Union, Intersection, and Refinement Types and Reasoning about Type Disjointness for Analyzing Protocol Implementations

Cătălin Hrițcu

Joint work with: Michael Backes and Matteo Maffei

A little bit of background

Analyzing cryptographic 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 is 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

Analyzing protocol implementations

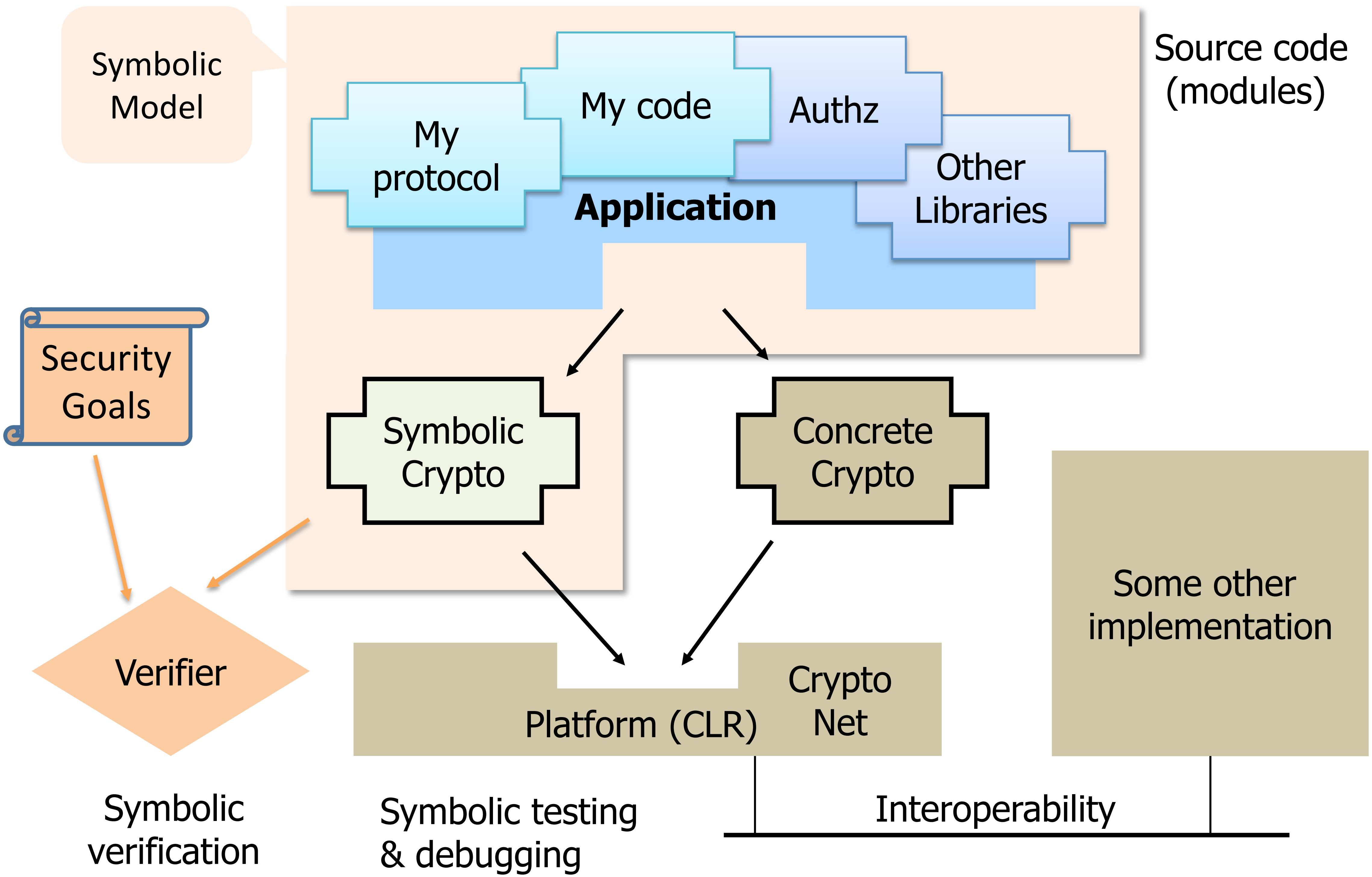
- Recently many approaches proposed
  - **program verification:**
    CSur [Goubault-Larrecq and Parrennes, VMCAI '05]
    ASPIER model checker for C [Chaki & Datta, CSF '09]
    VCgen for C [Dupressoir, Gordon, Jürjens & Naumann, CSF '11]
  - **extracting ProVerif models:**
    fs2pv [Bhargavan, Fournet, Gordon & Tse, CSF '06]
    symbolic execution for C [Aizatulin, Gordon, Jürjens, CCS '11]
  - **typing:**
    F7v1 [Bengtson, Bhargavan, Fournet, Gordon & Maffeis, CSF '08]
    F7v2 [Bhargavan, Fournet & Gordon, POPL '10]
    F* [Swamy, Chen, Fournet, Strub, Bharagavan & Yang, ICFP '11]
    - 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)

