_prog : exp =
 ELambda
  ELet ERead
  ELet (EApp (EVar 1) (EVar 0))
  EWrite (EVar 0)
```

Proof of equivalence

```
let proof
  : Lemma (prog ≈ extracted_prog)
  = ...
```

Validation: `extracted_prog` **and** `proof` **have to be type checked in the proof assistant, which can fail**

Our extraction model minimizes the use of translation validation

Shallow embedding

```
let prog lib : io unit =
 let msg = read () in
 let res = lib msg in
 write res
```

Metaprogram generates a typing derivation

Typing derivation

```
let tyj_prog: typing empty _ prog =
  ...
```

Validation: F* type checks derivation

Syntax generation
(normal F* function)

Deep embedding

```
let extracted_prog : exp =
 ELambda
  ELet ERead
  ELet (EApp (EVar 1) (EVar 0))
  EWrite (EVar 0)
```

Guarantee of secure extraction

extraction guarantees Robust Relational Hyperproperty Preservation (RrHP)

Shallow embedding

```
let prog lib : io unit =
 let msg = read () in
 let res = lib msg in
 write res
```

Typing derivation

```
let tyj_prog: typing empty _ prog =
  ...
```

Syntax generation
(normal F* function)

Part verified once and for all

Relates the shallowly embedded program with the deeply embedded one

Proved using two cross language (asymmetric) logical relations

Deep embedding

```
let extracted_prog : exp =
 ELambda
  ELet ERead
  ELet (EApp (EVar 1) (EVar 0))
  EWrite (EVar 0)
```

Relational Quotation

A translation validation technique for quoting shallowly embedded programs.

We define a **typing relation** for the subset of F⋆ we want to extract:

```
type typing : Γ:typ_env → a:Type → (eval_env Γ → a) → Type =
| Qfalse  : Γ:typ_env → typing Γ bool (λ _ → false)
| QVar    : Γ:typ_env → v:nat{Some? (Γ v)} →
            typing Γ (Some?.v (Γ v)) (λ σ → index σ v)
| QLambda : Γ:typ_env → a:Type → b:Type →
            body:(eval_env (extend a Γ) → b) →
            typing (extend a Γ) b body →
            typing Γ (a → b) (λ σ x → body (x::σ))
```

To support the **io** monad, we follow the typing rules of *fine-grain call-by-value* (P.B. Levy et al. 2003)

Typing derivation generated using program's structure

Shallow embedding

```
let prog lib : io unit =
  let msg = read () in
  let res = lib msg in
  write res
```

Same structure

Typing derivation

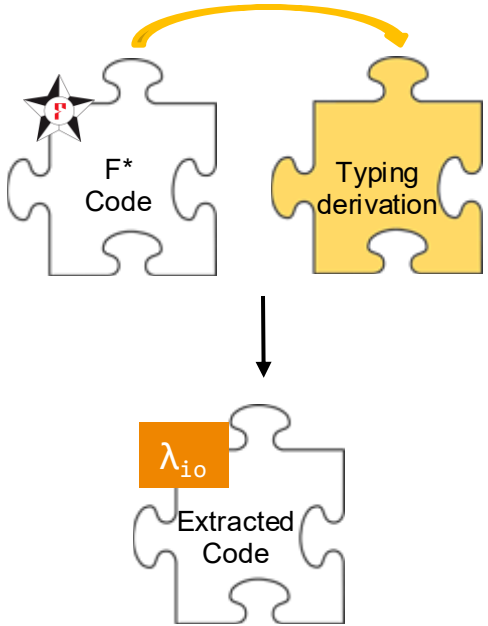
```
let tyj_prog : typing empty ((string → io string) → io unit) prog =
  QLambda
    QLet QRead
    QLet (QApp (QVar 1) (QVar 0))
    QWrite (QVar 0)
```

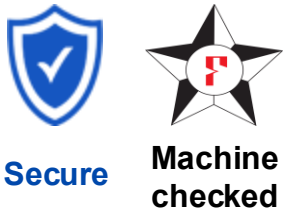
Only the constructors are needed, the rest is inferred by F*!

Secure extraction of IO programs

(Andrici et al. 2026)

NO
refinement types or
pre- and postconditions





Secure

Machine checked

Secure extraction of verified effectful F⋆ programs to ML

In this talk:

Secure compilation for verified IO programs

Secure compilation for verified stateful programs

Secure extraction of IO programs

Future work:

Combine the three projects into one: some engineering challenges.

Doing more effects: concurrent IO, higher-order store.

Target existing verified compilers for ML: CakeML via Lambda Box (Peregrine project)