Type-checking implementations of protocols based on zero-knowledge proofs

− work in progress −

Cătălin Hrițcu

Saarland University, Saarbrücken, Germany

Joint work with: Michael Backes, Matteo Maffei, and Thorsten Tarrach

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, etc.
  - **proved industrial protocols secure**
    EKE, JFK, TLS, DAA, Plutus, 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 were be proved, model and code will drift apart as the code evolves
- Most often the only "model" is the code itself
  - when given a proper semantics the security of code can be analyzed as well

Analyzing protocol implementations

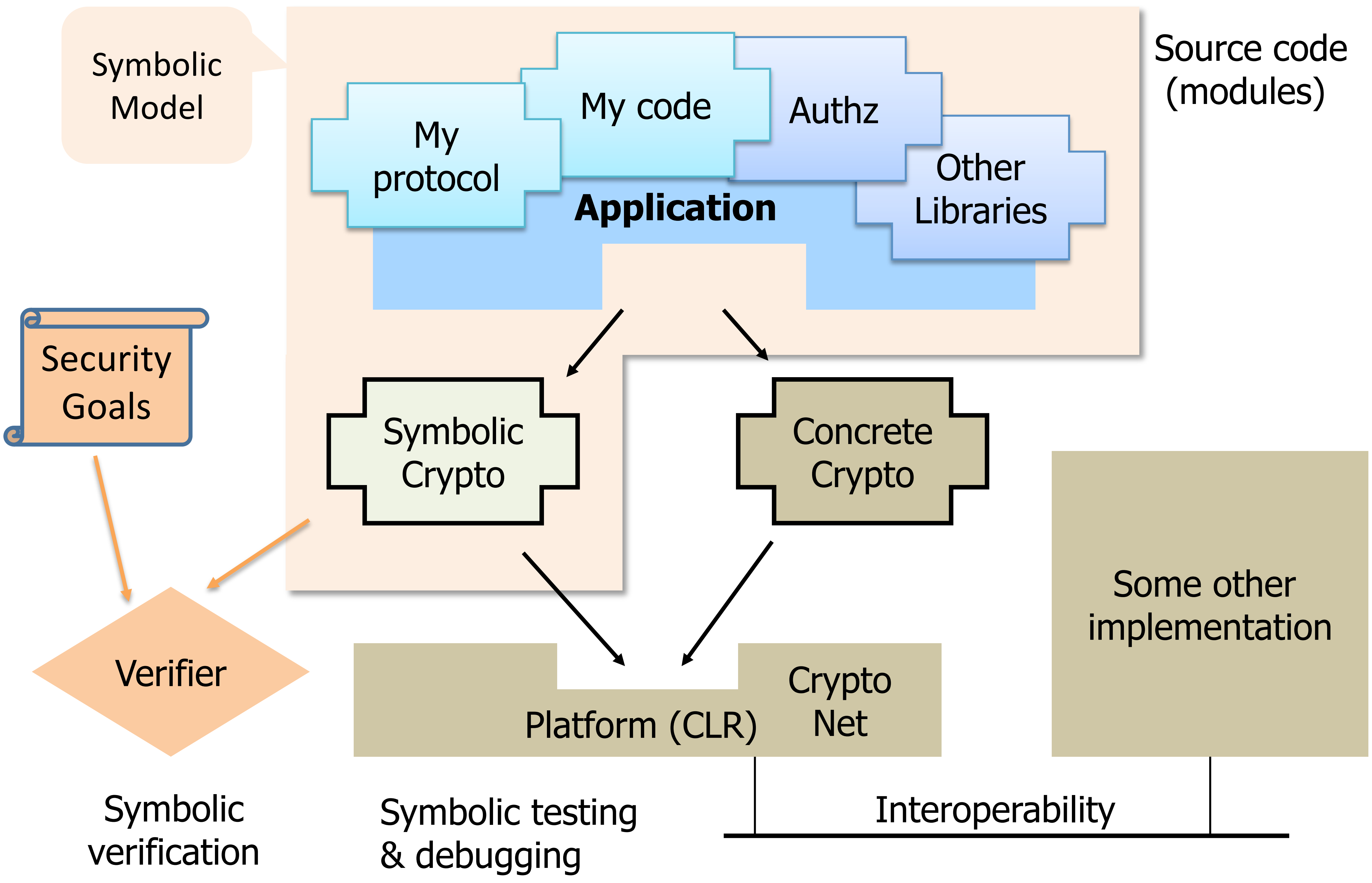
- Recently several approaches proposed
  - **static analysis:**
    CSur [Goubault-Larrecq and Parrennes, VMCAI'05]
  - **extracting ProVerif models:**
    fs2pv [Bhargavan, Fournet, Gordon & Tse, CSF '06]
  - **software model checking:**
    ASPIER [Chaki & Datta, CSF '09]
  - **typing:**
    F7 [Bengtson,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