F7 (& fs2pv) tool-chain



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

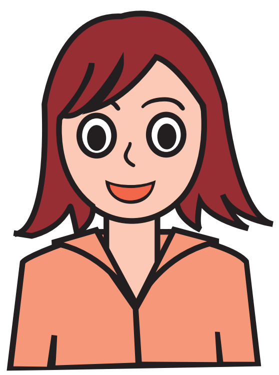
Security properties (informal)

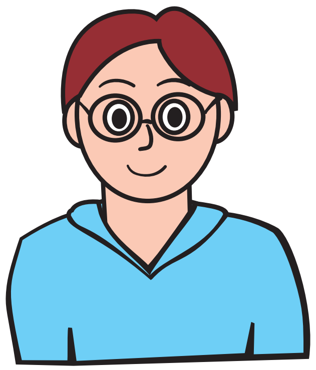
- **Safety:** in all executions all asserts succeed
  (i.e. asserts are logically entailed by the active assumes)
- **Robust safety:**
  safety in the presence of arbitrary DY attacker

  - 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

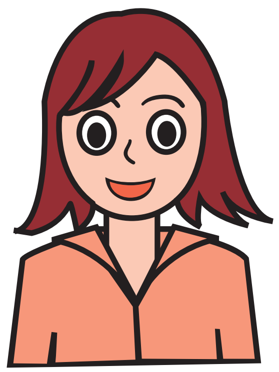
Why wasn’t this enough?

