ICFP 2025 Singapore

SecRef★: Securely Sharing Mutable References between Verified and Unverified Code in F★

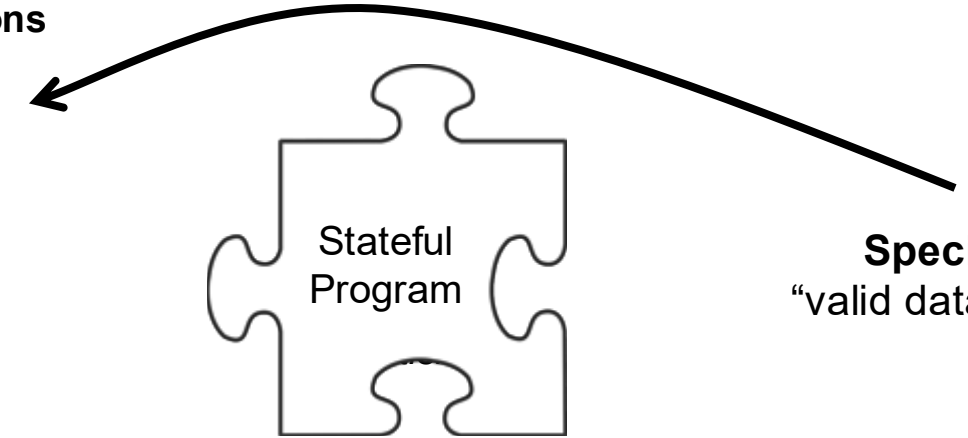
Cezar-Constantin **Andrici**, Danel **Ahman ,** Cătălin **Hriţcu,**

Ruxandra **Icleanu**, Guido **Martínez,** Exequiel **Rivas,** Théo **Winterhalter**

MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY · UNIVERSITY OF TARTU · Microsoft · THE UNIVERSITY of EDINBURGH · TAL TECH · Inria

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

We annotate the code with
refinement types and
pre- and post-conditions



Specification
“valid data structures”

The F⋆ type checker verifies
if the code satisfies the annotations.

Verification in F★ scales to realistic applications

EverCrypt

cryptographic provider

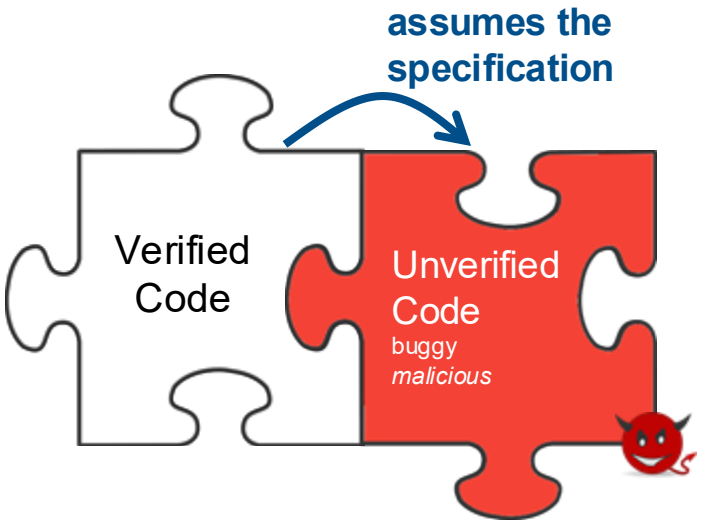
part of Mozilla Firefox, the Linux kernel, the Wireguard VPN.