F7 and RCF

F7 type-checker

- [Bengtson, Bhargavan, Fournet, Gordon & Maffeis CSF '08]
- Security type-checker for (large fragment of) F# (ML)
- Checks compliance with authorization policy
  - FOL used as authorization logic
  - proof obligations discharged using automated theorem prover
- Dual implementation of cryptographic library
  - symbolic (DY model): used for security verification, debugging
  - concrete (actual 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 — 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 — Nat = μα.α+unit

Access control example

assume CanWrite("/tmp") $\wedge$ $\forall$x.CanWrite(x) $\Rightarrow$ CanRead(x); (* policy *)

read : {file:String | CanRead(file)} → String

read = λfile. **assert** CanRead(file); … *actual read* …

delete : {file:String | CanWrite(file)} → unit

delete = λfile. **assert** CanWrite(file); … *actual delete* …

checkread : f:String → { unit | CanRead(f) }

checkread = λf. if f="README" then **assume** CanRead(f) else … *fail* …

let v1 = read "/tmp" in (* OK, allowed by policy *)

let v2 = read "/etc/passwd" in … (* ERROR, assert in read fails *)

checkread "README"; read "README"   (* OK, dynamically checked *)

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 closed assert-free RCF expressions
  - 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

Encoding symbolic cryptography

Symbolic cryptography

- RCF doesn't have any primitive for cryptography
- Instead, crypto primitives encoded using **dynamic sealing** [Morris, CACM '73]
- Advantage: adding new crypto primitives doesn't change RCF calculus, or type system, or soundness 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

      Seal<α> = (α→Un) * (Un→α)

      mkSeal<α> : unit → Seal<α>

Symmetric encryption

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

Key<α> = Seal<α> = (α→Un) * (Un→α)

mkKey<α> = mkSeal<α>

senc<α> = λk:Key<α>.λm:α. (fst k) m : Key<α>→α→Un

sdec<α> = λk:Key<α>.λn:Un. (snd k) n : Key<α>→Un→α

"Public-key" encryption

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

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

$mkDK\langle\alpha\rangle = mkSeal\langle\alpha\rangle$

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

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

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

“Public-key” encryption

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

$PK<\alpha> \;=\; \alpha \to Un$

$mkDK<\alpha> \;=\; mkSeal<\alpha>$

$mkPK<\alpha> \;=\; \lambda dk{:}DK<\alpha>.\ \mathrm{fst}\ dk \qquad : DK<\alpha> \to PK<\alpha>$

$enc<\alpha> \;=\; \lambda pk{:}PK<\alpha>.\lambda m{:}\alpha.\ pk\ m \qquad : PK<\alpha> \to \alpha \to Un$

$dec<\alpha> \;=\; \lambda dk{:}DK<\alpha>.\lambda n{:}Un.\ (\mathrm{snd}\ k)\ n \qquad : DK<\alpha> \to Un \to \alpha$

- A public key $pk{:}\ PK<\alpha>$ is only public when α is tainted!