An extremely simple example



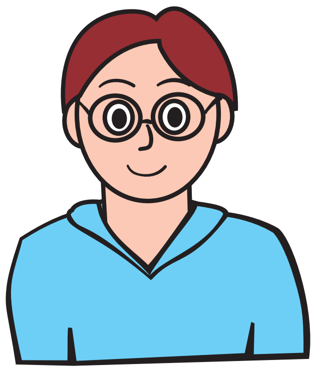


An extremely simple example



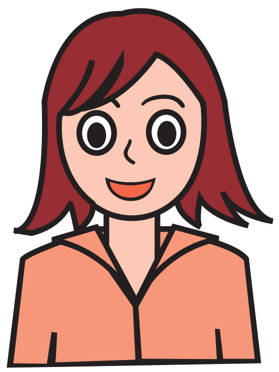
n : Private

public key
pk_B : PK<Private>



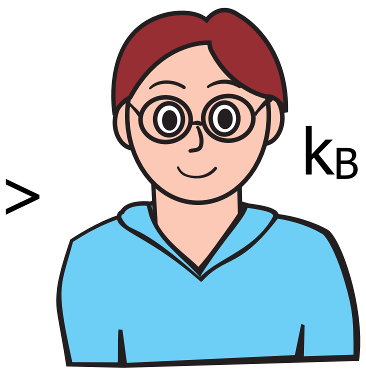
enc<Private> pk_B n

An extremely simple example



n : Private

public key
pk_B : PK<Private>

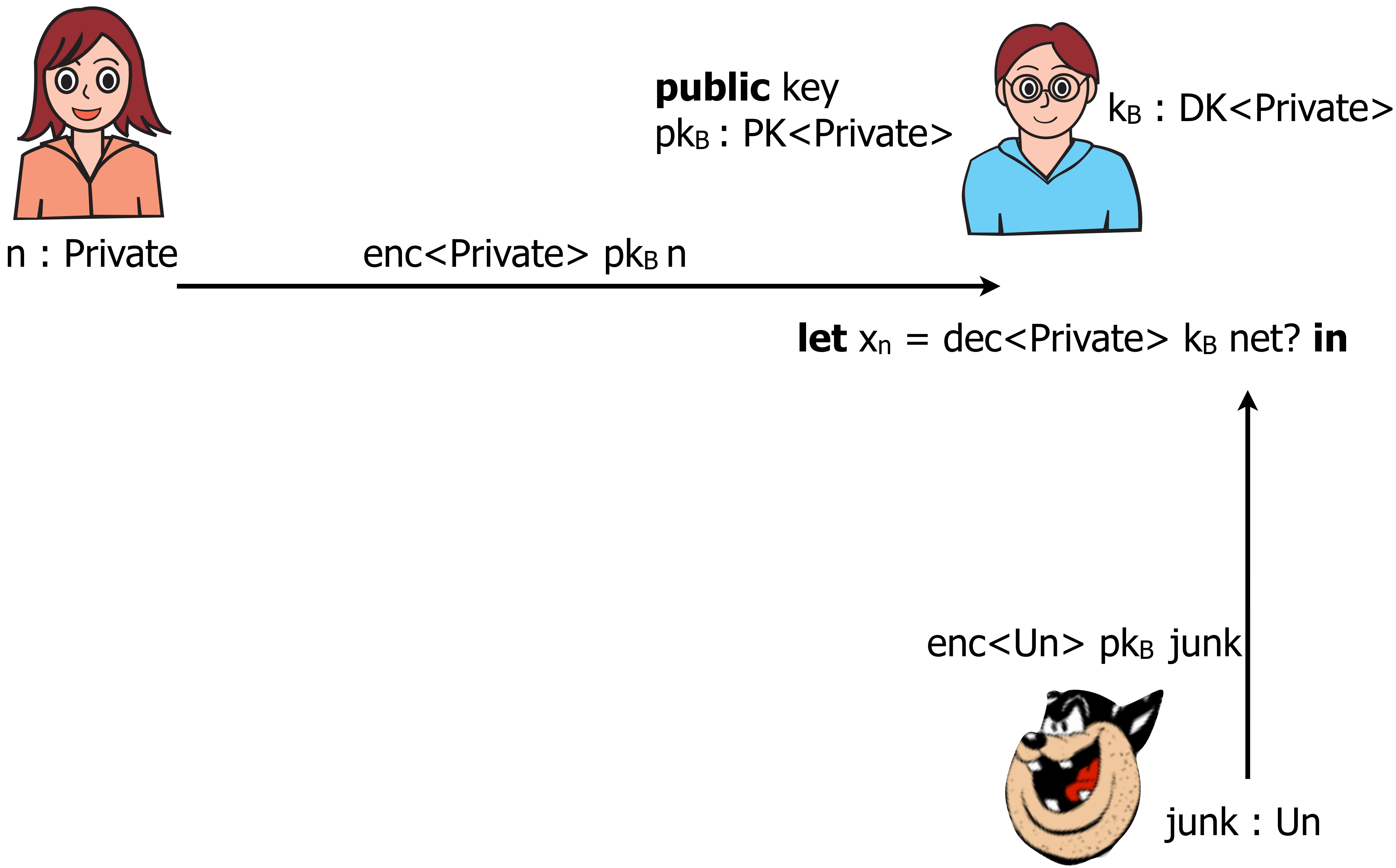


k_B : DK<Private>

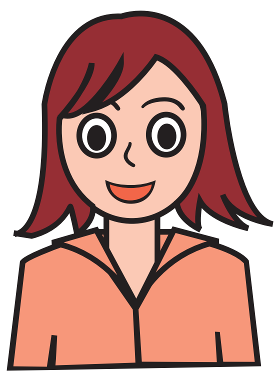
enc<Private> pk_B n

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

An extremely simple example

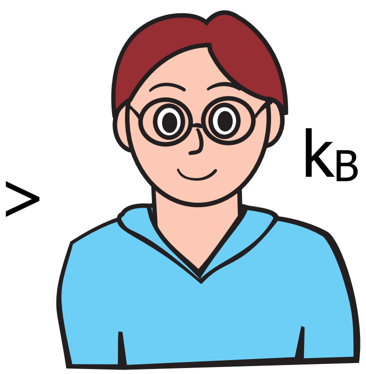


An extremely simple example



n : Private

public key
pk_B : PK<Private>



k_B : DK<Private>

enc<Private> pk_B n

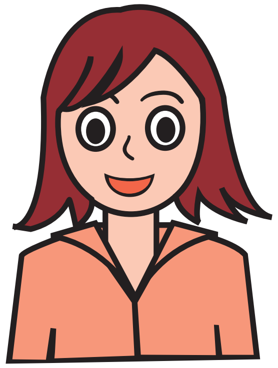
let x_n = dec<Private> k_B net? **in**

x_n : Private ∨ Un

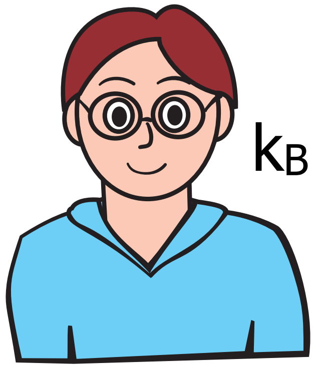
enc<Un> pk_B junk

junk : Un

An extremely simple example



n : Private



public key
pk_B : PK<Private>

k_B : DK<Private>

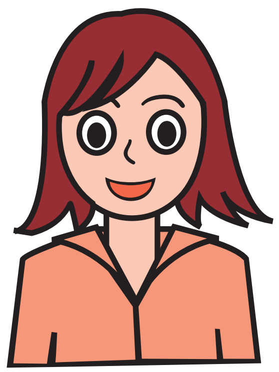
enc<Private> pk_B n

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

enc<Un> pk_B junk

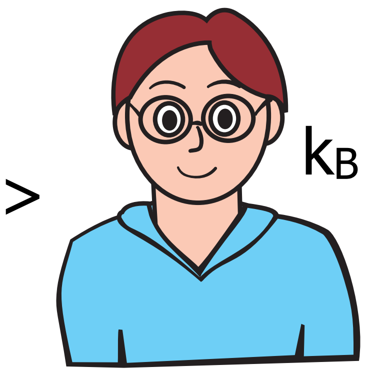
junk : Un

An extremely simple example



$pk_A : PK<T_A>$

public key
$pk_B : PK<Private>$



$k_B : DK<Private>$

$n : Private$

$enc<Private>\ pk_B\ n$

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

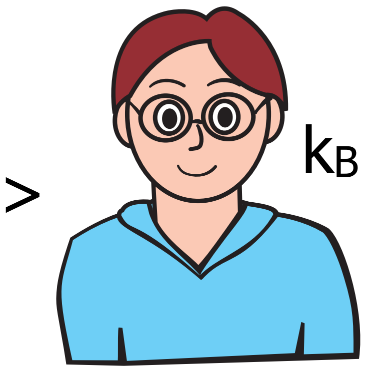
assume Auth(m,B,A)

