IS&C

UNIVERSITÄT DES SAARLANDES

Union and Intersection Types for Secure Protocol Implementations

Cătălin Hrițcu

Saarland University, Saarbrücken

Joint work with: Michael Backes and Matteo Maffei

A little bit of background

Analyzing protocol implementations

- Recently several approaches proposed
  - **static analysis:** CSur [Goubault-Larrecq and Parrennes, VMCAI'05]
  - **software model checking:** ASPIER [Chaki & Datta, CSF '09]
  - **extracting ProVerif models:** fs2pv [Bhargavan, Fournet, Gordon & Tse, CSF '06]
    - for TLS [Bhargavan, Corin, Fournet & Zalinescu CCS '08]
  - **typing:** F7 [Bengtson, Bhargavan, Fournet, Gordon & Maffeis, CSF '08]
    - advantages: modularity, scalability, infinite # of sessions, predictable termination behavior, early feedback

F7v1 type-checker

[Bengtson, Bhargavan, Fournet, Gordon & Maffeis CSF '08]

- Security type-checker for (fragment of) F# (ML)
- Checks compliance with authorization policy
  - FOL used as authorization logic
  - proof obligations discharged using SMT solver (Z3)
- Dual implementation of cryptographic library
  - symbolic (DY model): used for security verification, debugging
  - concrete (real crypto): used in actual deployment
- F# fragment encoded into expressive core calculus (RCF)

RCF (Refined Concurrent PCF)

- λ-calculus + concurrency & channel communication in the style of asynchronous π-calculus
  (new c) c!m | c? → (new c) m
- Minimal core calculus
  - as few primitives as possible, everything else encoded
    e.g. ML references encoded using channels
- Expressive type system
  - refinement types $\quad$ Pos = {x : Nat | x ≠ 0}
  - dependent pair and function types (pre&post-conditions)
    λx.x : (y:Nat → {z:Nat | z = y})
    pred : x:Pos → {y:Nat | x = fold (inl y)}
  - iso-recursive and disjoint union types $\quad$ Nat = μα.α+unit

Security properties (informal)

- **Safety:** in all executions all asserts succeed
  (i.e. asserts are logically entailed by the active assumes)
- **Robust safety:**
  safety in the presence of arbitrary DY attacker

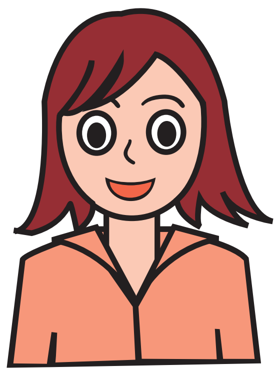
  - attacker is a closed assert-free RCF expression
  - attacker is Un-typed
    - type T is public if T <: Un
    - type T is tainted if Un <: T
- Type system ensures that well-typed programs are robustly safe

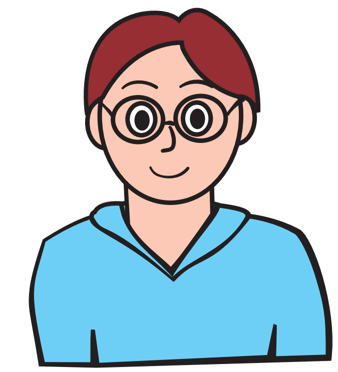
IS&C

UNIVERSITÄT DES SAARLANDES

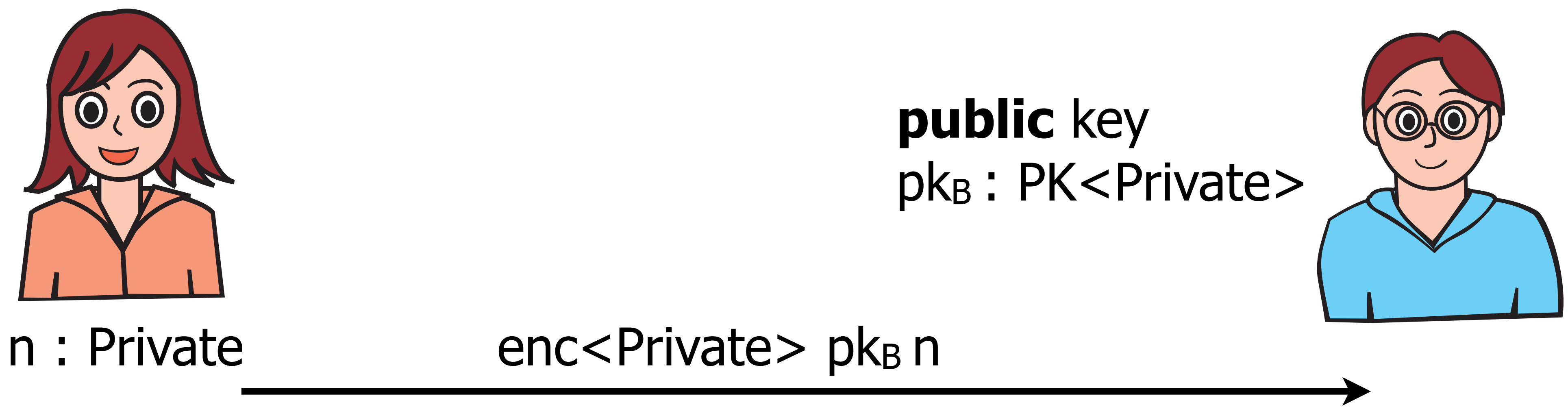
Why wasn't this enough?

An extremely simple example

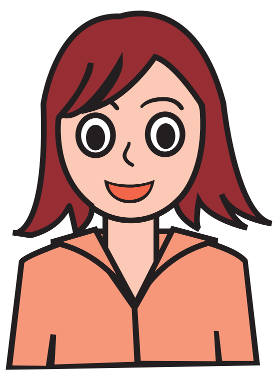




An extremely simple example

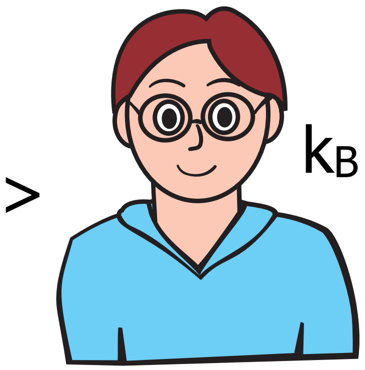


An extremely simple example



n : Private

public key
pk_B : PK<Private>

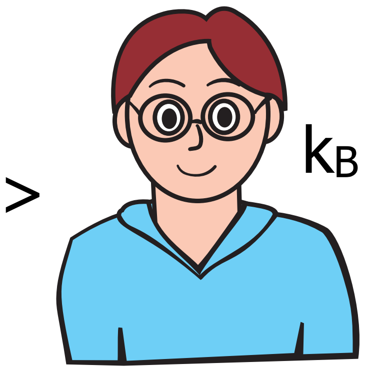


k_B : DK<Private>

enc<Private> pk_B n

let x_n = dec<Private> k_B net? **in**

An extremely simple example



public key
pk_B : PK<Private>

k_B : DK<Private>

n : Private

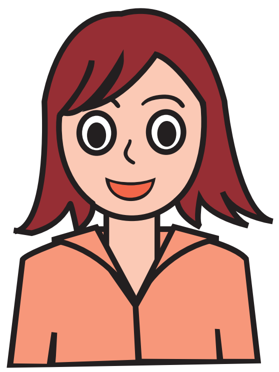
enc<Private> pk_B n

let x_n = dec<Private> k_B net? **in**

enc<Un> pk_B junk

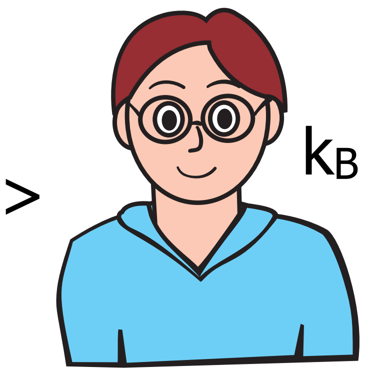
junk : Un

An extremely simple example



n : Private

public key
pk_B : PK<Private>



k_B : DK<Private>

enc<Private> pk_B n

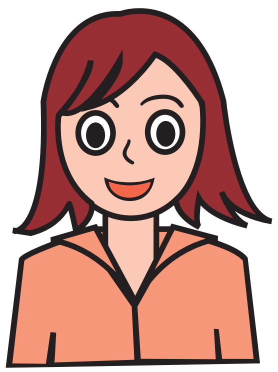
let x_n = dec<Private> k_B net? **in**

x_n : Private $\vee$ Un

enc<Un> pk_B junk

junk : Un

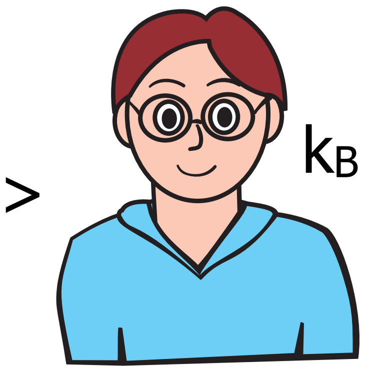
An extremely simple example



n : Private

enc<Private> pk_B n

public key
pk_B : PK<Private>



k_B : DK<Private>

let x_n = dec<Private> k_B net? **in**
assume Auth(m,B,A)

x_n : Private ∨ Un

enc<Un> pk_B junk

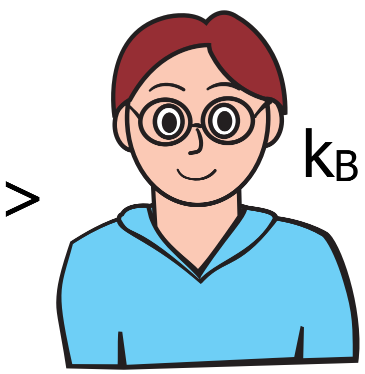
junk : Un

An extremely simple example

$pk_A : PK\langle T_A\rangle$

public key

$pk_B : PK\langle Private\rangle$



$k_B : DK\langle Private\rangle$

$n : Private$

$enc\langle Private\rangle\ pk_B\ n$ →

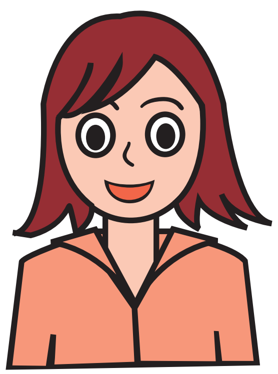
let $x_n = dec\langle Private\rangle\ k_B$ net? **in**

assume Auth(m,B,A)

← $enc\langle T_A\rangle\ pk_A\ (x_n,m)$

$x_n : Private \vee Un$

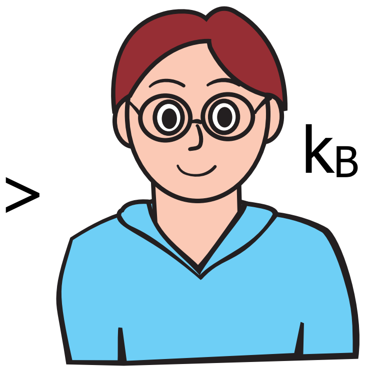
An extremely simple example



$pk_A : PK<T_A>$

public key

$pk_B : PK<Private>$



$k_B : DK<Private>$

$n : Private$

$enc<Private>\ pk_B\ n$

$T_A = Private \vee Un * \{y_m : Un \mid Auth(y_m,B,A)\}$

let $x_n = dec<Private>\ k_B\ net?$ **in**

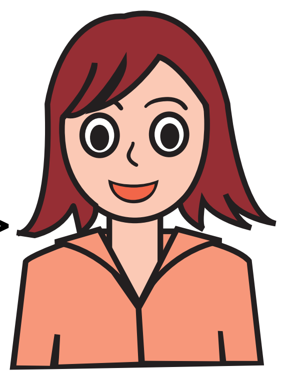
assume Auth(m,B,A)

$enc<T_A>\ pk_A\ (x_n,m)$

$x_n : Private \vee Un$

An extremely simple example

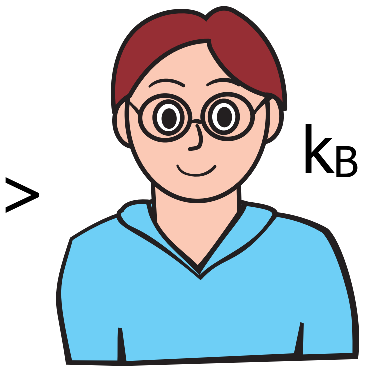
k_A : DK<T_A>



pk_A : PK<T_A>

public key

pk_B : PK<Private>



k_B : DK<Private>

n : Private

enc<Private> pk_B n

T_A=Private∨Un * {y_m:Un | Auth(y_m,B,A)}

let x_n = dec<Private> k_B net? **in**

assume Auth(m,B,A)

enc<T_A> pk_A (x_n,m)