"Public-key" encryption

DK<α> = Seal<α> = (α→Un) * (Un→α)

PK<α> = α→Un

mkDK<α> = mkSeal<α>

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

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

dec<α> = λdk:DK<α>.λn:Un. (snd k) n : 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 - FIXED

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

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

$\mathrm{mkDK}\langle\alpha\rangle = \mathrm{mkSeal}\langle\alpha\rangle$

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

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

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

- 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

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

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

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

$mkDK\langle\alpha\rangle = mkSeal\langle\alpha\rangle$

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

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

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

- 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

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

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

$mkDK\langle\alpha\rangle = mkSeal\langle\alpha\rangle$

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

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

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

- 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

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

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

$mkDK\langle\alpha\rangle = mkSeal\langle\alpha\rangle$

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

Union types introduced by subtyping

$m{:}\alpha$ and $\alpha <: \alpha\vee Un$

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

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

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

Digital signatures

SK<α> = Seal<α> = (α→Un) * (Un→α)

VK<α> = Un→α

mkSK<α> = mkSeal<α>

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

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

verify<α> = λvk:VK<α>.λn:Un.λm:Un.
let m′=vk n in
if m′=m then m′ : VK<α>→Un→Un→α

Digital signatures

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

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

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

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

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

$verify\langle\alpha\rangle = \lambda vk{:}VK\langle\alpha\rangle.\lambda n{:}Un.\lambda m{:}Un.$
let $m' = vk\ n$ in
if $m' = m$ then $m' \qquad : VK\langle\alpha\rangle \rightarrow Un \rightarrow Un \rightarrow \alpha$

- A key vk: $VK\langle\alpha\rangle$ is public only when α is public!

Digital signatures - FIXED

SealSig<α> = (α→Un) * (Un→((α∨Un)→α)∧(Un→Un))

SK<α> = SealSig<α>

VK<α> = Un→((α∨Un)→α)∧(Un→Un)

mkSK<α> = mkSealSig<α>

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

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

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

: VK<α>→[Un→((α∨Un)→α)∧(Un→Un)]

Digital signatures - FIXED

$SealSig\langle\alpha\rangle = (\alpha \to Un) * (Un \to ((\alpha \vee Un) \to \alpha) \wedge (Un \to Un))$

Verification keys are always public

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

$SK\langle\alpha\rangle = SealSig\langle\alpha\rangle$

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

$mkSK\langle\alpha\rangle = mkSealSig\langle\alpha\rangle$

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

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

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

$: VK\langle\alpha\rangle \to [Un \to ((\alpha \vee Un) \to \alpha) \wedge (Un \to Un)]$

Digital signatures - FIXED

SealSig<α> = (α→Un) * (Un→((α∨Un)→α)∧(Un→Un))

mkSealSig<α> = λ_:unit. let (s,u) = mkSeal () in
let v = λn:Un. λm:α∨Un ; Un.
if m = u n as z then z
in (s,v)

SK<α> = SealSig<α>

VK<α> = Un→((α∨Un)→α)∧(Un→Un)

mkSK<α> = mkSealSig<α>

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

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

verify<α> = λvk:VK<α>. λn:Un. λm:Un. vk n m
: VK<α>→[Un→((α∨Un)→α)∧(Un→Un)]

Encoding zero-knowledge proofs

Very simplified DAA-sign

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

T_{vki} = {x_f : Private | OkTPM(x_f)}

new c : Un. **let** ski = mkSK$<T_{vki}>$ () **in** **let** vki = mkVK$<T_{vki}>$ ski **in**

((* TPM *)
    (* abstract away the join protocol *)
    let f = mkPriv () **in**
    assume okTPM(f);
    let cert = sign$<T_{vki}>$ ski f **in**

    let m = mkUn () **in** **assume** Send(f, m);
    let zk = zk-create$_{daa}$ (vki, m, f, cert) **in**
    c!zk
) | ((* Verifier *)
    let x = c? **in**
    let (y_2,y_3) = zk-verify$_{daa}$ x vki **in**
    assert Authenticate(y_2)
)