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

An extremely simple example

$pk_A : \text{PK}<T_A>$

public key
$pk_B : \text{PK}<\text{Private}>$



$k_B : \text{DK}<\text{Private}>$

$n : \text{Private}$

$\text{enc}<\text{Private}>\ pk_B\ n$

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

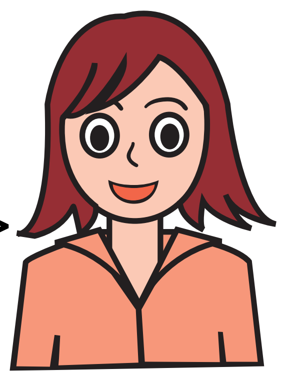
let $x_n = \text{dec}<\text{Private}>\ k_B\ \text{net?}$ **in**

assume Auth(m,B,A)

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

$x_n : \text{Private} \vee \text{Un}$

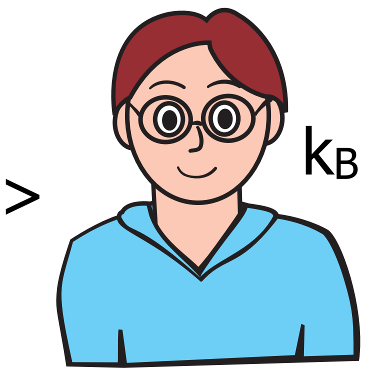
An extremely simple example



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)

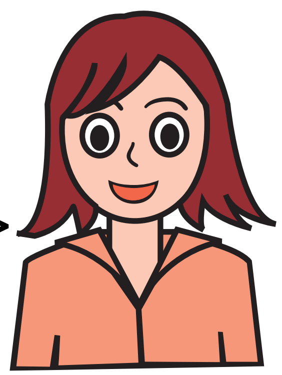
x_n : Private ∨ Un

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

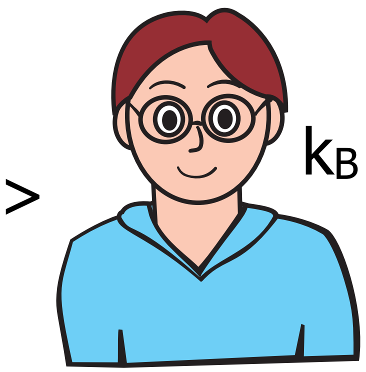
k_A : DK<T_A>



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**

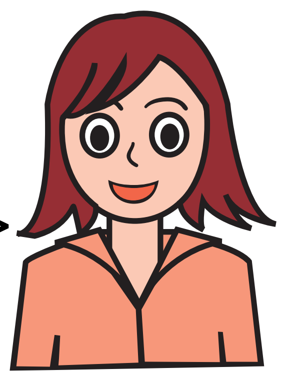
$y_n y_m$: T_A ∨ Un

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

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

enc<Un> pk_A junk

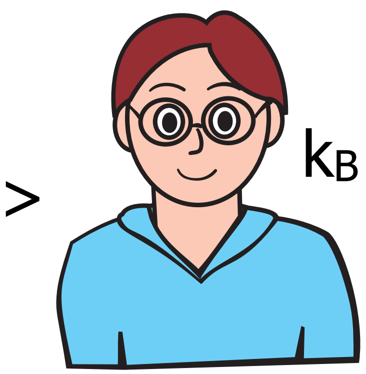
An extremely simple example



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)

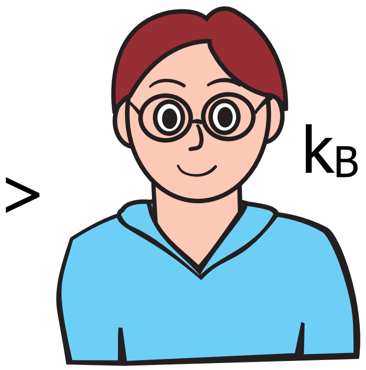
enc<Un> pk_A junk

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

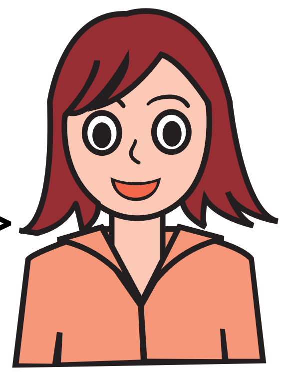
let $y_n y_m$ = dec<T_A> k_A net? **in**
case $y_n y_m'$ = $y_n y_m$: T_A ∨ Un **of**
let (y_n, y_m) = $y_n y_m'$ **in**
if y_n = n **then assert** Auth(y_m,B,A)

