

Misquoted No More: Securely Extracting F[★] programs with IO

Cezar-Constantin Andrici, Abigail Pribisova, Danel Ahman,

Cătălin Hrițcu, Exequiel Rivas, Théo Winterhalter



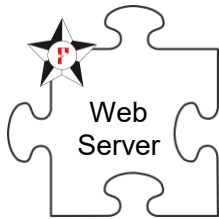
MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY



UNIVERSITY OF TARTU



Proof-oriented language $F^\star$ offers strong guarantees



satisfies security property π

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



Extract the verified code to run it



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

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

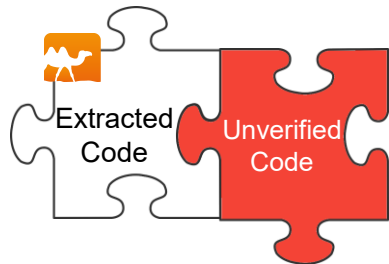


satisfies security property π

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



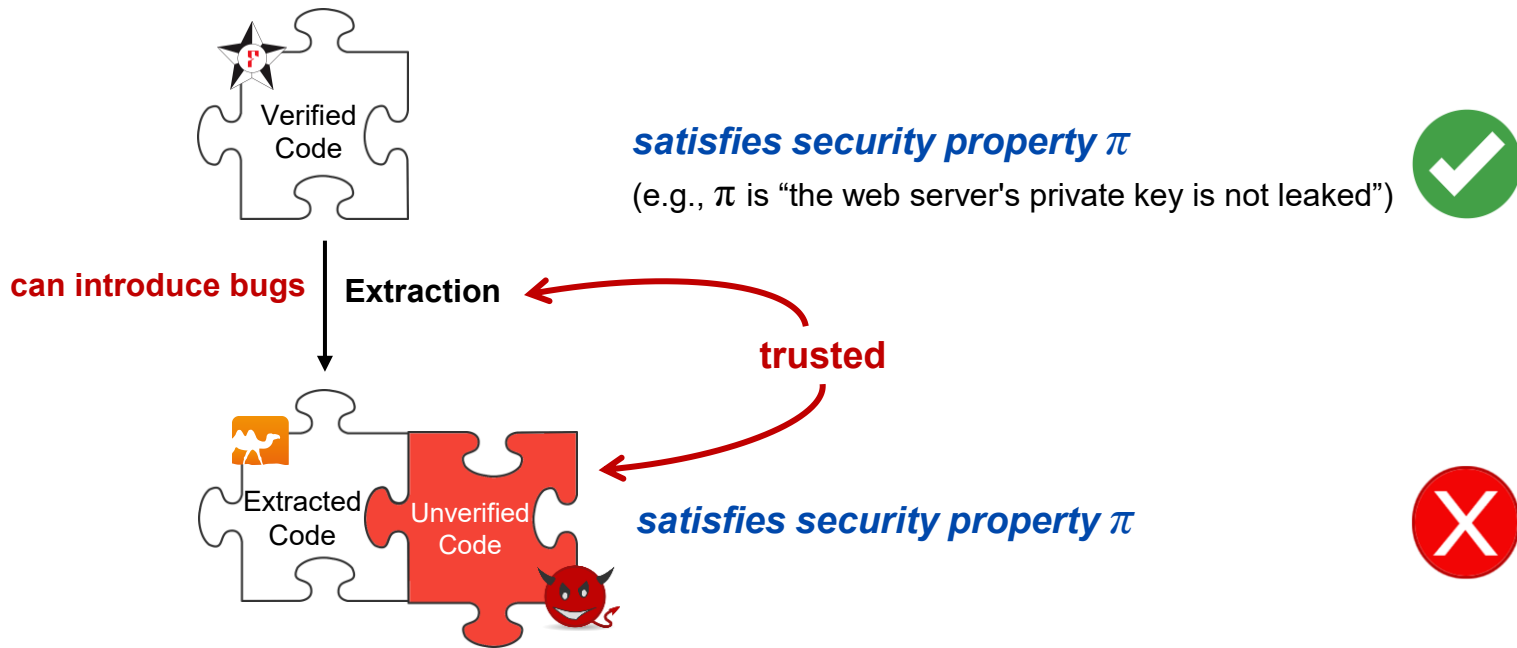
Extraction



satisfies security property π



Mixing verified code with unverified code can be **unsound and insecure**





We want extraction to satisfy
compiler correctness



Extraction



satisfies property π

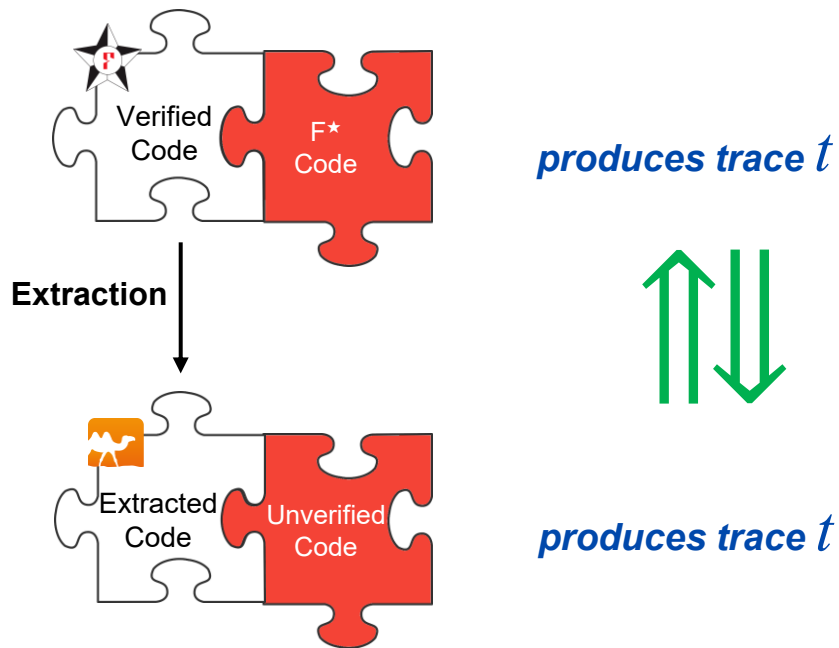


satisfies property π



We want extraction to satisfy
a secure compilation criterion

$\forall$ unverified code $\exists$ F^* code $\forall$ 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