EverParse

framework for secure parsers

part of Windows Hyper-V

Mixing verified code with unverified code can be problematic



$\models$

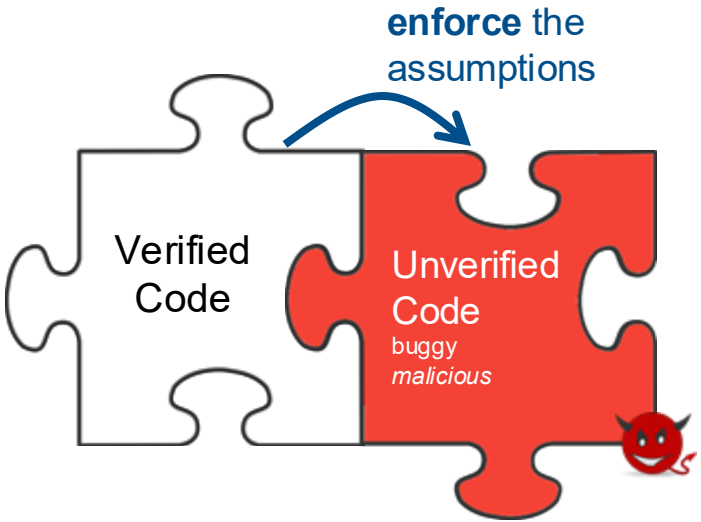
satisfies

Specification
"valid data structure"