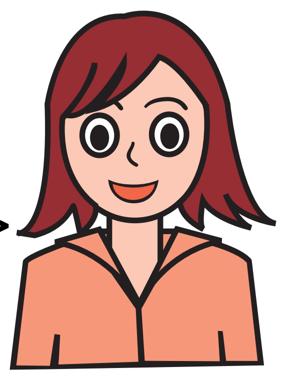
x_n : Private ∨ Un

let $y_n y_m$ = dec<T_A> k_A net? **in**

let (y_n, y_m) = $y_n y_m$ **in**

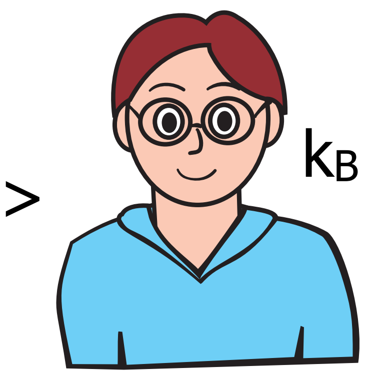
if y_n = n **then assert** Auth(y_m,B,A)

An extremely simple example



$k_A : DK\langle T_A\rangle$ $pk_A : PK\langle T_A\rangle$

public key
$pk_B : PK\langle Private\rangle$



$k_B : DK\langle Private\rangle$

$n : Private$ $enc\langle Private\rangle\ pk_B\ n$

$T_A = Private \vee Un * \{y_m : Un \mid Auth(y_m,B,A)\}$

let $x_n = dec\langle Private\rangle\ k_B\ net?$ **in**
assume $Auth(m,B,A)$

$enc\langle T_A\rangle\ pk_A\ (x_n,m)$ $x_n : Private \vee Un$

let $y_n y_m = dec\langle T_A\rangle\ k_A\ net?$ **in**
$y_n y_m : T_A \vee Un$
let $(y_n, y_m) = y_n y_m$ **in**
if $y_n = n$ **then assert** $Auth(y_m,B,A)$

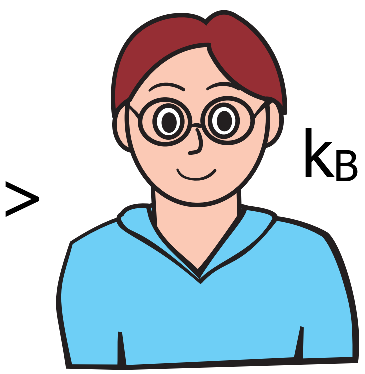
$enc\langle Un\rangle\ pk_A\ junk$

An extremely simple example

$k_A : DK<T_A>$ $pk_A : PK<T_A>$

public key

$pk_B : PK<Private>$



$k_B : DK<Private>$

$n : Private$ $enc<Private>\ pk_B\ n$ →

$T_A = Private \vee Un * \{y_m{:}Un \mid Auth(y_m,B,A)\}$

let $x_n = dec<Private>\ k_B\ net?$ **in**

assume $Auth(m,B,A)$

← $enc<T_A>\ pk_A\ (x_n,m)$ $x_n : Private \vee Un$

let $y_ny_m = dec<T_A>\ k_A\ net?$ **in**

case $y_ny_m' = y_ny_m : T_A \vee Un$ **of**

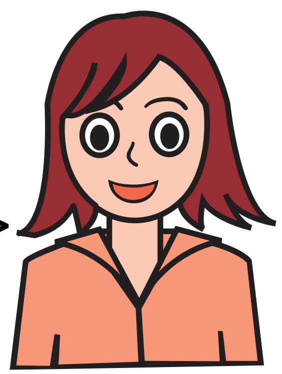
let $(y_n, y_m) = y_ny_m'$ **in**

if $y_n = n$ **then assert** $Auth(y_m,B,A)$

← $enc<Un>\ pk_A\ junk$

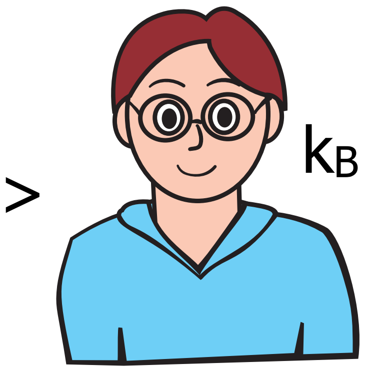
An extremely simple example

$k_A : \mathrm{DK}\langle T_A \rangle$



$pk_A : \mathrm{PK}\langle T_A \rangle$

public key
$pk_B : \mathrm{PK}\langle \mathrm{Private} \rangle$



$k_B : \mathrm{DK}\langle \mathrm{Private} \rangle$

$n : \mathrm{Private}$

$\mathrm{enc}\langle \mathrm{Private} \rangle\ pk_B\ n$

$T_A = \mathrm{Private} \vee \mathrm{Un} * \{y_m : \mathrm{Un} \mid \mathrm{Auth}(y_m, B, A)\}$

let $x_n = \mathrm{dec}\langle \mathrm{Private} \rangle\ k_B$ net? **in**
assume Auth(m,B,A)

$\mathrm{enc}\langle T_A \rangle\ pk_A\ (x_n, m)$ $x_n : \mathrm{Private} \vee \mathrm{Un}$

let $y_n y_m = \mathrm{dec}\langle T_A \rangle\ k_A$ net? **in**
case $y_n y_m' = y_n y_m : T_A \vee \mathrm{Un}$ **of**
let $(y_n, y_m) = y_n y_m'$ **in**
if $y_n = n$ **then assert** $\mathrm{Auth}(y_m, B, A)$

Honest sender case: $y_m : \{y_m : \mathrm{Un} \mid \mathrm{Auth}(y_m, B, A)\}$
assert succeeds

$\mathrm{enc}\langle \mathrm{Un} \rangle\ pk_A$ junk

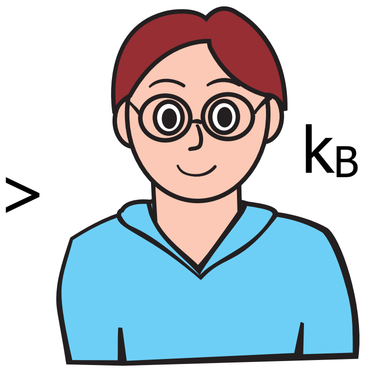
An extremely simple example

k_A : DK<T_A>

pk_A : PK<T_A>

public key

pk_B : PK<Private>



k_B : DK<Private>

n : Private

enc<Private> pk_B n

T_A=Private∨Un * {y_m:Un | Auth(y_m,B,A)}

let x_n = dec<Private> k_B net? **in**

assume Auth(m,B,A)

enc<T_A> pk_A (x_n,m)

x_n : Private ∨ Un

let $y_n y_m$ = dec<T_A> k_A net? **in**

case $y_n y_m'$ = $y_n y_m$: T_A ∨ Un **of**

let (y_n, y_m) = $y_n y_m'$ **in**

if y_n = n **then assert** Auth(y_m,B,A)

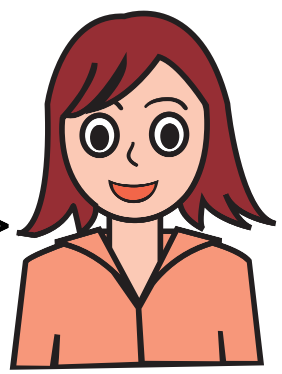
Dishonest sender case: y_n : Un, n : Private

Un ∩ Private = ∅ **so assert won't be executed**

enc<Un> pk_A junk

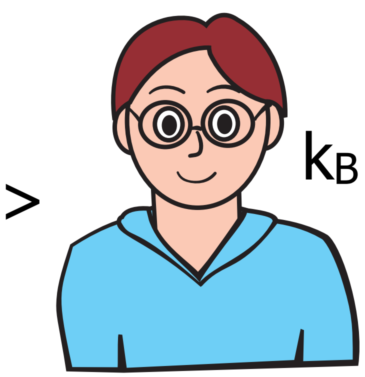
An extremely simple example

$k_A : DK\langle T_A \rangle$



$pk_A : PK\langle T_A \rangle$

public key
$pk_B : PK\langle Private \rangle$



$k_B : DK\langle Private \rangle$

$n : Private$

$enc\langle Private \rangle\ pk_B\ n$

$T_A = Private \vee Un * \{y_m{:}Un \mid Auth(y_m,B,A)\}$

let $x_n = dec\langle Private \rangle\ k_B$ net? **in**
assume Auth(m,B,A)

$enc\langle T_A \rangle\ pk_A\ (x_n,m)$ $x_n : Private \vee Un$

let $y_n y_m = dec\langle T_A \rangle\ k_A$ net? **in**
case $y_n y_m' = y_n y_m : T_A \vee Un$ **of**
let $(y_n, y_m) = y_n y_m'$ **in**
if $y_n = n$ **then assert** $Auth(y_m,B,A)$

Dishonest sender case: $y_n : Un,\ n : Private$
$Un \cap Private = \varnothing$ **so assert won't be executed**

$enc\langle Un \rangle\ pk_A$ junk

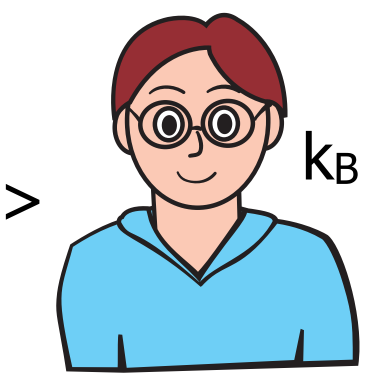
An extremely simple example

k_A : DK<T_A>

pk_A : PK<T_A>

public key

pk_B : PK<Private>



k_B : DK<Private>

n : Private

enc<Private> pk_B n

T_A=Private∨Un * {y_m:Un | Auth(y_m,B,A)}

let x_n = dec<Private> k_B net? **in**

assume Auth(m,B,A)

enc<T_A> pk_A (x_n,m)

x_n : Private ∨ Un

let $y_n y_m$ = dec<T_A> k_A net? **in**

case $y_n y_m'$ = $y_n y_m$: T_A ∨ Un **of**

let (y_n, y_m) = $y_n y_m'$ **in**

if y_n = n **then assert** Auth(y_m,B,A)

Dishonest sender case: y_n : Un, n : Private

Un ∩ Private = ∅ **so assert won't be executed**

enc<Un> pk_A junk

F7v1 can't handle this

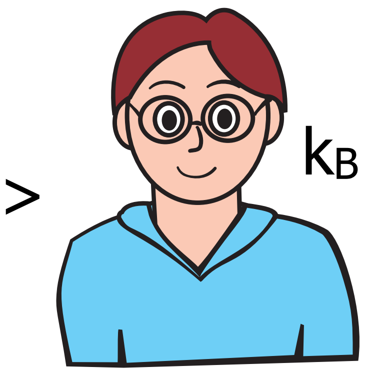
An extremely simple example

simplified variant of Needham-Schroeder-Lowe

k_A : DK<T_A>

pk_A : PK<T_A>

public key
pk_B : PK<Private>



k_B : DK<Private>

n : Private

enc<Private> pk_B n

T_A=Private∨Un * {y_m:Un | Auth(y_m,B,A)}

let x_n = dec<Private> k_B net? **in**
assume Auth(m,B,A)

enc<T_A> pk_A (x_n,m)

x_n : Private ∨ Un

let $y_n y_m$ = dec<T_A> k_A net? **in**
case $y_n y_m'$ = $y_n y_m$: T_A ∨ Un **of**
let (y_n, y_m) = $y_n y_m'$ **in**
if y_n = n **then assert** Auth(y_m,B,A)

Dishonest sender case: y_n : Un, n : Private
Un ∩ Private = ∅ **so assert won't be executed**

enc<Un> pk_A junk

F7v1 can't handle this

We propose ...

- a new type-system for verifying protocol implementations
  - combines the refinement types from F7v1/RCF [BBFGM '08] with *union*, *intersection*, and *polymorphic* types ($RCF^{\forall}_{\wedge\vee}$)
  - novel ability: statically reasoning about *disjointness of types*

We propose ...