Honest sender case: y_m : {y_m:Un | Auth(y_m,B,A)}
assert succeeds

enc<Un> pk_A junk

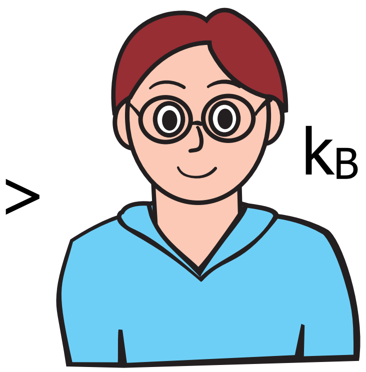
An extremely simple example

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 | 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

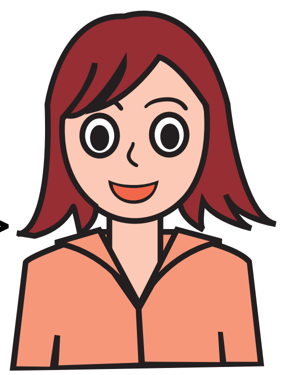
let y_ny_m = dec<T_A> k_A net? **in**
case y_ny_m' = y_ny_m : T_A $\vee$ Un **of**
let (y_n, y_m) = y_ny_m' **in**
if y_n = n **then assert** Auth(y_m,B,A)

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

enc<Un> pk_A junk

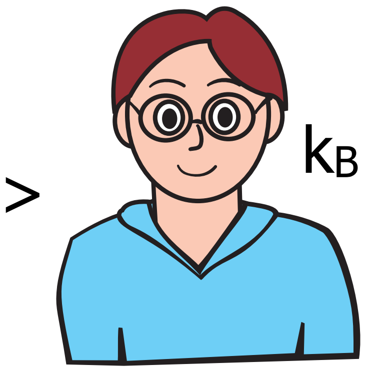
An extremely simple example

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

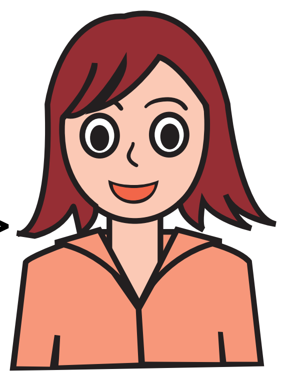
let y_ny_m = dec<T_A> k_A net? **in**
case y_ny_m′ = y_ny_m : T_A ∨ Un **of**
let (y_n, y_m) = y_ny_m′ **in**
if y_n = n **then assert** Auth(y_m,B,A)

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

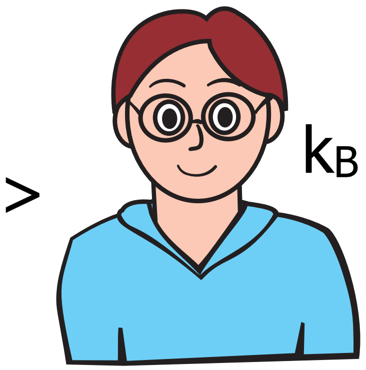
$k_A : DK<T_A>$



$pk_A : PK<T_A>$

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** $Auth(y_m,B,A)$

Dishonest sender case: $y_n : Un,\ n : Private$

$Un \cap Private = \varnothing$ **so assert won't be executed**

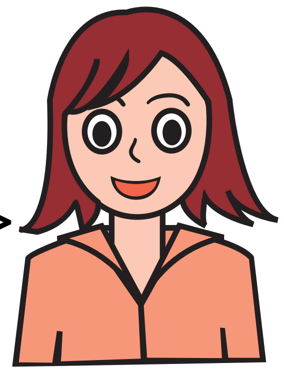
$enc<Un>\ pk_A\ junk$

F7v1 can't handle this

An extremely simple example

simplified variant of Needham-Schroeder-Lowe

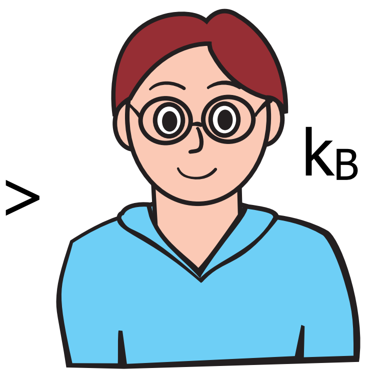
k_A : DK<T_A>



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$\vee$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 $\vee$ 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)

Dishonest sender case: y_n : Un, n : Private

Un $\cap$ Private = $\varnothing$ **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 ($\mathrm{RCF}^{\forall}_{\wedge\vee}$)
  - 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 ($\mathrm{RCF}^{\forall}_{\wedge\vee}$)
  - 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