No guarantee that the assumptions are satisfied

Unsound & Insecure

Solution: enforce the assumptions *dynamically* and *statically*

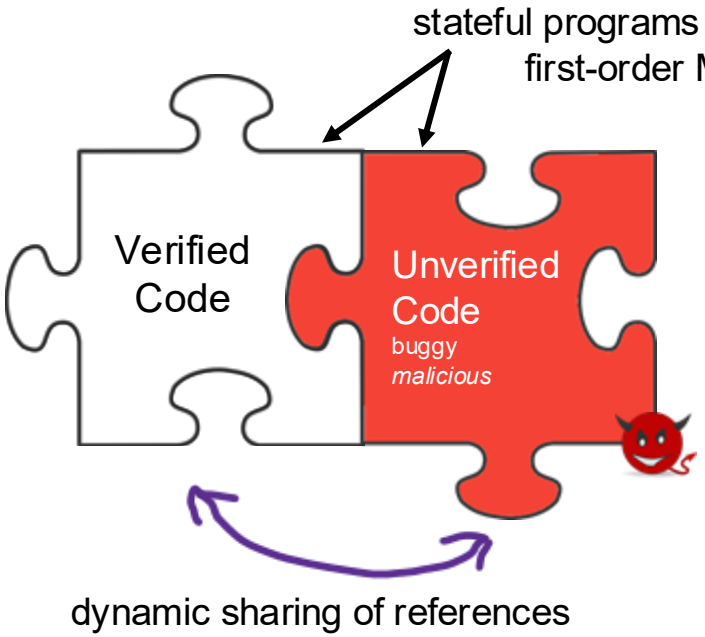


$\models$

satisfies

Global specification
relation between initial and final state

Solution: enforce the assumptions *dynamically* and *statically*

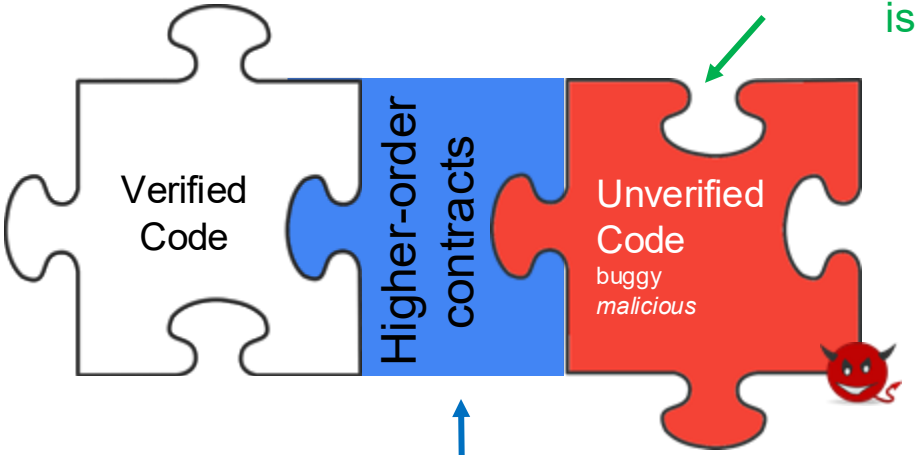


$\models$

satisfies

Global specification
relation between initial and final state

Solution: enforce the assumptions *dynamically* and *statically*



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

$\models$

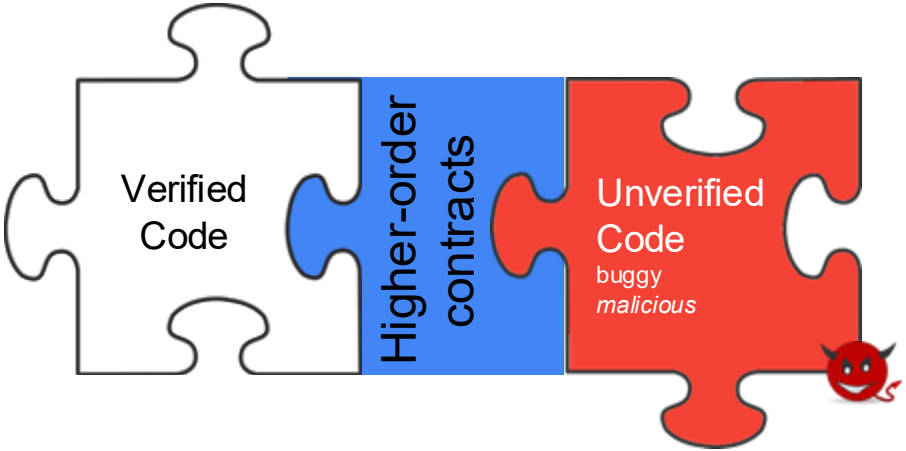
satisfies

Global specification
relation between initial and final state

dynamically enforce assumptions about *shared references*