- a new type-system for verifying protocol implementations
  - combines the refinement types from F7v1/RCF [BBFGM '08] with *union*, *intersection*, and *polymorphic* types ($RCF^{\forall}_{\wedge\vee}$)
  - novel ability: statically reasoning about *disjointness of types*
- What does this buy us?
  1. successfully type-checking larger class of protocols

     e.g. authenticity achieved by showing knowledge of secret data (NSL, ZK sign)
  2. a proper sealing-based encoding of asymmetric cryptography
  3. type-checking applications based on NI-ZK (DAA, Civitas, etc.)

We propose ...

- a new type-system for verifying protocol implementations
  - combines the refinement types from F7v1/RCF [BBFGM '08] with *union*, *intersection*, and *polymorphic* types ($RCF^{\forall}_{\wedge\vee}$)
  - novel ability: statically reasoning about *disjointness of types*
- What does this buy us?
  1. successfully type-checking larger class of protocols

     e.g. authenticity achieved by showing knowledge of secret data (NSL, ZK sign)
  2. a proper sealing-based encoding of asymmetric cryptography
  3. type-checking applications based on NI-ZK (DAA, Civitas, etc.)

\+ Machine-checked soundness proof + cool implementation

Encoding symbolic cryptography using dynamic seals

Symbolic cryptography

- RCF doesn’t have any primitive for cryptography
- Instead, crypto primitives can be encoded using **dynamic sealing** [Morris, CACM ’73]
- Advantage: adding new crypto primitives doesn’t change RCF calculus, or type system, or any proof
- Nice idea that (to a certain extent) works for: symmetric and PK encryption, signatures, hashes, MACs
- Dynamic sealing not primitive either
  - encoded using references, lists, pairs and functions

    $\mathrm{Seal}\langle\alpha\rangle = (\alpha \to \mathrm{Un}) * (\mathrm{Un} \to \alpha)$

    $\mathrm{mkSeal} : \forall\alpha.\ \mathrm{unit} \to \mathrm{Seal}\langle\alpha\rangle$

Symmetric encryption

$\text{Key}\langle\alpha\rangle = \text{Seal}\langle\alpha\rangle = (\alpha \to \text{Un}) * (\text{Un} \to \alpha)$

$\text{mkKey} = \Lambda\alpha.\text{mkSeal}\langle\alpha\rangle$

$\text{senc} = \Lambda\alpha.\lambda k{:}\text{Key}\langle\alpha\rangle.\lambda m{:}\alpha.\ (\text{fst } k)\ m \quad : \forall\alpha.\text{Key}\langle\alpha\rangle \to \alpha \to \text{Un}$

$\text{sdec} = \Lambda\alpha.\lambda k{:}\text{Key}\langle\alpha\rangle.\lambda n{:}\text{Un}.\ (\text{snd } k)\ n \quad : \forall\alpha.\text{Key}\langle\alpha\rangle \to \text{Un} \to \alpha$

- Dynamic sealing directly corresponds to sym. encryption
    - First observed by [Sumii & Pierce, ’03 & ’07]

“Public”-key encryption

$DK\langle\alpha\rangle = Seal\langle\alpha\rangle = (\alpha \to Un) * (Un \to \alpha)$

$PK\langle\alpha\rangle = \alpha \to Un$

$mkDK = \Lambda\alpha.mkSeal\langle\alpha\rangle \quad : \forall\alpha.unit \to DK\langle\alpha\rangle$

$mkPK = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\ fst\ dk \quad : \forall\alpha.DK\langle\alpha\rangle \to PK\langle\alpha\rangle$

$enc = \Lambda\alpha.\lambda pk{:}PK\langle\alpha\rangle.\lambda m{:}\alpha.\ pk\ m \quad : \forall\alpha.PK\langle\alpha\rangle \to \alpha \to Un$

$dec = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\lambda n{:}Un.\ (snd\ k)\ n \quad : \forall\alpha.DK\langle\alpha\rangle \to Un \to \alpha$

"Public"-key encryption

$DK\langle\alpha\rangle = Seal\langle\alpha\rangle = (\alpha \to Un) * (Un \to \alpha)$

$PK\langle\alpha\rangle = \alpha \to Un$

$mkDK = \Lambda\alpha.mkSeal\langle\alpha\rangle \quad : \forall\alpha.unit \to DK\langle\alpha\rangle$

$mkPK = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\ fst\ dk \quad : \forall\alpha.DK\langle\alpha\rangle \to PK\langle\alpha\rangle$

$enc = \Lambda\alpha.\lambda pk{:}PK\langle\alpha\rangle.\lambda m{:}\alpha.\ pk\ m \quad : \forall\alpha.PK\langle\alpha\rangle \to \alpha \to Un$

$dec = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\lambda n{:}Un.\ (snd\ k)\ n \quad : \forall\alpha.DK\langle\alpha\rangle \to Un \to \alpha$

- A "public" key $pk: PK\langle\alpha\rangle$ is only public when α is tainted!

"Public"-key encryption

DK<α> = Seal<α> = (α→Un) * (Un→α)

PK<α> = α→Un

mkDK = Λα.mkSeal<α> : ∀α.unit→DK<α>

mkPK = Λα.λdk:DK<α>. fst dk : ∀α.DK<α>→PK<α>

enc = Λα.λpk:PK<α>.λm:α. pk m : ∀α.PK<α>→α→Un

dec = Λα.λdk:DK<α>.λn:Un. (snd k) n : ∀α.DK<α>→Un→α

- A "public" key pk: PK<α> is only public when α is tainted!
- A function type T→U is public only when
  - return type U is public
    (otherwise λ_:unit.m_{secret} would be public)
  - argument type T is tainted
    (otherwise λk:Key<Private>.c_{pub}!(senc k m_{secret}) is public)

"Public"-key encryption

$DK\langle\alpha\rangle = Seal\langle\alpha\rangle = (\alpha \to Un) * (Un \to \alpha)$

$PK\langle\alpha\rangle = \alpha \to Un$

$mkDK = \Lambda\alpha.mkSeal\langle\alpha\rangle \quad : \forall\alpha.unit \to DK\langle\alpha\rangle$

$mkPK = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\ fst\ dk \quad : \forall\alpha.DK\langle\alpha\rangle \to PK\langle\alpha\rangle$

$enc = \Lambda\alpha.\lambda pk{:}PK\langle\alpha\rangle.\lambda m{:}\alpha.\ pk\ m \quad : \forall\alpha.PK\langle\alpha\rangle \to \alpha \to Un$

$dec = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\lambda n{:}Un.\ (snd\ k)\ n \quad : \forall\alpha.DK\langle\alpha\rangle \to Un \to \alpha$

- A "public" key pk: $PK\langle\alpha\rangle$ is only public when α is tainted!
- A function
  - retur
    (otherwise λ_ ... ic)
  - argument type T is tainted
    (otherwise $\lambda k{:}Key\langle Private\rangle.c_{pub}!(senc\ k\ m_{secret})$ is public)

Remember:
in NSL α is Private
(not public and **not tainted**)
$\Rightarrow$ strange attacker model

Public-key encryption - FIXED

DK<α> = Seal<α∨Un> = ((α∨Un)→Un) * ((α∨Un)→α)

PK<α> = (α∨Un)→Un

mkDK = Λα.mkSeal<α> : ∀α.unit→DK<α>

mkPK = Λα.λdk:DK<α>. fst dk : ∀α.DK<α>→PK<α>

enc = Λα.λpk:PK<α>.λm:α. pk m : ∀α.PK<α>→α→Un

dec = Λα.λdk:DK<α>.λn:Un. (snd k) n : ∀α.DK<α>→Un→(α∨Un)

- Public keys are now always public
  - A type T∨Un is always tainted since Un <: T∨Un for all T

Public-key encryption - FIXED

$\mathrm{DK}\langle\alpha\rangle = \mathrm{Seal}\langle\alpha\vee\mathrm{Un}\rangle = ((\alpha\vee\mathrm{Un})\rightarrow\mathrm{Un}) * ((\alpha\vee\mathrm{Un})\rightarrow\alpha)$

$\mathrm{PK}\langle\alpha\rangle = (\alpha\vee\mathrm{Un})\rightarrow\mathrm{Un}$

$\mathrm{mkDK} = \Lambda\alpha.\mathrm{mkSeal}$

$\mathrm{mkPK} = \Lambda\alpha.\lambda dk{:}\mathrm{DK}\langle\alpha\rangle.$

$\mathrm{enc} = \Lambda\alpha.\lambda pk{:}\mathrm{PK}\langle\alpha\rangle.\lambda m{:}\alpha.\ pk\ m \quad : \forall\alpha.\mathrm{PK}\langle\alpha\rangle\rightarrow\alpha\rightarrow\mathrm{Un}$

$\mathrm{dec} = \Lambda\alpha.\lambda dk{:}\mathrm{DK}\langle\alpha\rangle.\lambda n{:}\mathrm{Un}.\ (\mathrm{snd}\ k)\ n \quad : \forall\alpha.\mathrm{DK}\langle\alpha\rangle\rightarrow\mathrm{Un}\rightarrow(\alpha\vee\mathrm{Un})$

Union type: sealed values can come from honest participant (α) or from the attacker (Un)

- Public keys are now always public
  - A type $T\vee\mathrm{Un}$ is always tainted since $\mathrm{Un} <: T\vee\mathrm{Un}$ for all T

Public-key encryption - FIXED

$DK\langle\alpha\rangle = Seal\langle\alpha \vee Un\rangle = ((\alpha \vee Un) \to Un) * ((\alpha \vee Un) \to \alpha)$

$PK\langle\alpha\rangle = (\alpha \vee Un) \to Un$

$mkDK = \Lambda\alpha.mkSeal\langle\alpha\rangle \quad : \forall\alpha.unit \to DK\langle\alpha\rangle$

$mkPK = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\ fst\ dk \quad : \forall\alpha.DK\langle\alpha\rangle \to PK\langle\alpha\rangle$

$enc = \Lambda\alpha.\lambda pk{:}PK\langle\alpha\rangle.\lambda m{:}\alpha.\ pk\ m \quad : \forall\alpha.PK\langle\alpha\rangle \to \alpha \to Un$

$dec = \Lambda\alpha.\lambda dk{:}DK\langle\alpha\rangle.\lambda n{:}Un.\ (snd\ k)\ n \quad : \forall\alpha.DK\langle\alpha\rangle \to Un \to (\alpha \vee Un)$

- Public keys are now always public
  - A type $T \vee Un$ is always tainted since $Un <: T \vee Un$ for all T

Public-key encryption - FIXED

$DK<\alpha> \;=\; Seal<\alpha \vee Un> \;=\; ((\alpha \vee Un) \to Un) * ((\alpha \vee Un) \to \alpha)$

$PK<\alpha> \;=\; (\alpha \vee Un) \to Un$

$mkDK = \Lambda\alpha.mkSeal<\alpha>$

$mkPK = \Lambda\alpha.\lambda dk{:}DK<\alpha>.\ fst\ dk$

Union types introduced by subtyping $m{:}\alpha$ and $\alpha <: \alpha \vee Un$

$enc = \Lambda\alpha.\lambda pk{:}PK<\alpha>.\lambda m{:}\alpha.\ pk\ m \quad : \forall\alpha.PK<\alpha> \to \alpha \to Un$

$dec = \Lambda\alpha.\lambda dk{:}DK<\alpha>.\lambda n{:}Un.\ (snd\ k)\ n \quad : \forall\alpha.DK<\alpha> \to Un \to (\alpha \vee Un)$

- Public keys are now always public
  - A type $T \vee Un$ is always tainted since $Un <: T \vee Un$ for all T

Digital signatures

$SK\langle\alpha\rangle = Seal\langle\alpha\rangle = (\alpha \rightarrow Un) * (Un \rightarrow \alpha)$

$VK\langle\alpha\rangle = Un \rightarrow \alpha$

$mkSK = \Lambda\alpha.mkSeal\langle\alpha\rangle$

$mkVK = \Lambda\alpha.\lambda sk{:}SK\langle\alpha\rangle.\ snd\ sk \qquad : \forall\alpha.SK\langle\alpha\rangle \rightarrow VK\langle\alpha\rangle$

$sign = \Lambda\alpha.\lambda sk{:}SK\langle\alpha\rangle.\lambda m{:}\alpha.\ (fst\ sk)\ m \qquad : \forall\alpha.SK\langle\alpha\rangle \rightarrow \alpha \rightarrow Un$

$verify = \Lambda\alpha.\lambda vk{:}VK\langle\alpha\rangle.\lambda m{:}Un.\lambda s{:}Any.$

let m'=vk s **in**

if m'=m **then** m'

else failwith "bad signature"

$: \forall\alpha.VK\langle\alpha\rangle \rightarrow Un \rightarrow Any \rightarrow \alpha$

Digital signatures

SK<α> = Seal<α> = (α→Un) * (Un→α)

VK<α> = Un→α

mkSK = Λα.mkSeal<α>

mkVK = Λα.λsk:SK<α>. snd sk : ∀α.SK<α>→VK<α>

sign = Λα.λsk:SK<α>.λm:α. (fst sk) m : ∀α.SK<α>→α→Un

verify = Λα.λvk:VK<α>.λm:Un.λs:Any.
let m′=vk s **in**
if m′=m **then** m′
else failwith "bad signature"
: ∀α.VK<α>→Un→Any→α

- Verification key vk: VK<α> is public only when α is public!
  - Strange, since verify leaks only one additional bit about m (i.e. is m a proper signature of n or not)

Digital signatures - FIXED

SK<α> = (α→Un) * VK<α>

VK<α> = Un→(Any→α)∧(Un→Un)

mkSK = ...