Not today ~~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

Not today ~~3. type-checking applications based on NI-ZK (DAA, Civitas, etc.)~~

\+ Machine-checked soundness proof + ~~cool implementation~~

Encoding symbolic cryptography using dynamic sealing

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, functions and ν

    Seal<α> = (α→Un) * (Un→α)

    mkSeal : ∀α. unit → Seal<α>

Symmetric encryption

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

$\text{mkKey} = \text{mkSeal}$

$\text{senc} = \Lambda\alpha.\lambda k{:}\text{Key}\langle\alpha\rangle.\ \text{fst}\ k \qquad : \forall\alpha.\text{Key}\langle\alpha\rangle \rightarrow \alpha \rightarrow \text{Un}$

$\text{sdec} = \Lambda\alpha.\lambda k{:}\text{Key}\langle\alpha\rangle.\ \text{snd}\ k \qquad : \forall\alpha.\text{Key}\langle\alpha\rangle \rightarrow \text{Un} \rightarrow \alpha$

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

“Public”-key encryption

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

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

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

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

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

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

“Public”-key encryption

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

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

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

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

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

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

- 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 = mkSeal : ∀α.unit→DK<α>

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

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

dec = Λα.λdk:DK<α>. snd k : ∀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

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

$\mathrm{PK}\langle\alpha\rangle = \alpha\to\mathrm{Un}$

$\mathrm{mkDK} = \mathrm{mkSeal} \qquad : \forall\alpha.\mathrm{unit}\to\mathrm{DK}\langle\alpha\rangle$

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

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

$\mathrm{dec} = \Lambda\alpha.\lambda dk{:}\mathrm{DK}\langle\alpha\rangle.\ \mathrm{snd}\ k \qquad : \forall\alpha.\mathrm{DK}\langle\alpha\rangle\to\mathrm{Un}\to\alpha$

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

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

Public-key encryption - FIXED

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

PK<α> = (α∨Un)→Un

mkDK = mkSeal : ∀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>. snd k : ∀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

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

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

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

$mkDK = mkSeal$

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

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

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

- 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

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

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

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

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

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

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

- 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

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

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

$mkDK = mkSeal$: $\forall($

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

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

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

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

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

PK<α> = (α∨Un)→Un

mkDK = mkSeal : ∀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>. snd k : ∀α.DK<α>→Un→(α∨Un)

normal!

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

Digital signatures

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

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

$mkSK = mkSeal$

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

$sign = \Lambda\alpha.\lambda sk{:}SK\langle\alpha\rangle.\ fst\ sk \quad : \forall\alpha.SK\langle\alpha\rangle\rightarrow\alpha\rightarrow Un$

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

if m = vk n **then** vk n

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 = mkSeal

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

sign = Λα.λsk:SK<α>. fst sk : ∀α.SK<α>→α→Un

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

$SK<\alpha> = (\alpha \to Un) * VK<\alpha>$

$VK<\alpha> = Un \to ((Any \to \alpha) \wedge (Un \to Un))$

mkSK = ...

: $\forall\alpha.unit \to SK<\alpha>$

$mkVK = \Lambda\alpha.\lambda sk{:}SK<\alpha>.\ snd\ sk$: $\forall\alpha.SK<\alpha> \to VK<\alpha>$

$sign = \Lambda\alpha.\lambda sk{:}SK<\alpha>.\ fst\ sk$: $\forall\alpha.SK<\alpha> \to \alpha \to Un$

$verify = \Lambda\alpha.\lambda vk{:}VK<\alpha>.\ vk$: $\forall\alpha.VK<\alpha> \to Un \to Any \to \alpha$

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 = ...

Verification keys are always public

$T \wedge Un$ is always public since $T \wedge Un <: Un$

$: \forall\alpha.unit \to SK\langle\alpha\rangle$

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

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

$verify = \Lambda\alpha.\lambda vk{:}VK\langle\alpha\rangle.\ vk \quad : \forall\alpha.VK\langle\alpha\rangle \to Un \to Any \to \alpha$

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

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

sign = Λα.λsk:SK<α>. fst sk : ∀α.SK<α>→α→Un

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

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

else failwith "bad signature"

in (s, vk) : ∀α.unit→SK<α>

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

sign = Λα.λsk:SK<α>. fst sk : ∀α.SK<α>→α→Un

verify = Λα.λvk:VK<α>. vk : ∀α.VK<α>→Un→Any→α

Digital signatures - FIXED

SK<α> = (α→Un) * VK<α>

VK<α> = Un→((Any→α)∧(Un→Un))

Introduces intersection of 2 function types

mkSK = Λα.λ_:unit. **let** (s,u) = mkSeal () in

let vk = λn:Un. λm:Any ; Un.