A extraction framework **with formal secure compilation guarantees** for $F^\star$ programs with IO

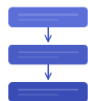


Compiler correctness



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 $F^\star$

Verifying extraction is challenging

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

Shallow embedding
(using the `io` monad)

Trace-producing semantics:

[`EvRead` msg; ; `EvWrite` res]

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

Deep embedding
(lambda calculus with IO)

Extraction is a metaprogram

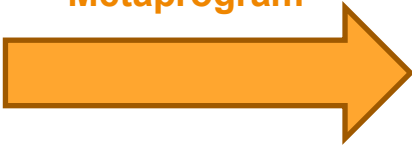
Verifying extraction is challenging:
involves **verifying quotation**—an open
challenge in *all proof assistants*

Related work uses **translation validation** to obtain formal correctness guarantees

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

We introduce **Relational Quotation**

a new translation validation technique
for quoting shallowly embedded programs using typing relations

Shallow embedding

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

**simple
metaprogram**



Typing derivation

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

Validation: F[★] type checks the derivation,
checking it corresponds to prog

The Typing Relation is indexed by **F[★] types** and **values**

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

```
type typing :  $\Gamma$ :typ_env  $\rightarrow$  a:Type  $\rightarrow$  (eval_env  $\Gamma \rightarrow$  a)  $\rightarrow$  Type =  
| Qfalse      :  $\Gamma$ :typ_env  $\rightarrow$  typing  $\Gamma$  bool ( $\lambda$  _  $\rightarrow$  false)  
| QVar        :  $\Gamma$ :typ_env  $\rightarrow$  v:nat{Some? ( $\Gamma$  v)}  $\rightarrow$   
                typing  $\Gamma$  (Some?.v ( $\Gamma$  v)) ( $\lambda$   $\sigma \rightarrow$  index  $\sigma$  v)  
| QLambda     :  $\Gamma$ :typ_env  $\rightarrow$  a:Type  $\rightarrow$  b:Type  $\rightarrow$   
                body:(eval_env (extend a  $\Gamma$ )  $\rightarrow$  b)  $\rightarrow$   
                typing (extend a  $\Gamma$ ) b body  $\rightarrow$   
                typing  $\Gamma$  (a  $\rightarrow$  b) ( $\lambda$   $\sigma$  x  $\rightarrow$  body (x:: $\sigma$ ))
```

This construction is based on *Deeper Shallow Embeddings* (J. Prinz et al., ITP 2022)

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

Dependent Types enable F[★] to infer most of the derivation

Writing a **derivation**, we only give the **typing** derivations of the sub-terms

```
type typing :  $\Gamma$ :typ_env  $\rightarrow$  a:Type  $\rightarrow$  (eval_env  $\Gamma \rightarrow$  a)  $\rightarrow$  Type =  
| Qfalse   :  $\Gamma$ :typ_env  $\rightarrow$  typing  $\Gamma$  bool ( $\lambda$  _  $\rightarrow$  false)  
| QVar     :  $\Gamma$ :typ_env  $\rightarrow$  v:nat{Some? ( $\Gamma$  v)}  $\rightarrow$   
             typing  $\Gamma$  (Some?.v ( $\Gamma$  v)) ( $\lambda$   $\sigma \rightarrow$  index  $\sigma$  v)  
| QLambda  :  $\Gamma$ :typ_env  $\rightarrow$  a:Type  $\rightarrow$  b:Type  $\rightarrow$   
             body:(eval_env (extend a  $\Gamma$ )  $\rightarrow$  b)  $\rightarrow$   
             typing (extend a  $\Gamma$ ) b body  $\rightarrow$   
             typing  $\Gamma$  (a  $\rightarrow$  b) ( $\lambda$   $\sigma$  x  $\rightarrow$  body (x:: $\sigma$ ))
```

F[★] infers **environments**, **types** and the **shallow terms** — simplifying the metaprogram.

Metaprogram simply follows the program's structure

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 ((string → io string) → io unit) prog =  
  QLambda  
    QLet QRead  
    QLet (QApp (QVar 1) (QVar 0))  
    QWrite (QVar 0)
```

These are inferred, but when spelled out, they are *unified*, obtaining the **syntax match**

Only the constructors are needed, the rest is inferred by F★

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
```

Typing derivation

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

Syntax generation
(normal F[★] function)
verified once and for all

Deep embedding

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

The two are related using
logical relations that give us:



Compiler correctness



Secure extraction

Misquoted No More: Securely Extracting F[★] programs with IO

More in the paper:

Support for refinement types!



Compiler
correctness



Secure
RrHP



Machine
checked

Future work:

Doing more effects: divergence, exceptions, mutable state (Abrahamsson et al. 2020, Pit-Claudel et al. 2022)

Target existing verified compilers for ML: CakeML via $\lambda\Box$ (Peregrine project)