: ∀α.unit→SK<α>

mkVK = Λα.λsk:SK<α>. snd sk : ∀α.SK<α>→VK<α>

sign = Λα.λsk:SK<α>.λm:α. (fst sk) m : ∀α.SK<α>→α→Un

verify = Λα.λvk:VK<α>. λn:Un. λm:Any. vk n m

: ∀α.VK<α>→Un→Any→α

Digital signatures - FIXED

SK<α> = (α→Un) * VK<α>

VK<α> = Un→(Any→α)∧(Un→Un)

mkSK = ...

Verification keys are always public
T∧Un is always public since T∧Un <: Un

: ∀α.unit→SK<α>

mkVK = Λα.λsk:SK<α>. snd sk : ∀α.SK<α>→VK<α>

sign = Λα.λsk:SK<α>.λm:α. (fst sk) m : ∀α.SK<α>→α→Un

verify = Λα.λvk:VK<α>. λn:Un. λm:Any. vk n m

: ∀α.VK<α>→Un→Any→α

Digital signatures - FIXED

SK<α> = (α→Un) * VK<α>

VK<α> = Un→(Any→α)∧(Un→Un)

mkSK = Λα.λ_:unit. **let** (s,u) = mkSeal () in
let v = λn:Un. λm:Any ; Un.
if m = u n **as** z **then** z
else failwith "bad signature"
in (s, v) : ∀α.unit→SK<α>

mkVK = Λα.λsk:SK<α>. snd sk : ∀α.SK<α>→VK<α>

sign = Λα.λsk:SK<α>.λm:α. (fst sk) m : ∀α.SK<α>→α→Un

verify = Λα.λvk:VK<α>. λn:Un. λm:Any. vk n m
: ∀α.VK<α>→Un→Any→α

Digital signatures - FIXED

Introduces intersection of 2 function types

```
SK<α> = (α→Un) * VK<α>
VK<α> = Un→(Any→α)∧(Un→Un)
mkSK = Λα.λ_:unit. let (s,u) = mkSeal () in
                    let v = λn:Un. λm:Any ; Un.
                      if m = u n as z then z
                      else failwith “bad signature”
                    in (s, v)                  : ∀α.unit→SK<α>
mkVK = Λα.λsk:SK<α>. snd sk                    : ∀α.SK<α>→VK<α>
sign = Λα.λsk:SK<α>.λm:α. (fst sk) m           : ∀α.SK<α>→α→Un
verify = Λα.λvk:VK<α>. λn:Un. λm:Any. vk n m
                                               : ∀α.VK<α>→Un→Any→α
```

Digital signatures - FIXED

```
SK<α> = (α→Un) * VK<α>
VK<α> = Un→(Any→α)∧(Un→Un)
mkSK = Λα.λ_:unit. let (s,u) = mkSeal () in
                   let v = λn:Un. λm:Any ; Un.
                     if m = u n as z then z
                     ...n "bad signature"
                                                  : ∀α.unit→SK<α>
mkVK = ... sk                                     : ∀α.SK<α>→VK<α>
sign = Λα.λsk:SK<α>.λm:α. (fst sk) m              : ∀α.SK<α>→α→Un
verify = Λα.λvk:VK<α>. λn:Un. λm:Any. vk n m
                                                  : ∀α.VK<α>→Un→Any→α
```

Introduces intersection of 2 function types

If m : Any, u n : α
then z : Any ∧ α <: α

Digital signatures - FIXED

$SK\langle\alpha\rangle = (\alpha \to Un) * VK\langle\alpha\rangle$

$VK\langle\alpha\rangle = Un \to (Any \to \alpha) \wedge (Un \to Un)$

$mkSK = \Lambda\alpha.\lambda\_{:}unit.$ **let** $(s,u) = mkSeal\ ()$ in

let $v = \lambda n{:}Un.\ \lambda m{:}Any\ ;\ Un.$

if $m = u\ n$ **as** z **then** z

n "bad

Introduces intersection of 2 function types

If $m : Any,\ u\ n : \alpha$ then $z : Any \wedge \alpha <: \alpha$

If $m : Un,\ u\ n : \alpha$ then $z : Un \wedge \alpha <: Un$

mkVK =

$sign = \Lambda\alpha.\lambda sk{:}SK\langle\alpha\rangle.\lambda m{:}\alpha.\ (fst\ sk)\ m \quad : \forall\alpha.SK\langle\alpha\rangle \to \alpha \to Un$

$verify = \Lambda\alpha.\lambda vk{:}VK\langle\alpha\rangle.\ \lambda n{:}Un.\ \lambda m{:}Any.\ vk\ n\ m$

$: \forall\alpha.VK\langle\alpha\rangle \to Un \to Any \to \alpha$

Digital signatures - FIXED

```
SK<α> = (α→Un) * VK<α>
VK<α> = Un→(Any→α)∧(Un→Un)
mkSK = Λα.λ_:unit. let (s,u) = mkSeal () in
                   let v = λn:Un. λm:Any ; Un.
                     if m = u n as z then z
                     else failwith "bad signature"
                   in (s, v)                  : ∀α.unit→SK<α>
mkVK = Λα.λsk:SK<α>. snd sk                   : ∀α.SK<α>→VK<α>
sign = Λα.λsk:SK<α>.λm:α. (fst sk) m          : ∀α.SK<α>→α→Un
verify = Λα.λvk:VK<α>. λn:Un. λm:Any. vk n m
                                          : ∀α.VK<α>→Un→Any→α
```

Union and intersection types allow us to give a more faithful seal-based encoding of asymmetric crypto

Encoding zero-knowledge proofs

Very simplified DAA-sign

assume $\forall m. (\exists f.\ Send(f,m) \wedge OkTPM(f)) \Rightarrow Authenticate(m));$

$T_i = \{x_f : Private \mid OkTPM(x_f)\}$ vki : VK<T_i>

TPM/User

Verifier

f : T_i
cert = sign<T_i> ski f
m : Un
assume Send(f, m)

Very simplified DAA-sign

assume $\forall m. (\exists f.\ Send(f,m) \wedge OkTPM(f)) \Rightarrow Authenticate(m));$

$T_i = \{x_f : Private \mid OkTPM(x_f)\}$ $\quad$ $vki : VK<T_i>$

TPM/User

Verifier

$f : T_i$
$cert = sign<T_i>\ ski\ f$
$m : Un$
assume Send(f, m)

$\xrightarrow{zk\text{-}create_{daa}\ (vki,\ m,\ f,\ cert)}$

Very simplified DAA-sign

assume $\forall m. (\exists f. \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m));$

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ vki : VK<T_i>

TPM/User

Verifier

f : T_i
cert = sign<T_i> ski f
m : Un
assume Send(f, m)

zk-create$_{daa}$ (vki, m, f, cert) →

ZK proof shows that
“verify<T_i> vki cert f” succeeds

Very simplified DAA-sign

assume $\forall$m. ($\exists$f. Send(f,m) $\wedge$ OkTPM(f)) $\Rightarrow$ Authenticate(m));

T_i = {x_f : Private | OkTPM(x_f)}   vki : VK<T_i>

TPM/User

Verifier

f : T_i
cert = sign<T_i> ski f
m : Un
assume Send(f, m)

zk-create$_{daa}$ (vki, m, f, cert) →

Without revealing f or cert (secret witnesses)

ZK proof shows that "verify<T_i> vki cert f" succeeds

Very simplified DAA-sign

assume $\forall m.\ (\exists f.\ \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m));$

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ vki : VK<T_i>

TPM/User

Verifier

f : T_i
cert = sign<T_i> ski f
m : Un
assume Send(f, m)

zk-create$_{daa}$ (vki, m, f, cert) →

Without revealing f or cert (secret witnesses)

ZK proof shows that "verify<T_i> vki cert f" succeeds

Proof non-malleable, authenticity of m proved by showing knowledge of secret f

Very simplified DAA-sign

assume $\forall$m. ($\exists$f. Send(f,m) $\wedge$ OkTPM(f)) $\Rightarrow$ Authenticate(m));

T_i = {x_f : Private | OkTPM(x_f)}   vki : VK<T_i>

TPM/User

f : T_i
cert = sign<T_i> ski f
m : Un
assume Send(f, m)

Verifier

Without revealing f or cert (secret witnesses)

zk-create$_{daa}$ (vki, m, f, cert)

let (y_2,y_3) = zk-verify$_{daa}$ c? vki **in**
assert Authenticate(y_2)

ZK proof shows that "verify<T_i> vki cert f" succeeds

Proof non-malleable, authenticity of m proved by showing knowledge of secret f

High-level specification

assume $\forall m. (\exists f. \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m));$

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ $\quad$ $vki : VK\langle T_i \rangle$

zkdef daa =

matched = $[y_{vki} : VK\langle T_i \rangle]$

returned = $[y_m : \mathrm{Un}]$

secret = $[x_f : T_i, x_{cert} : \mathrm{Un}]$

statement = $[x_f = \mathrm{verify}\langle T_i \rangle\ y_{vki}\ x_{cert}\ x_f]$

promise = $[\mathrm{Send}(x_f, y_m)]$.

High-level specification

assume $\forall m. (\exists f.\ \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m))$;

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ $\quad$ $vki : VK\langle T_i \rangle$

Public value known to the verifier

zkdef daa =

matched = [$y_{vki} : VK\langle T_i \rangle$]

returned = [$y_m : \mathrm{Un}$]

secret = [$x_f : T_i,\ x_{cert} : \mathrm{Un}$]

statement = [$x_f = \mathrm{verify}\langle T_i \rangle\ y_{vki}\ x_{cert}\ x_f$]

promise = [$\mathrm{Send}(x_f, y_m)$].

High-level specification

assume $\forall m.\ (\exists f.\ \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m));$

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ $\quad$ $vki : VK\langle T_i\rangle$

zkdef daa =

matched = [y_{vki} :

Public value not known to the verifier

returned = [y_m : Un]

secret = [x_f : T_i, x_{cert} : Un]

statement = [x_f = verify$\langle T_i\rangle$ y_{vki} x_{cert} x_f]

promise = [Send(x_f,y_m)].

High-level specification

assume $\forall m. (\exists f. \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m));$

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ $\quad$ $vki : VK\langle T_i\rangle$

zkdef daa =

matched = $[y_{vki} : VK\langle T_i\rangle]$

returned = $[y_m : \mathrm{Un}]$

secret = $[x_f : T_i, x_{cert} : \mathrm{Un}]$

statement = $[x_f = \mathrm{verify}\langle T_i\rangle\ y_{vki}\ x_{cert}\ x_f]$

promise = $[\mathrm{Send}(x_f,y_m)]$.

Witnesses, never revealed
(but prover has to know them)

High-level specification

assume $\forall m. (\exists f. \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m)$;

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ $\quad$ $vki : VK<T_i>$

zkdef daa =

matched = $[y_{vki} : VK<T_i>]$

returned = $[y_m : \mathrm{Un}]$

secret = $[x_f : T_i,\ x_{cert} : \mathrm{Un}]$

statement = $[x_f = \mathrm{verify}<T_i>\ y_{vki}\ x_{cert}\ x_f]$

promise = $[\mathrm{Send}(x_f,y_m)]$.

Statement of the proof
(positive Boolean formula)

High-level specification

assume $\forall m. (\exists f. \mathrm{Send}(f,m) \wedge \mathrm{OkTPM}(f)) \Rightarrow \mathrm{Authenticate}(m));$

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$ vki : $VK\langle T_i\rangle$

zkdef daa =

matched = $[y_{vki} : VK\langle T_i\rangle]$

returned = $[y_m : \mathrm{Un}]$

secret = $[x_f : T_i, x_{cert} : \mathrm{Un}]$

statement = $[x_f = \mathrm{verify}\langle T_i\rangle\ y_{vki}\ x_{cert}\ x_f]$

promise = $[\mathrm{Send}(x_f,y_m)]$.