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

n "bad signature"

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

: ∀α.unit→SK<α>

mkVK = Λ ... sk : ∀α.SK<α>→VK<α>

sign = Λα.λsk:SK<α>. fst sk : ∀α.SK<α>→α→Un

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

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

n “bad

Introduces intersection of 2 function types

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

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

mkVK = Λα. … sk :

sign = Λα.λsk:SK<α>. fst sk : ∀α.SK<α>→α→Un

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

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

Disjointness of types

Disjointness of types

- Definition: T_1 and T_2 are *disjoint* ($T_1 \# T_2$) if $E \vdash v : T_1$ and $E \vdash v : T_2$ implies $E \vdash$ false

Disjointness of types

- Definition: T_1 and T_2 are *disjoint* ($T_1 \# T_2$) if $E \vdash v : T_1$ and $E \vdash v : T_2$ 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* ($T_1 \# T_2$) if $E \vdash v : T_1$ and $E \vdash v : T_2$ implies $E \vdash \text{false}$
- How to encode a type disjoint from Un? (hard since Un <:> Un→Un <:> Un*Un <:> ...)
  - Private = $\{f : \{\text{false}\} \rightarrow \text{Un} \mid \exists x.\ f = \lambda y.\ \text{assert false};\ x\}$

Disjointness of types

- Definition: T_1 and T_2 are *disjoint* ($T_1 \# T_2$) if
  $E \vdash v : T_1$ and $E \vdash v : T_2$ implies $E \vdash$ false

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

  - Private = $\{f : \{\mathrm{false}\} \to \mathrm{Un} \mid \exists x.\ f = \lambda y.\ \mathrm{assert\ false};\ x\}$

- We lift this to more complex types
  $\mathrm{tree}\langle\alpha\rangle = \mu\beta.\ \alpha + (\alpha * \beta * \beta)$
  $\mathrm{tree}\langle\mathrm{Private}\rangle \ \#\ \mathrm{Un}$

$$\dfrac{\dfrac{\dfrac{\mathrm{Private}\ \#\ \mathrm{Un} \qquad \dfrac{\mathrm{Private}\ \#\ \mathrm{Un}}{(\mathrm{Private} * \mathrm{tree}\langle\mathrm{Private}\rangle * \mathrm{tree}\langle\mathrm{Private}\rangle)\ \#\ (\mathrm{Un} * \mathrm{tree}\langle\mathrm{Un}\rangle * \mathrm{tree}\langle\mathrm{Un}\rangle)}}{\mathrm{Private} + (\mathrm{Private} * \mathrm{tree}\langle\mathrm{Private}\rangle * \mathrm{tree}\langle\mathrm{Private}\rangle)\ \#\ \mathrm{Un} + (\mathrm{Un} * \mathrm{tree}\langle\mathrm{Un}\rangle * \mathrm{tree}\langle\mathrm{Un}\rangle)}}{\mu\beta.\ \mathrm{Private} + (\mathrm{Private} * \beta * \beta)\ \#\ \mu\beta.\ \mathrm{Un} + (\mathrm{Un} * \beta * \beta)}}{\mathrm{tree}\langle\mathrm{Private}\rangle\ \#\ \mathrm{Un}}$$

Non-Disjointness Judgment

(ND Private Un)
$$\frac{fv(C) = \emptyset}{\vdash \mathsf{Private}_C \circledcirc \mathsf{Un} \leadsto C}$$

(ND True)
$$\frac{}{\vdash T_1 \circledcirc T_2 \leadsto \mathsf{true}}$$

(ND Sym)
$$\frac{\vdash T_2 \circledcirc T_1 \leadsto C}{\vdash T_1 \circledcirc T_2 \leadsto C}$$

(ND Refine)
$$\frac{\vdash T_1 \circledcirc T_2 \leadsto C}{\vdash \{x : T_1 \mid C_1\} \circledcirc T_2 \leadsto C}$$

(ND Rec)
$$\frac{\vdash (T\{\alpha/\mu\alpha.T\}) \circledcirc (U\{\beta/\mu\beta.U\}) \leadsto C}{\vdash (\mu\alpha.T) \circledcirc (\mu\beta.U) \leadsto C}$$

(ND Pair)
$$\frac{\vdash T_1 \circledcirc U_1 \leadsto C_1 \quad \vdash T_2 \circledcirc U_2 \leadsto C_2}{\vdash (T_1 * T_2) \circledcirc (U_1 * U_2) \leadsto C_1 \wedge C_2}$$

(ND Sum)
$$\frac{\vdash T_1 \circledcirc U_1 \leadsto C_1 \quad \vdash T_2 \circledcirc U_2 \leadsto C_2}{\vdash (T_1 + T_2) \circledcirc (U_1 + U_2) \leadsto (C_1 \vee C_2)}$$

(ND And)
$$\frac{\vdash T_1 \circledcirc U \leadsto C_1 \quad \vdash T_2 \circledcirc U \leadsto C_2}{\vdash (T_1 \wedge T_2) \circledcirc U \leadsto C_1 \wedge C_2}$$

(ND Or)
$$\frac{\vdash T_1 \circledcirc U \leadsto C_1 \quad \vdash T_2 \circledcirc U \leadsto C_2}{\vdash (T_1 \vee T_2) \circledcirc U \leadsto C_1 \vee C_2}$$

(ND Entails)
$$\frac{E \vdash T_1 \circledcirc T_2 \leadsto C \quad E, C \vdash C'}{E \vdash T_1 \circledcirc T_2 \leadsto C'}$$

(ND Sub)
$$\frac{E \vdash T \circledcirc U \leadsto C \quad E \vdash U' <: U}{E \vdash T \circledcirc U' \leadsto C}$$