Solution: enforce the assumptions *dynamically* and *statically*

∀ unverified code

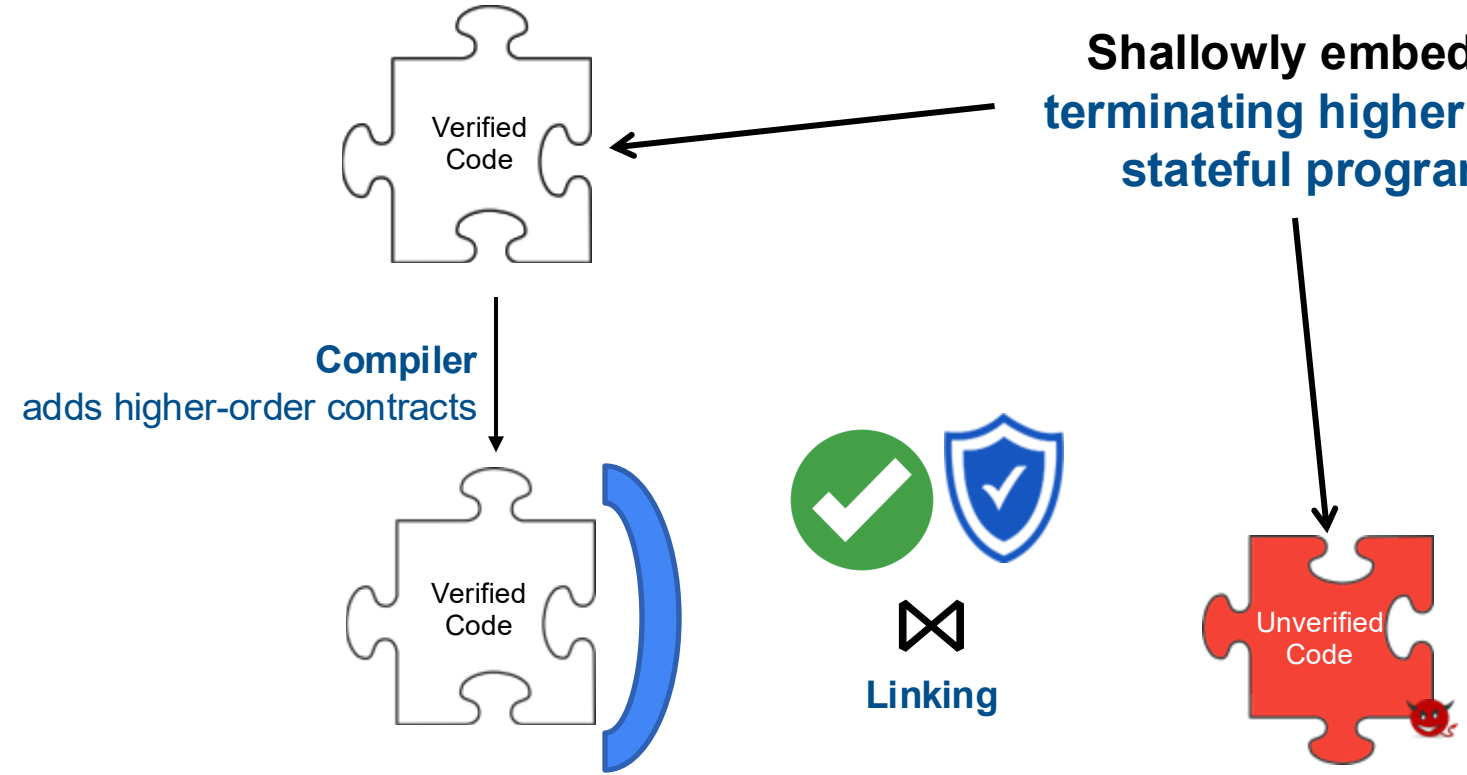


⊨

satisfies

Global specification
relation between initial and final state

SecRef★: a verified secure compilation framework for the sound verification of stateful partial programs



Contributions

Sound verification of partial programs

SecRef★ is verified

Verified compilation and linking. Proof of soundness.

SecRef★ is secure

Mechanized proof of Robust Relational Hyperproperty Preservation.

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

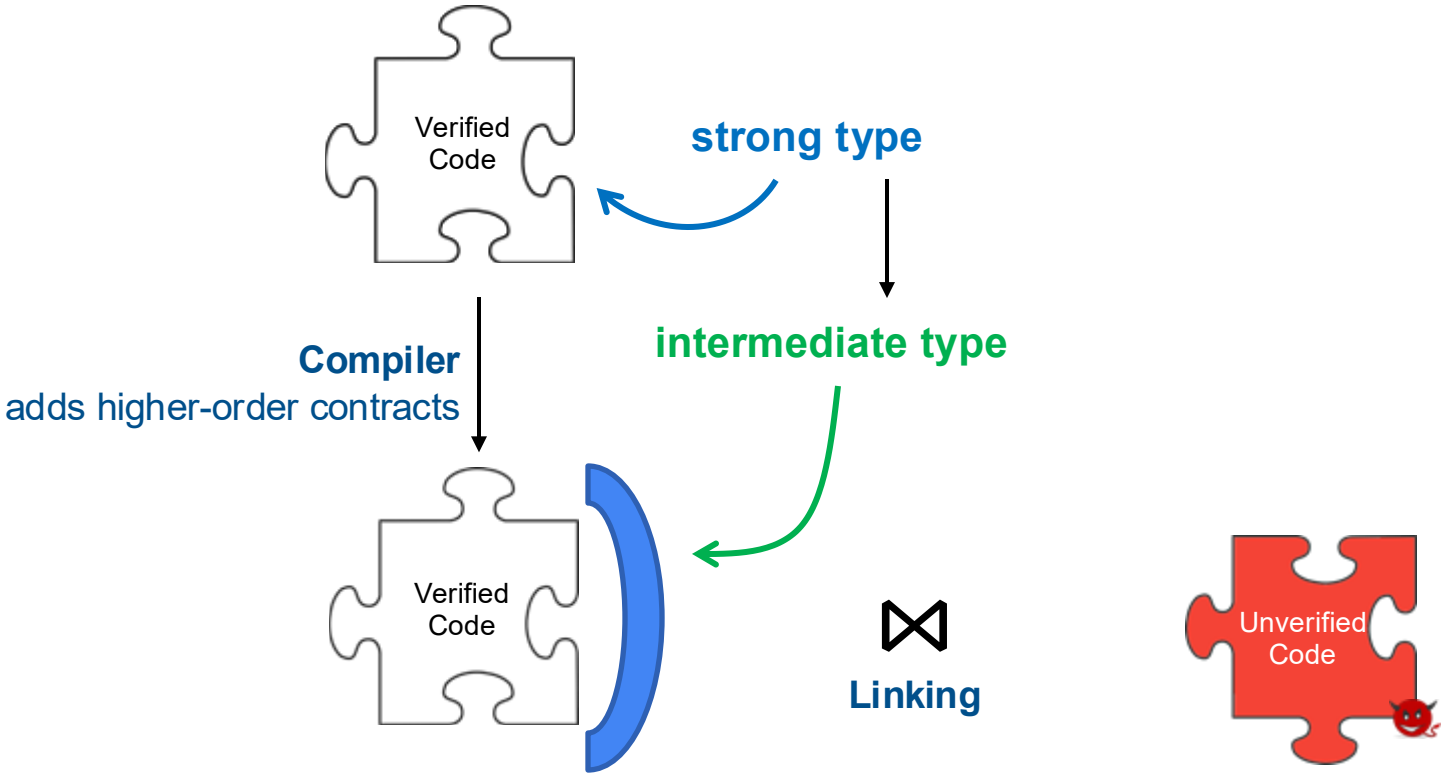
```
secret ↦ 42
r' ↦ ?
r ↦ ?
r'' ↦ ?

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

The verified code assumes a **strong type**

containing refinements and pre-post conditions