Logical formula that is conveyed by the proof if prover is honest

Generated implementation

T_i = {x_f : Private | OkTPM(x_f)}

T_{daa} = y_{vki} : VK<T_i> * y_m : Un * x_f : T_i * x_{cert} : {x:Un | Send(x_f,y_m)}

Generated implementation

T_i = {x_f : Private | OkTPM(x_f)}

T_{daa} = y_{vki} : VK<T_i> * y_m : Un * x_f : T_i * x_{cert} : {x:Un | Send(x_f,y_m)}

k_{daa} : Seal<$T_{daa} \vee$Un>

Generated implementation

$T_i = \{x_f : \text{Private} \mid \text{OkTPM}(x_f)\}$

$T_{daa} = y_{vki} : \text{VK}\langle T_i \rangle * y_m : \text{Un} * x_f : T_i * x_{cert} : \{x{:}\text{Un} \mid \text{Send}(x_f, y_m)\}$

$k_{daa} : \text{Seal}\langle T_{daa} \vee \text{Un} \rangle$

$\text{zk-create}_{daa} = \lambda w{:}T_{daa} \vee \text{Un}.\ (\text{fst}\ k_{daa})\ v \qquad : T_{daa} \vee \text{Un} \rightarrow \text{Un}$

Generated implementation

$T_i = \{x_f : \text{Private} \mid \text{OkTPM}(x_f)\}$

$T_{daa} = y_{vki} : \text{VK}\langle T_i\rangle * y_m : \text{Un} * x_f : T_i * x_{cert} : \{x{:}\text{Un} \mid \text{Send}(x_f, y_m)\}$

$k_{daa} : \text{Seal}\langle T_{daa} \vee \text{Un}\rangle$

$\text{zk-create}_{daa} = \lambda w{:}T_{daa}\vee\text{Un}.\ (\text{fst}\ k_{daa})\ v \qquad : T_{daa}\vee\text{Un} \rightarrow \text{Un}$

$\text{zk-public}_{daa} = \lambda z{:}\text{Un}.\ \textbf{case}\ w' = (\text{snd}\ k_{daa})\ z : T_{daa}\vee\text{Un}\ \textbf{of} \qquad : \text{Un} \rightarrow \text{Un}$
$\qquad \textbf{let}\ (y_{vki}, y_m, s) = w'\ \textbf{in}\ (y_{vki}, y_m)$

Generated implementation

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$

$T_{daa} = y_{vki} : \mathrm{VK}<T_i> * y_m : \mathrm{Un} * x_f : T_i * x_{cert} : \{x{:}\mathrm{Un} \mid \mathrm{Send}(x_f, y_m)\}$

$k_{daa} : \mathrm{Seal}<T_{daa} \vee \mathrm{Un}>$

$\text{zk-create}_{daa} = \lambda w{:}T_{daa} \vee \mathrm{Un}.\ (\mathrm{fst}\ k_{daa})\ v \qquad : T_{daa} \vee \mathrm{Un} \rightarrow \mathrm{Un}$

$\text{zk-public}_{daa} = \lambda z{:}\mathrm{Un}.\ \mathbf{case}\ w' = (\mathrm{snd}\ k_{daa})\ z : T_{daa} \vee \mathrm{Un}\ \mathbf{of} \qquad : \mathrm{Un} \rightarrow \mathrm{Un}$
$\quad \mathbf{let}\ (y_{vki}, y_m, s) = w'\ \mathbf{in}\ (y_{vki}, y_m)$

$\text{zk-verify}_{daa} = \lambda z{:}\mathrm{Un}.\ \lambda y_{vki}' : \mathrm{VK}<T_i>;\ \mathrm{Un}.$
$\quad \mathbf{case}\ w = (\mathrm{snd}\ k_{daa})\ z : T_{daa} \vee \mathrm{Un}\ \mathbf{of}$
$\quad \mathbf{let}\ (y_{vki}, y_m, x_f, x_{cert}) = w\ \mathbf{in}$
$\quad \mathbf{if}\ y_{vki} = y_{vki}'\ \mathbf{as}\ y_{vki}''\ \mathbf{then}$
$\qquad \mathbf{if}\ x_f = \mathrm{verify}<T_i>\ y_{vki}''\ x_{cert}\ x_f\ \mathbf{then}\ (y_m)$
$\qquad \mathbf{else}$ failwith "statement not valid"
$\quad \mathbf{else}$ failwith "y_{vki} does not match"

Generated implementation

$T_i = \{x_f : \text{Private} \mid \text{OkTPM}(x_f)\}$

$T_{daa} = y_{vki} : \text{VK}<T_i> * y_m : \text{Un} * x_f : T_i * x_{cert} : \{x{:}\text{Un} \mid \text{Send}(x_f,y_m)\}$

$k_{daa} : \text{Seal}<T_{daa} \vee \text{Un}>$

$\text{zk-create}_{daa} = \lambda w{:}T_{daa}\vee\text{Un}.\ (\text{fst } k_{daa})\ v \qquad : T_{daa}\vee\text{Un} \rightarrow \text{Un}$

$\text{zk-public}_{daa} = \lambda z{:}\text{Un}.\ \textbf{case } w' = (\text{snd } k_{daa})\ z : T_{daa}\vee\text{Un}\ \textbf{of} \qquad : \text{Un}\rightarrow\text{Un}$
$\qquad \textbf{let } (y_{vki}, y_m, s) = w'\ \textbf{in } (y_{vki}, y_m)$

$\text{zk-verify}_{daa} = \lambda z{:}\text{Un}.\ \lambda y_{vki}' : \text{VK}<T_i>\textbf{;}\ \text{Un}.$
$\qquad \textbf{case } w = (\text{snd } k_{daa})\ z : T_{daa}\vee\text{Un}\ \textbf{of}$
$\qquad \textbf{let } (y_{vki}, y_m, x_f, x_{cert}) = w\ \textbf{in}$
$\qquad \textbf{if } y_{vki} = y_{vki}'\ \textbf{as } y_{vki}''\ \textbf{then}$
$\qquad\quad \textbf{if } x_f = \text{verify}<T_i>\ y_{vki}''\ x_{cert}\ x_f\ \textbf{then } (y_m)$
$\qquad\quad$ **else** failwith "statement not valid"
$\qquad$ **else** failwith "y_{vki} does not match"

$: \text{Un}\rightarrow((y_{vki}{:}\text{VK}<T_i>\rightarrow\{y_m{:}\text{Un}\mid\exists x_f, x_{cert}.\ \text{OkTPM}(x_f)\wedge\text{Send}(x_f,y_m)\})\wedge(\text{Un}\rightarrow\text{Un})$

Case #1: honest verifier, honest prover

T_i = {x_f : Private | OkTPM(x_f)}

T_{daa} = y_{vki} : VK<T_i> * y_m : Un * x_f : T_i * x_{cert} : {x:Un | Send(x_f,y_m)}

k_{daa} : Seal<$T_{daa} \vee$Un>

zk-create$_{daa}$ (vki, m, f, cert)

zk-verify$_{daa}$ =

λz:Un. λy_{vki}' : VK<T_i>; Un. — y_{vki}' : VK<T_{vki}>

case w = (snd k_{daa}) z : $T_{daa} \vee$Un **of** — w : T_{daa}

let (y_{vki}, y_m, x_f, x_{cert}) = w **in** — Send(x_f,y_m) — y_{vki} : VK<T_{vki}>

if y_{vki} = y_{vki}' **as** y_{vki}'' **then** — y_{vki}'' : VK<T_{vki}>

  if x_f = verify<T_i> y_{vki}'' x_{cert} x_f **then** (y_m) — OkTPM(x_f) — y_m : Un

  else failwith "statement not valid"

else failwith "y_{vki} does not match"

: Un→((y_{vki}:VK<T_i>→{y_m:Un|∃x_f, x_{cert}. OkTPM(x_f)∧Send(x_f,y_m)})∧(Un→Un)

Case #2: honest verifier, dishonest prover

$T_i = \{x_f : \text{Private} \mid \text{OkTPM}(x_f)\}$

$T_{daa} = y_{vki} : \text{VK}<T_i> * y_m : \text{Un} * x_f : T_i * x_{cert} : \{x{:}\text{Un} \mid \text{Send}(x_f,y_m)\}$

$k_{daa} : \text{Seal}<T_{daa} \vee \text{Un}>$

zk-create$_{daa}$ junk

zk-verify$_{daa}$ =

$\lambda z{:}\text{Un}.\ \lambda y_{vki}' : \text{VK}<T_i>;\ \text{Un}.$	$y_{vki}' : \text{VK}<T_{vki}>$
case $w = (\text{snd } k_{daa})\ z : T_{daa} \vee \text{Un}$ **of**	$w : \text{Un}$
let $(y_{vki}, y_m, x_f, x_{cert}) = w$ **in**	$\cancel{\text{Send}(x_f,y_m)}$ $x_f : \text{Un}$
if $y_{vki} = y_{vki}'$ **as** y_{vki}'' **then**	$y_{vki}'' : \text{Un} \wedge \text{VK}<T_{vki}>$
if $x_f = \text{verify}<T_i>\ y_{vki}''\ x_{cert}\ x_f$ **then** (y_m)	"$\text{Un} \cap \text{Private} = \varnothing$"; (y_m) dead code
else failwith "statement not valid"	
else failwith "y_{vki} does not match"	

$: \text{Un} \rightarrow ((y_{vki}{:}\text{VK}<T_i> \rightarrow \{y_m{:}\text{Un} \mid \exists x_f, x_{cert}.\ \text{OkTPM}(x_f) \wedge \text{Send}(x_f,y_m)\}) \wedge (\text{Un} \rightarrow \text{Un})$

Cases #3 & #4: dishonest verifier

T_i = {x_f : Private | OkTPM(x_f)}

T_{daa} = y_{vki} : VK<T_i> * y_m : Un * x_f : T_i * x_{cert} : {x:Un | Send(x_f,y_m)}

k_{daa} : Seal<T_{daa}∨Un>

zk-verify$_{daa}$ =

λz:Un. λy_{vki}′ : VK<T_i>; Un.		y_{vki}′ : Un
case w = (snd k_{daa}) z : T_{daa}∨Un **of**	w : Un (#3)	w : T_{daa} (#4)
let (y_{vki}, y_m, x_f, x_{cert}) = w **in**	x_f : Un (#3)	x_f : T_i (#4)
if y_{vki} = y_{vki}′ **as** y_{vki}″ **then**		y_{vki}″ : Un ∧ ...
if x_f = verify<T_i> y_{vki}″ x_{cert} x_f **then** (y_m)		y_m : Un
else failwith "statement not valid"		
else failwith "y_{vki} does not match"		

: Un→((y_{vki}:VK<T_i>→{y_m:Un|∃x_f, x_{cert}. OkTPM(x_f)∧Send(x_f,y_m)})∧(Un→Un)

Cases #3 & #4: dishonest verifier

$T_i = \{x_f : \mathrm{Private} \mid \mathrm{OkTPM}(x_f)\}$

$T_{daa} = y_{vki} : \mathrm{VK}\langle T_i\rangle * y_m : \mathrm{Un} * x_f : T_i * x_{cert} : \{x{:}\mathrm{Un} \mid \mathrm{Send}(x_f, y_m)\}$

$k_{daa} : \mathrm{Seal}\langle T_{daa} \vee \mathrm{Un}\rangle$

> not sufficient that $\mathrm{verify}\langle\alpha\rangle : \mathrm{VK}\langle\alpha\rangle \rightarrow \ldots$
>
> in our library we actually have that
> $\mathrm{verify}\langle\alpha\rangle : (\mathrm{VK}\langle\alpha\rangle \rightarrow \ldots) \wedge \mathrm{Un} \rightarrow \mathrm{Un} \rightarrow \ldots \rightarrow \mathrm{Un}$

zk-verify$_{daa}$ =

$\lambda z{:}\mathrm{Un}.\ \lambda y_{vki}' : \mathrm{VK}\langle T_i\rangle;\ \mathrm{Un}.$		$y_{vki}' : \mathrm{Un}$
case $w = (\mathrm{snd}\ k_{daa})\ z : T_{daa} \vee \mathrm{Un}$ **of**	$w : \mathrm{Un}$ (#3)	$w : T_{daa}$ (#4)
let $(y_{vki}, y_m, x_f, x_{cert}) = w$ **in**	$x_f : \mathrm{Un}$ (#3)	$x_f : T_i$ (#4)
if $y_{vki} = y_{vki}'$ **as** y_{vki}'' **then**		$y_{vki}'' : \mathrm{Un} \wedge \ldots$
if $x_f = \mathrm{verify}\langle T_i\rangle\ y_{vki}''\ x_{cert}\ x_f$ **then** (y_m)		$y_m : \mathrm{Un}$
else failwith "statement not valid"		
else failwith "y_{vki} does not match"		

$: \mathrm{Un} \rightarrow ((y_{vki}{:}\mathrm{VK}\langle T_i\rangle \rightarrow \{y_m{:}\mathrm{Un} \mid \exists x_f, x_{cert}.\ \mathrm{OkTPM}(x_f) \wedge \mathrm{Send}(x_f, y_m)\}) \wedge (\mathrm{Un} \rightarrow \mathrm{Un})$

Disjointness of types

Disjointness of types

- Definition: T_1 and T_2 are *disjoint* if
  $E \vdash M : T_1$ and $E \vdash M : T_2$ implies $E \vdash \text{false}$

Disjointness of types

- Definition: T_1 and T_2 are *disjoint* if
  $E \vdash M : T_1$ and $E \vdash M : T_2$ implies $E \vdash$ false
- How to encode a type disjoint from Un?
  (hard since Un <:> Un→Un <:> Un*Un <:> ...)