Soundness

Calculus

- Surface calculus ($\mathrm{RCF}^{\forall}_{\wedge\vee}$)
  - explicitly 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 ($RCF^{\forall}_{\wedge\vee}$)
    - explicitly 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-$RCF^{\forall}_{\wedge\vee}$)
    - implicitly typed
    - 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 ($RCF^{\forall}_{\wedge\vee}$)
  - explicitly 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-$RCF^{\forall}_{\wedge\vee}$)
  - implicitly typed
  - 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 $\lambda x : ???\,.\,x$: $(\text{Private} \rightarrow \text{Private}) \wedge (\text{Un} \rightarrow \text{Un})$

$RCF^{\forall}_{\wedge\vee}$: 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 $\lambda x : ??? . x :$ $(Private \rightarrow Private) \wedge (Un \rightarrow Un)$

- $\lambda x{:}T_1; T_2.\ M$ [Reynolds '86, '96]
  - $(\lambda x{:}Private;Un.\ x)\ :\ (Private \rightarrow Private) \wedge (Un \rightarrow Un)$
  - can't write terms of type $(T_1 \rightarrow T_1 \rightarrow U_1) \wedge (T_2 \rightarrow T_2 \rightarrow U_2)$
    - you can use uncurried version $(T_1 \times T_1 \rightarrow U_1) \wedge (T_2 \times T_2 \rightarrow U_2)$ 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 $\lambda x : ??? . x : (Private \to Private) \wedge (Un \to Un)$

- $\lambda x{:}T_1;\ T_2.\ M$ [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

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

- 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)

Erasure crucial for soundness

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

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

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

- 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
    $\lfloor \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M \rfloor = \lfloor M \rfloor$

Erasure crucial for soundness

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

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

- 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
    $\lfloor \text{for } \alpha \text{ in } T_1;T_2 \text{ do } M \rfloor = \lfloor M \rfloor$
  - Fors don't have an operational semantics anyway!

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]
- Found+fixed 3 relatively small bugs in previous proofs
  - Public Down / Tainted Up, Robust Safety, Strengthening (claim weakened)
- Available at: http://www.infsec.cs.uni-saarland.de/projects/F5/

Transitivity of subtyping

- Cardelli's Amber rule makes transitivity proof a mess

$$\text{(Sub Rec)}\quad \frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \neq \alpha' \quad \alpha \notin \mathit{ftv}(T') \quad \alpha' \notin \mathit{ftv}(T)}{E \vdash \mu\alpha.T <: \mu\alpha'.T'}$$

(1) $E_{01} \vdash T <: T'$ and $E_{12} \vdash T' <: T''$ imply $E_{02} \vdash T <: T''$

(2) $E_{12} \vdash T'' <: T'$ and $E_{01} \vdash T' <: T$ imply $E_{02} \vdash T'' <: T$

where E_{01}, E_{12}, and E_{02} take the form

$$\begin{aligned} E_{01} &= E[(\alpha_i\ R_i\ \alpha_i')\ ^{i \in 1..n}] \\ E_{12} &= E[(\alpha_i'\ R_i\ \alpha_i'')\ ^{i \in 1..n}] \\ E_{02} &= E[(\alpha_i\ R_i\ \alpha_i'')\ ^{i \in 1..n}] \end{aligned}$$

for some number n, distinct type variables α_i, α_i', α_i'', relations $R_i \in \{<:, <:^{-1}\}$ for $i \in 1..n$, and executable environment E with $E \vdash \diamond$.

Transitivity of subtyping

- Cardelli's Amber rule makes transitivity proof a mess

$$\text{(Sub Rec)}\quad \frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \neq \alpha' \quad \alpha \notin \mathit{ftv}(T') \quad \alpha' \notin \mathit{ftv}(T)}{E \vdash \mu\alpha.\, T <: \mu\alpha'.\, T'}$$

- Went for a simpler rule instead
[Val Tannen, LICS '89]

$$\text{(Sub Pos Rec*)}\quad \frac{E, \alpha \vdash T <: U \quad \alpha \text{ only occurs positively in } T \text{ and } U}{E \vdash \mu\alpha.\, T <: \mu\alpha.\, U}$$

IS&C

UNIVERSITÄT DES SAARLANDES

Random thoughts for the future

- 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 / $RCF^{\forall}_{\wedge\vee}$
- Automating FO authorization logic with says (constructive)
- Study methods for establishing observational equivalence in RCF / $RCF^{\forall}_{\wedge\vee}$ (logical relations, bisimulations, etc.)

Other things I worked on so far ...

- Mechanized formalization of expi2java (useful tool)
- Automatically verifying typing constraints for Dminor (general refinement types + dynamic type-tests)
    - using (semantic subtyping **or** VCgen) + SMT solver
- Achieving security despite compromise using ZK proofs
- Type-checking protocols that use zero-knowledge proofs
- Automated verifying electronic voting protocols
- Step-indexed semantics of object calculi

Thank you!