```
unverified_lib :
  r:ref (ref int) → LR (...)
    (requires (λ h₀ → is_shareable r ∧
                      is_even (sel (sel r h₀)) h₀))
    (ensures (λ h₀ x h₁ → modifies_only_shareable h₀ h₁ ∧
                          is_shareable x ∧
                          sel r h₀ == sel r h₁))
```

The types contain the assumptions



Intermediate type

refinements and pre-post conditions convert to dynamic checks

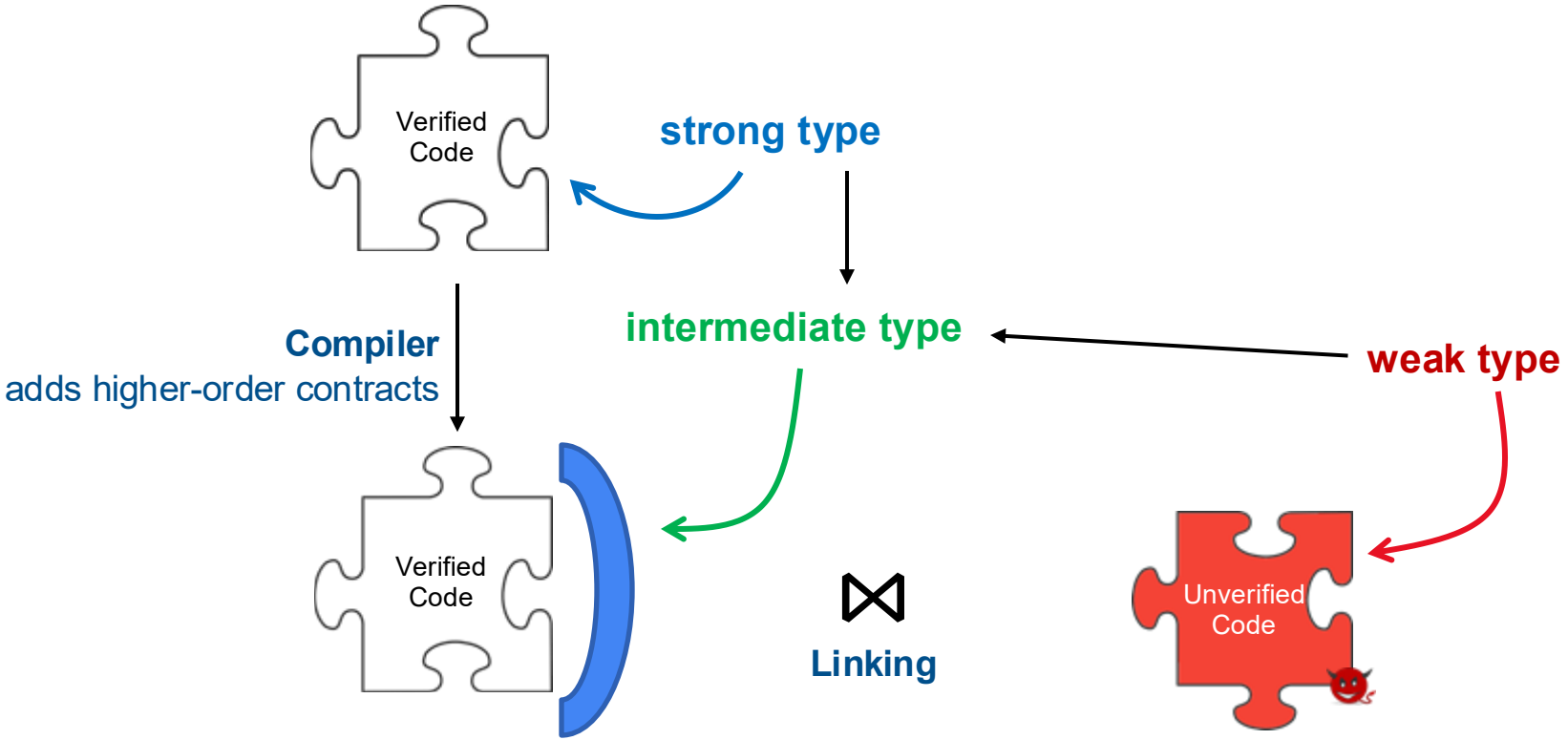
```
unverified_lib :
  r:ref (ref int) → LR (...)
    (requires (λ h₀ → is_shareable r ∧
                      is_even (sel (sel r h₀)) h₀))
    (ensures (λ h₀ x h₁ → modifies_only_shareable h₀ h₁ ∧
                          is_shareable x ∧
                          sel r h₀ == sel r h₁))
```

Intermediate type

refinements and pre-post conditions convert to dynamic checks