Disjointness of types

- Definition: T_1 and T_2 are *disjoint* if
  $E \vdash M : T_1$ and $E \vdash M : T_2$ implies $E \vdash$ false
- How to encode a type disjoint from Un?
  (hard since Un <:> Un→Un <:> Un*Un <:> ...)
  - Private = {f : {false}→Un | ∃x. f = λy. assert false; x}

Disjointness of types

- Definition: T_1 and T_2 are *disjoint* if
  $E \vdash M : T_1$ and $E \vdash M : T_2$ implies $E \vdash$ false

- How to encode a type disjoint from Un?
  (hard since Un <:> Un→Un <:> Un*Un <:> ...)

  - Private = {f : {false}→Un | ∃x. f = λy. assert false; x}

- We lift this to more complex types
  tree<α>=μβ. α + (α * β * β)
  tree<Private> disjoint from tree<Un>

$$\dfrac{\dfrac{\text{Private disjoint Un} \qquad \dfrac{\text{Private disjoint Un}}{\text{(Private * tree<Private> * tree<Private>) disjoint (Un * tree<Un> * tree<Un>)}}}{\text{Private + (Private * tree<Private> * tree<Private>) disjoint Un + (Un * tree<Un> * tree<Un>)}}}{\mu\beta.\ \text{Private + (Private} * \beta * \beta\text{) disjoint } \mu\beta.\ \text{Un + (Un} * \beta * \beta)}$$

Soundness

Calculus

- Surface calculus ($\mathrm{RCF}^{\forall}_{\wedge\vee}$)
  - Church-style (intrinsically typed)
  - informal (alpha-renaming convention)
  - named → human-readable
  - used by our type-checker, in the paper, on slides, etc.
  - operational semantics only by erasure into Formal-$\mathrm{RCF}^{\forall}_{\wedge\vee}$

Calculus x 2

- Surface calculus ($RCF^{\forall}_{\wedge\vee}$)
  - Church-style (intrinsically typed)
  - informal (alpha-renaming convention)
  - named → human-readable
  - used by our type-checker, in the paper, on slides, etc.
  - operational semantics only by erasure into Formal-$RCF^{\forall}_{\wedge\vee}$
- Formal calculus (Formal-$RCF^{\forall}_{\wedge\vee}$)
  - Curry-style (extrinsically typed like original RCF, very similar semantics)
  - formalized using Coq proof assistant
  - locally nameless representation (de Bruijn for bound variables)
  - machine-checked soundness proof (well-typed programs are robustly safe)

Calculus x 2

- Surface calculus ($RCF^{\forall}_{\wedge\vee}$)
  - Church-style (intrinsically typed)
  - informal (alpha-renaming convention)
  - named → human-readable
  - used by our type-checker, in the paper, on slides, etc.
  - operational semantics only by erasure into Formal-$RCF^{\forall}_{\wedge\vee}$
- Formal calculus (Formal-$RCF^{\forall}_{\wedge\vee}$)
  - Curry-style (extrinsically typed like original RCF, very similar semantics)
  - formalized using Coq proof assistant
  - locally nameless representation (de Bruijn for bound variables)
  - machine-checked soundness proof (well-typed programs are robustly safe)

\+ Adequacy: well-typed in $RCF^{\forall}_{\wedge\vee} \Rightarrow$ erasure well-typed in Formal-$RCF^{\forall}_{\wedge\vee}$

RCF$^{\forall}_{\wedge\vee}$: intersection introduction

- Because of type annotations following rule not enough

$$\frac{E \vdash M : T_1 \quad E \vdash M : T_2}{E \vdash M : T_1 \wedge T_2}$$

e.g (Private→Private)∧(Un→Un)

RCF$^{\forall}_{\wedge\vee}$: intersection introduction

- Because of type annotations following rule not enough

$$\frac{E \vdash M : T_1 \quad E \vdash M : T_2}{E \vdash M : T_1 \wedge T_2}$$

e.g (Private→Private)∧(Un→Un)

- $\lambda x{:}T_1;\ T_2.\ M$ [Reynolds ’86, ’96]
  - (λx:Private;Un. x) : (Private→Private)∧(Un→Un)
  - can’t write terms of type $(T_1 \to T_1 \to U_1) \wedge (T_2 \to T_2 \to U_2)$
    - you can use uncurried version $(T_1 \times T_1 \to U_1) \wedge (T_2 \times T_2 \to U_2)$ but then no partial application

$RCF^{\forall}_{\wedge\vee}$: intersection introduction

- Because of type annotations following rule not enough

$$\frac{E \vdash M : T_1 \quad E \vdash M : T_2}{E \vdash M : T_1 \wedge T_2}$$

e.g $(Private \to Private) \wedge (Un \to Un)$

- $\lambda x{:}T_1;\ T_2.\ M$ [Reynolds '86, '96]
  - $(\lambda x{:}Private;Un.\ x)\ :\ (Private \to Private) \wedge (Un \to Un)$
  - can't write terms of type $(T_1 \to T_1 \to U_1) \wedge (T_2 \to T_2 \to U_2)$
    - you can use uncurried version $(T_1 \times T_1 \to U_1) \wedge (T_2 \times T_2 \to U_2)$ but then no partial application
- Type alternation: for α in T; U do M [Pierce, MSCS '97]
  - More general ($\lambda x{:}T_1;\ T_2.\ M$ = for α in T_1; T_2 do $\lambda x{:}\alpha.\ M$)
  - for α in T_1;T_2 do $\lambda x{:}\alpha.\lambda x{:}\alpha.M : (T_1 \to T_1 \to U_1) \wedge (T_2 \to T_2 \to U_2)$

Erasure crucial for soundness

- polymorphism, intersections, unions vs. side-effects (known)

Erasure crucial for soundness

- polymorphism, intersections, unions vs. side-effects (known)
- Type refinements

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x{:}T \mid C\}}$$

Type alternation

$$\frac{E \vdash M\{T_i/\alpha\} : T \quad i \in 1,2}{E \vdash \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M : T}$$

Erasure crucial for soundness

- polymorphism, intersections, unions vs. side-effects (known)
- Type refinements

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x{:}T \mid C\}}$$

Type alternation

$$\frac{E \vdash M\{T_i/\alpha\} : T \quad i \in 1,2}{E \vdash \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M : T}$$

- Type refinements vs. type alternation

$$\frac{\dfrac{\vdash M\{T_1/\alpha\}{:}T \quad \vdash M\{T_1/\alpha\}{=}M\{T_1/\alpha\}}{\vdash M\{T_1/\alpha\} : \{x{:}T \mid x{=}M\{T_1/\alpha\}\}}}{\vdash \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M : \{x{:}T \mid x{=}M\{T_1/\alpha\}\}}$$

Erasure crucial for soundness

- polymorphism, intersections, unions vs. side-effects (known)
- Type refinements — Type alternation

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x{:}T \mid C\}} \qquad \frac{E \vdash M\{T_i/\alpha\} : T \quad i \in 1,2}{E \vdash \mathrm{for}\ \alpha\ \mathrm{in}\ T_1;T_2\ \mathrm{do}\ M : T}$$

- Type refinements vs. type alternation

$$\frac{\dfrac{\vdash M\{T_1/\alpha\}:T \quad \vdash M\{T_1/\alpha\}=M\{T_1/\alpha\}}{\vdash M\{T_1/\alpha\} : \{x{:}T \mid x=M\{T_1/\alpha\}\}}}{\vdash \mathrm{for}\ \alpha\ \mathrm{in}\ T_1;T_2\ \mathrm{do}\ M : \{x{:}T \mid x=M\{T_1/\alpha\}\}}$$

  - This can only possibly work if (for α in $T_1;T_2$ do M) = $M\{T_1/\alpha\}$ (both operationally and in the authorization logic)

Erasure crucial for soundness

- polymorphism, intersections, unions vs. side-effects (known)
- Type refinements

$$\frac{E \vdash M : T \quad E \vdash C\{M/x\}}{E \vdash M : \{x{:}T \mid C\}}$$

Type alternation

$$\frac{E \vdash M\{T_i/\alpha\} : T \quad i \in 1,2}{E \vdash \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M : T}$$

- Type refinements vs. type alternation

$$\frac{\dfrac{\vdash M\{T_1/\alpha\}:T \quad \vdash M\{T_1/\alpha\}=M\{T_1/\alpha\}}{\vdash M\{T_1/\alpha\} : \{x{:}T \mid x=M\{T_1/\alpha\}\}}}{\vdash \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M : \{x{:}T \mid x=M\{T_1/\alpha\}\}}$$

  - This can only possibly work if (for α in $T_1;T_2$ do M) = $M\{T_1/\alpha\}$ (both operationally and in the authorization logic)
  - Fors and type annotations **need** to be erased away $\lfloor \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M \rfloor = \lfloor M \rfloor$

Formalization

- 14k+LOC of Coq, 6+ months of work (Coq beginner)
  - 1.5+kLOC of definitions, most generated from **Ott** spec + quite big patch [Sewell, Nardelli, Owens, Peskine, Ridge, Sarkar & Strnisa, JFP '10]
  - 12+kLOC Software-Foundations-style proofs with very little automation
  + 25kLOC of "infrastructure" lemmas generated by wonderful **LNgen** tool [Aydemir & Weirich, Draft '10]
- Reasonably complete
  - One notable exception: transitivity of subtyping - paper proof goes by induction on the size of derivations, very informal
- Found+fixed 3 relatively small bugs in previous proofs
  - Public Down / Tainted Up, Robust Safety, Strengthening (claim weakened)
- Will be open sourced, once polished

Counterintuitive inversion lemmas

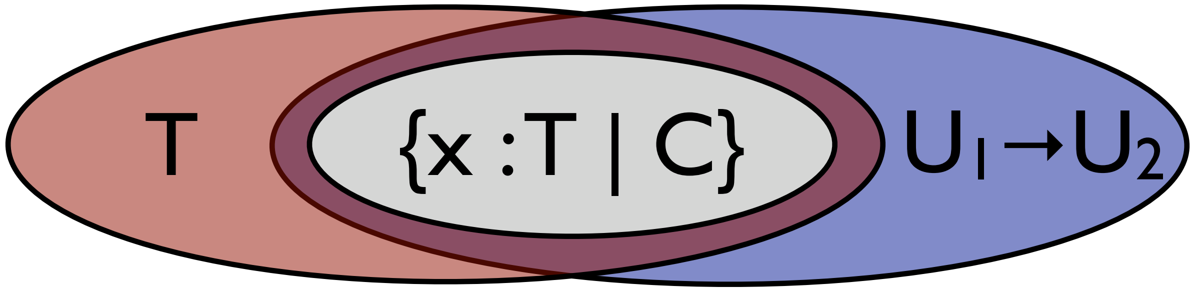
- Intuitively, types are sets of values
  - $\{x : T \mid C\}$ intuitively contains the values of T that satisfy C
  - $T_1 \wedge T_2$ intuitively contains the values that are in T_1 and in T_2
  - intuitively subtyping is (somehow related with) set inclusion
  - but with syntactic subtyping this intuition is dead wrong(!)

Counterintuitive inversion lemmas

- Intuitively, types are sets of values
  - $\{x : T \mid C\}$ intuitively contains the values of T that satisfy C
  - $T_1 \wedge T_2$ intuitively contains the values that are in T_1 and in T_2
  - intuitively subtyping is (somehow related with) set inclusion
  - but with syntactic subtyping this intuition is dead wrong(!)
- Lemma: If $E \vdash \{x : T \mid C\} <: U_1 \rightarrow U_2$ then $E \vdash T <: U_1 \rightarrow U_2$

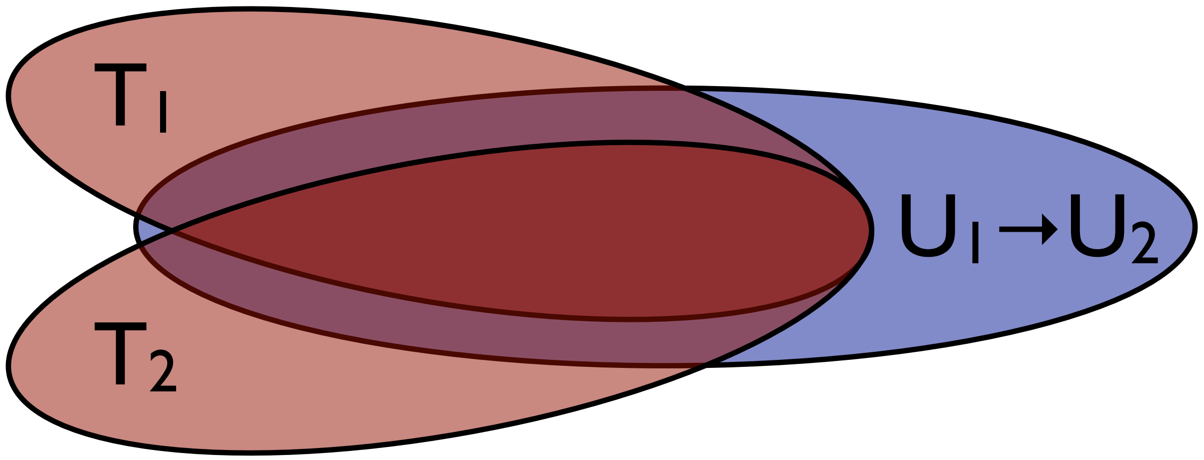
Counterintuitive inversion lemmas

- Intuitively, types are sets of values
  - $\{x : T \mid C\}$ intuitively contains the values of T that satisfy C
  - $T_1 \wedge T_2$ intuitively contains the values that are in T_1 and in T_2
  - intuitively subtyping is (somehow related with) set inclusion
  - but with syntactic subtyping this intuition is dead wrong(!)
- Lemma: If $E \vdash \{x : T \mid C\} <: U_1 \to U_2$ then $E \vdash T <: U_1 \to U_2$



Counterintuitive inversion lemmas

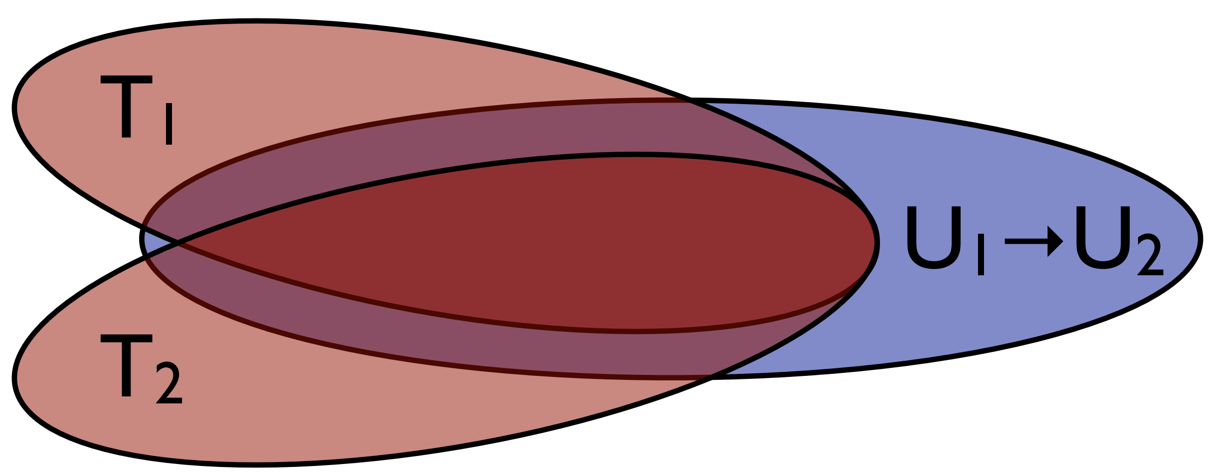
- Intuitively, types are sets of values
  - $\{x : T \mid C\}$ intuitively contains the values of T that satisfy C
  - $T_1 \wedge T_2$ intuitively contains the values that are in T_1 and in T_2
  - intuitively subtyping is (somehow related with) set inclusion
  - but with syntactic subtyping this intuition is dead wrong(!)
- Lemma: If $E \vdash \{x : T \mid C\} <: U_1 \to U_2$ then $E \vdash T <: U_1 \to U_2$
- Lemma: If $E \vdash T_1 \wedge T_2 <: U_1 \to U_2$ then
  $E \vdash T_1 <: U_1 \to U_2$ or $E \vdash T_2 <: U_1 \to U_2$



Counterintuitive inversion lemmas

- Intuitively, types are sets of values
  - $\{x : T \mid C\}$ intuitively contains the values of T that satisfy C
  - $T_1 \wedge T_2$ intuitively contains the values that are in T_1 and in T_2
  - intuitively subtyping is ...) set inclusion
  - but with syntactic ... wrong(!)
- Lemma: If $E \vdash \{x : T$ ... $T <: U_1 \rightarrow U_2$
- Lemma: If $E \vdash T_1 \wedge T_2 <: U_1 \rightarrow U_2$ then $E \vdash T_1 <: U_1 \rightarrow U_2$ or $E \vdash T_2 <: U_1 \rightarrow U_2$

Calling them intersection types is just deceiving! How about GLB types?



Counterintuitive inversion lemmas

- Intuitively, types are sets of values
  - $\{x : T \mid C\}$ intuitively contains the values of T that satisfy C
  - $T_1 \wedge T_2$ intuitively contains the values that are in T_1 and in T_2
  - intuitively subtyping is (somehow related with) set inclusion
  - but with syntactic subtyping this intuition is dead wrong(!)
- Lemma: If $E \vdash \{x : T \mid C\} <: U_1 \to U_2$ then $E \vdash T <: U_1 \to U_2$
- Lemma: If $E \vdash T_1 \wedge T_2 <: U_1 \to U_2$ then
  $E \vdash T_1 <: U_1 \to U_2$ or $E \vdash T_2 <: U_1 \to U_2$
- Still, such inversion lemmas are **crucial** to our proofs

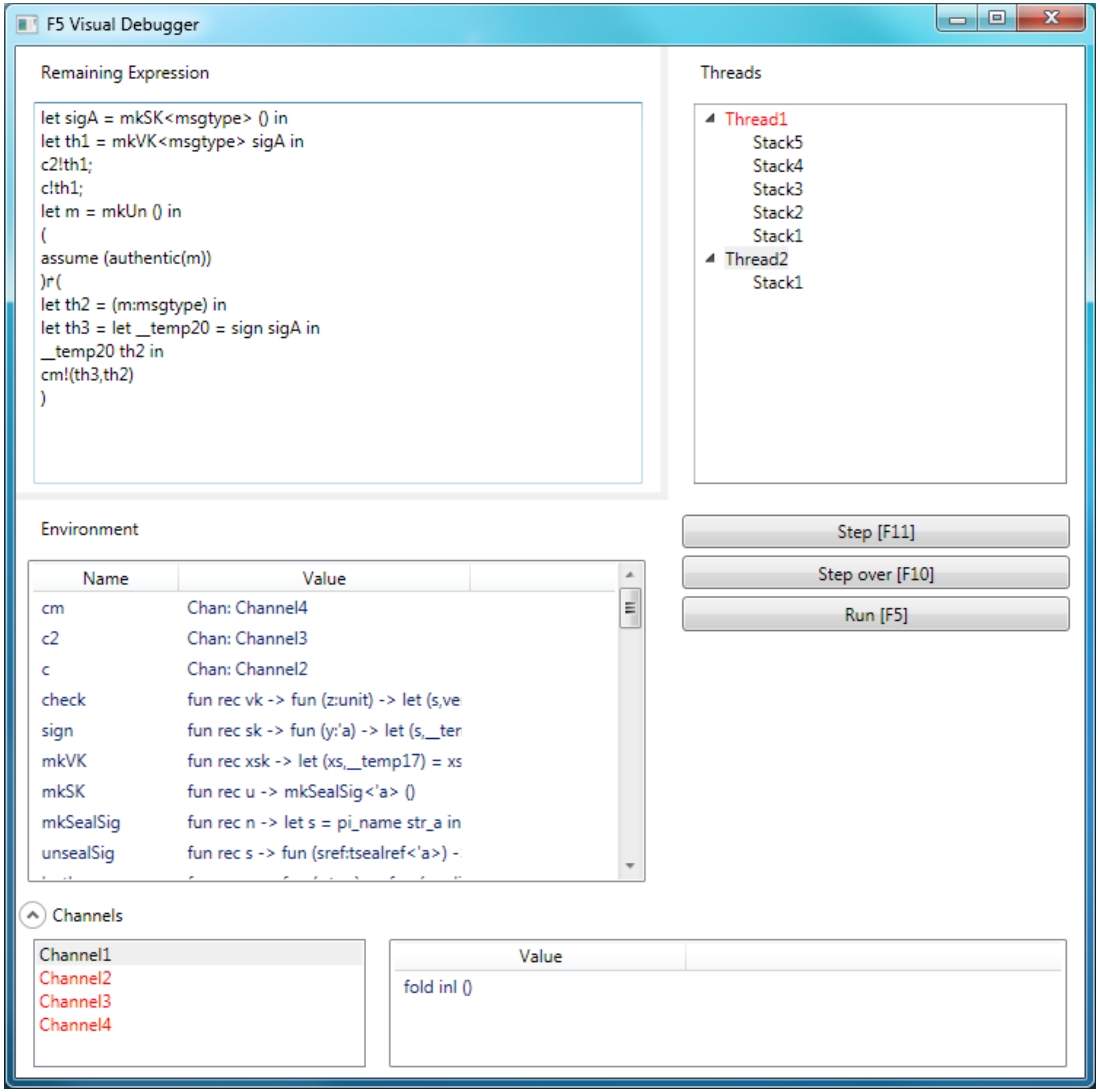
Implementation

(Kudos to Thorsten Tarrach)