```
unverified_lib :
  r:ref (ref int) → LR (...)
    (requires (λ h₀ → is_shareable r))
    (ensures (λ h₀ x h₁ → modifies_only_shareable h₀ h₁ ∧
                          is_shareable x))
```

The types contain the assumptions



The unverified code has a **weak type**

no concrete refinements and pre-post conditions

```
unverified_lib :
  r:ref (ref int) → LR (...)
    (requires (λ h₀ → is_shareable r))
    (ensures (λ h₀ x h₁ → modifies_only_shareable h₀ h₁ ∧
                          is_shareable x))
```

The unverified code has a **weak type**

no concrete refinements and pre-post conditions

```
unverified_lib :
  r:ref (ref int) → LR (...)
    (requires (λ h₀ → φ r))
    (ensures (λ h₀ x h₁ → modifies_only_shareable h₀ h₁ ∧
                          φ x))
```

The unverified code has a **weak type**

no concrete refinements and pre-post conditions

```
unverified_lib :
  r:ref (ref int) → LR (...)
    (requires (λ h₀ → φ r))
    (ensures (λ h₀ x h₁ → h₀ ≼ h₁ ∧
                          φ x))
```

The unverified code has a **weak type**

no concrete refinements and pre-post conditions

By instantiating with the previous predicates, we get that unverified code modifies only `shareable` references.

```
unverified_lib : φ:_ → ≼:_ → ... →
  r:ref (ref int) → LR (...)
    (requires (λ h₀ → φ r))
    (ensures (λ h₀ x h₁ → h₀ ≼ h₁ ∧
                          φ x))
```

Full representation of unverified code and why it is appropriate in the paper.

Contributions

Sound verification of partial programs

SecRef★ is verified

Verified compilation and linking. Proof of soundness.

SecRef★ is secure

Mechanized proof of Robust Relational Hyperproperty Preservation.

Robust Relational Hyperproperty Preservation (RrHP)

- Strongest criterion of Abate et al. (CSF'19). **Stronger than full abstraction.**
- **Compilation preserves**:
  - Observational equivalence
  - Noninterference
  - Trace properties
- 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

Contributions

Sound verification of partial programs

SecRef★ is verified

Verified compilation and linking.

SecRef★ is secure

Mechanized proof of Robust Relational Hyperproperty Preservation.

More in the paper

The shallow embedding: a Dijkstra Monad!

Monadic representation for Monotonic State + proof of soundness.
Labeling mechanism encoded in Monotonic State.
More labels: `encapsulated` references.

Proofs about SecRef★

Proofs for both cases of who has initial control.
Syntactic representation of unverified code.

Case study of a simple cooperative multi-threading scheduler

Written, verified, compiled and secured against unverified threads using SecRef★.

Cezar Andrici, MPI-SP: cezar.andrici@mpi-sp.org