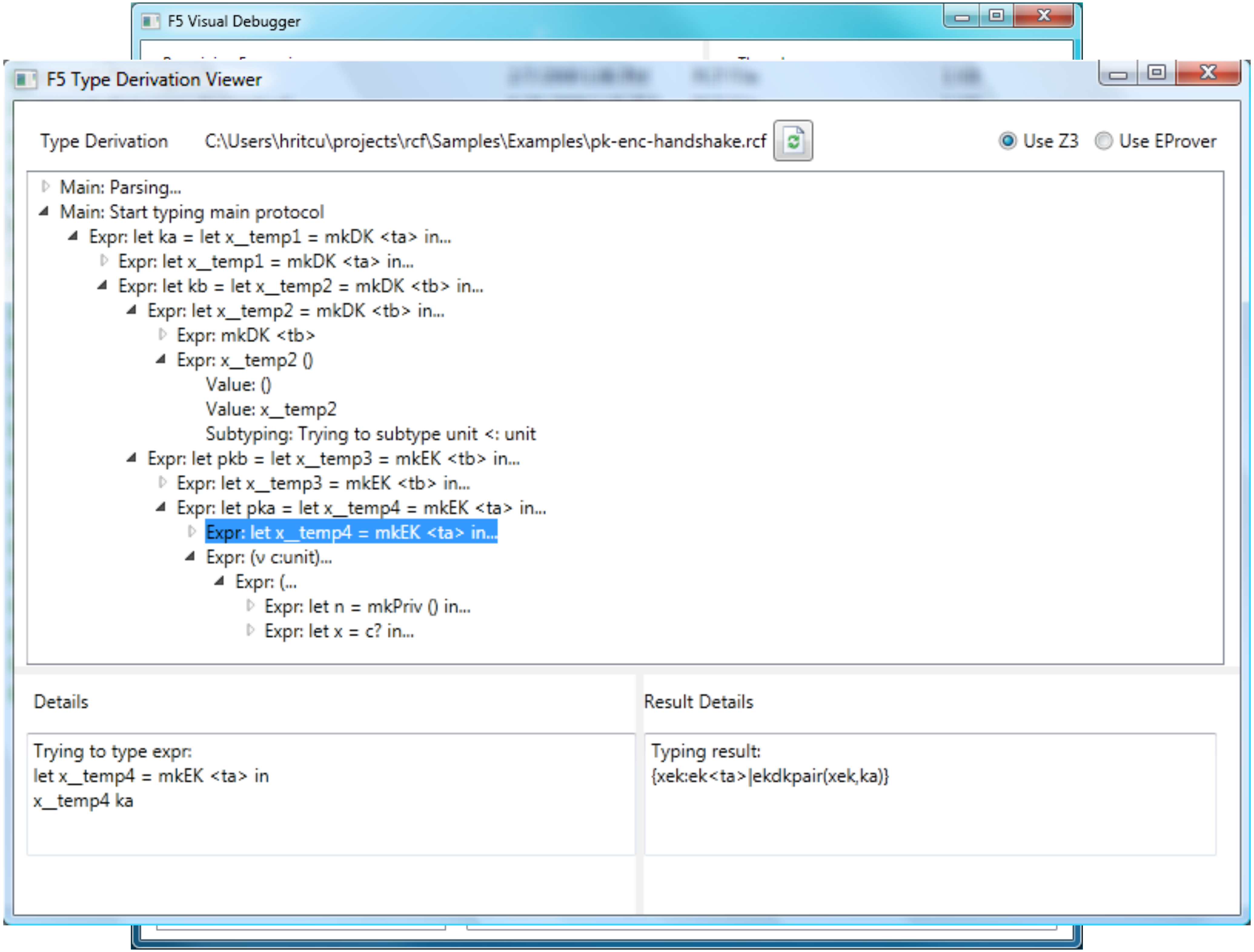
F5: tool-chain for RCF$^{\forall}_{\wedge\vee}$

- Type-checker for RCF$^{\forall}_{\wedge\vee}$
  - Extended syntax: simple modules, ADTs, recursive functions, typedefs, mutable references (all encoded into RCF$^{\forall}_{\wedge\vee}$)
  - Very limited type inference: some polymorphic instantiations
  - (Partial) type derivation can be inspected in visualizer
  - Can use SMT solvers (Z3) or FOL provers (eprover)
  - Efficient (especially with Z3)
    type-checks 1500+LOC in ~12 seconds on normal laptop
- Automatic code generator for zero-knowledge
- Interpreter + visual debugger
- ~5000LOC, first release coming soon (open source)

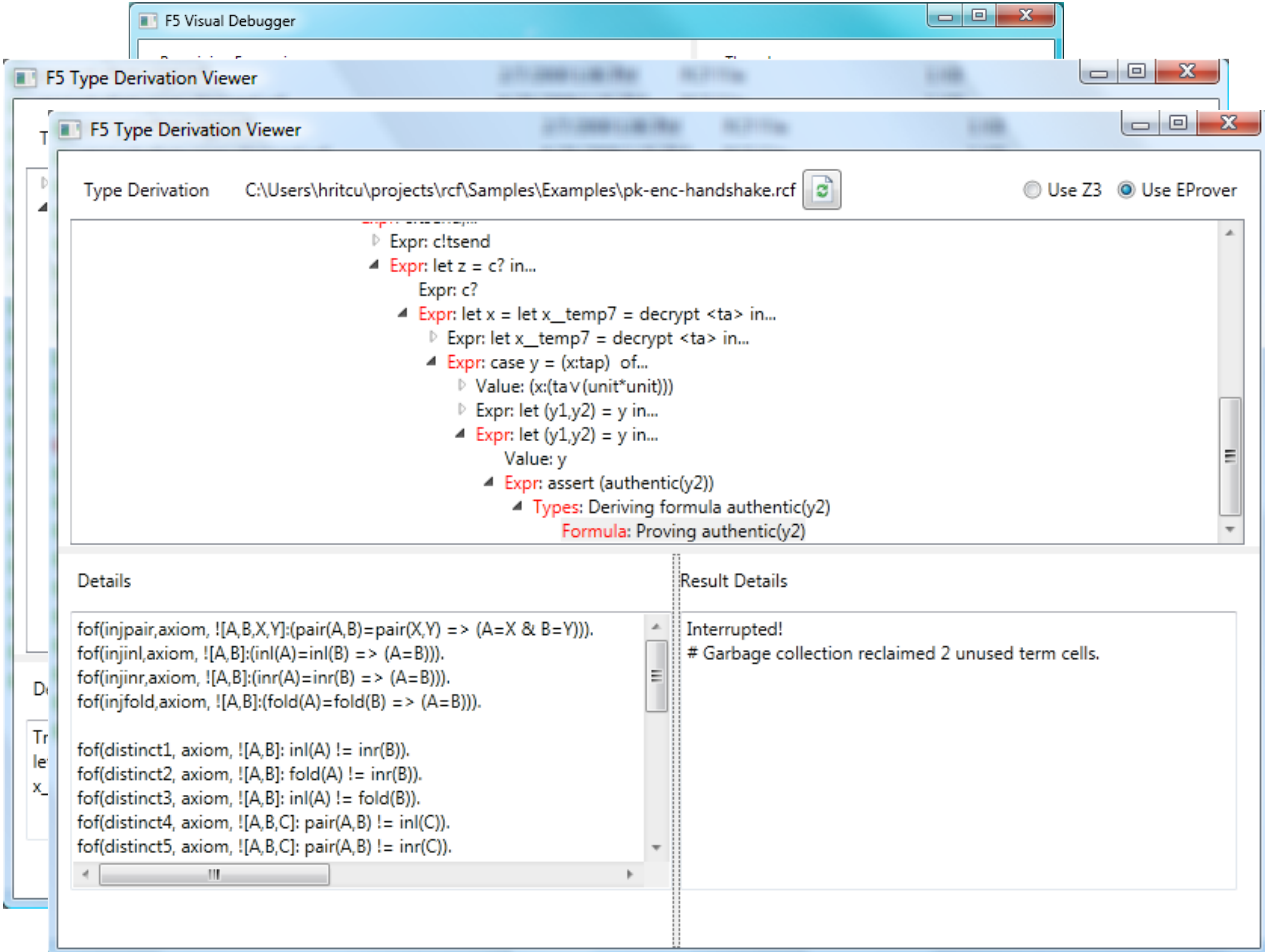
Screenshots



Screenshots



Screenshots



Random thoughts for the future

- Bigger case studies (already started with Civitas)
- Study type inference, maybe in restricted setting
  - Our type-checker is efficient for a good reason
- Study relation to F7v2
- Semantic subtyping for RCF ... is it possible? λ + {x:T|C}
- Develop semantic model for RCF / $RCF^{\forall}_{\wedge\vee}$
- Study methods for establishing observational equivalence in RCF / $RCF^{\forall}_{\wedge\vee}$ (logical relations, bisimulations, etc.)
- Automatically generate zero-knowledge proof system corresponding to abstract statement specification (concrete crypto -- efficiency big challenge)

Thank you!

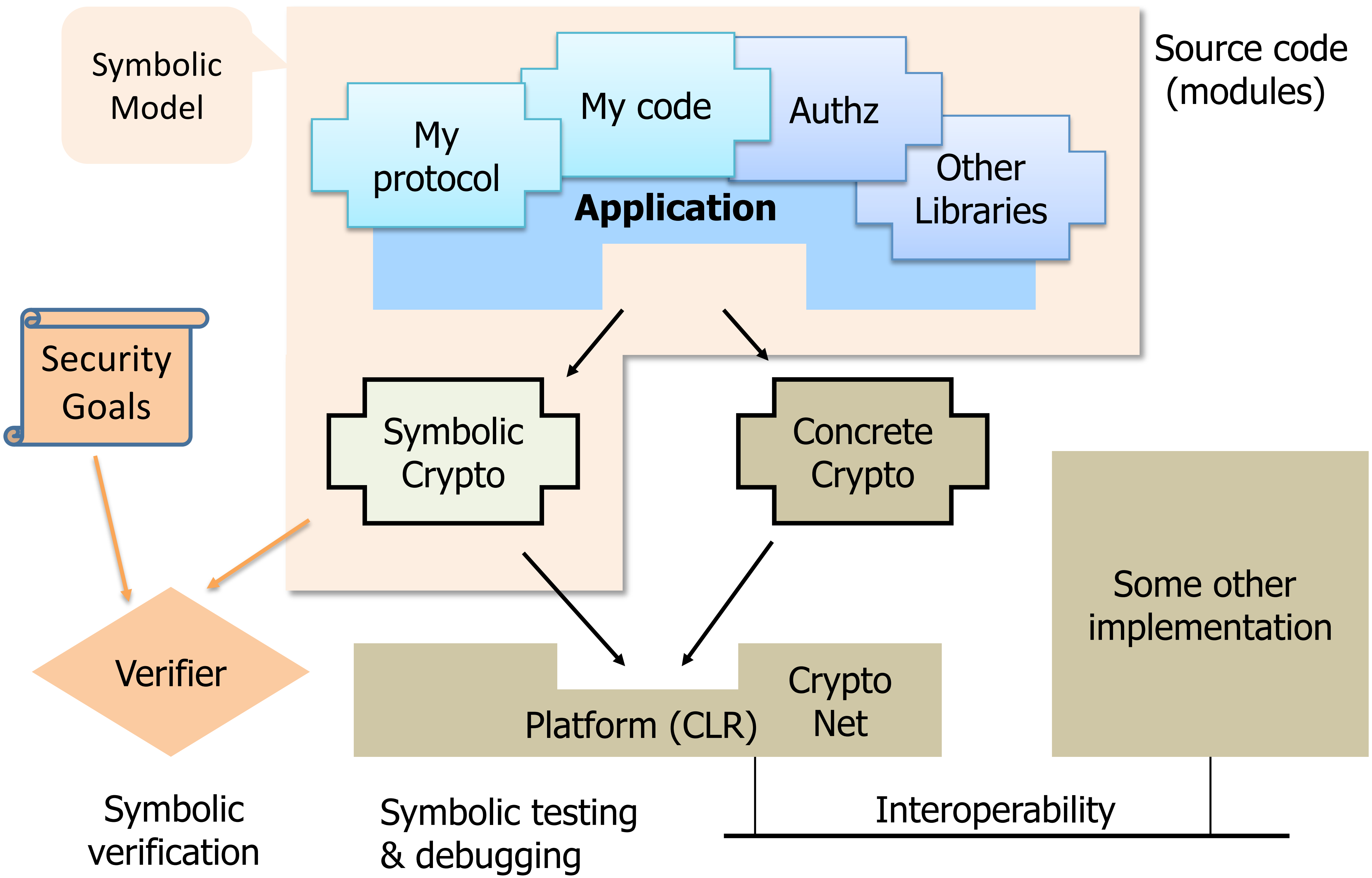
Analyzing protocols

- Analyzing protocol **models**: successful research field
  - **modelling languages:**
    strand spaces, CSP, spi calculus, applied-π, PCL, etc.
  - **security properties:**
    from secrecy & authenticity all the way to coercion-resistance
  - **automated analysis tools:**
    Casper, AVISPA, ProVerif, Cryptyc & other type-checkers, etc.
  - **found bugs in deployed protocols**
    SSL, PKCS, Microsoft Passport, Kerberos, Plutus, etc.
  - **proved industrial protocols secure**
    EKE, JFK, TLS, DAA, etc.

Abstract models vs. actual code

- Still, only limited impact in practice!
- Researchers prove properties of abstract models
- Developers write and execute actual code
- Usually no relation between the two
  - Even if correspondence was proved, model and code will drift apart as the code evolves
- Most often the only "model" is the code itself
  - **The good news:** when given a proper semantics the security of code can be analyzed as well

F7 (& fs2pv) tool-chain



Case studies (work in progress)

1. A new implementation of the complete DAA protocol
2. Automatically generated implementations of automatically strengthened protocols
   - "Achieving security despite compromise using zero-knowledge" [Backes, Grochulla, Hritcu & Maffei, CSF '09]
3. Civitas electronic voting system [Clarkson, Chong & Myers, SSP '08]
   - Work in progress (Matteo Maffei & Fabienne Eigner)
   - Other complex primitives: distributed encryption with re-encryption and plaintext equivalence testing (PET)