Very simplified DAA-sign

assume ∀m. (∃f. Send(f,m) ∧ OkTPM(f)) ⇒ Authenticate(m)); (* policy *)

T_{vki} = {x_f : Private | OkTPM(x_f)}

new c : Un. **let** ski = mkSK<T_{vki}> () **in** **let** vki = mkVK<T_{vki}> ski **in**

((* TPM *)
    (* abstract away the join protocol *)
    let f = mkPriv () **in**
    assume okTPM(f);
    let cert = sign<T_{vki}> ski f **in**

    let m = mkUn () **in** **assume** Send(f, m);
    let zk = zk-create$_{daa}$ (vki, m, f, cert) **in**
    c!zk
) | ((* Verifier *)
    let x = c? **in**
    let (y_2,y_3) = zk-verify$_{daa}$ ...
    assert Authenticate(y_2)
)

ZK proof shows that
“verify<T_{vki}> vki cert f” succeeds

Very simplified DAA-sign

assume ∀m. (∃f. Send(f,m) ∧ OkTPM(f)) ⇒ Authenticate(m)); (* policy *)

T_{vki} = {x_f : Private | OkTPM(x_f)}

new c : Un. **let** ski = mkSK<T_{vki}> () **in let** vki = mkVK<T_{vki}> ski **in**

((* TPM *)
    (* abstract away the join protocol *)
    let f = mkPriv () **in**
    assume okTPM(f);
    let cert = sign<T_{vki}> ski f **in**

    let m = mkUn () **in assume** Sen… ,m);
    let zk = zk-create$_{daa}$ (vki, m, f, cert) **in**
    c!zk
) | ((* Verifier *)
    let x = c? **in**
    let (y_2,y_3) = zk-verify$_{daa}$ …
    assert Authenticate(y_2)
)

Without revealing f or cert
(secret witnesses)

ZK proof shows that
"verify<T_{vki}> vki cert f" succeeds

Very simplified DAA-sign

assume ∀m. (∃f. Send(f,m) ∧ OkTPM(f)) ⇒ Authenticate(m)); (* policy *)

T_{vki} = {x_f : Private | OkTPM(x_f)}

new c : Un. **let** ski = mkSK<T_{vki}> () **in** **let** vki = mkVK<T_{vki}> ski **in**

```
( (* TPM *)
    (* abstract away the join protocol *)
    let f = mkPriv () in
    ... f in
    ... me Sen...
    let zk = zk-create_daa (vki, m, f, cert) in
    c!zk
) | ( (* Verifier *)
    let x = c? in
    let (y2,y3) = zk-verify_daa ...
    assert Authenticate(y2)
)
```

Proof non-malleable,
authenticity of m desired

Without revealing f or cert
(secret witnesses)

ZK proof shows that
"verify<T_{vki}> vki cert f" succeeds

High-level specification

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

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

zkdef daa =

matched = $[y_{vki} : \mathrm{VK}<T_{vki}>]$

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

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

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

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

High-level specification

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

$T_{vki} = \{x_f : Private \mid OkTPM(x_f)\}$

Public value known to the verifier

zkdef daa =

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

returned = [y_m : Un]

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

statement = [x_f = verify<T_{vki}> y_{vki} x_{cert} x_f]

promise = [Send(x_f,y_m)].

High-level specification

assume ∀m. (∃f. Send(f,m) ∧ OkTPM(f)) ⇒ Authenticate(m)); (* policy *)

T_{vki} = {x_f : Private | OkTPM(x_f)}

zkdef daa =

matched = [y_{vki}

Public value not known to the verifier

returned = [y_m : Un]

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

statement = [x_f = verify<T_{vki}> y_{vki} x_{cert} x_f]

promise = [Send(x_f,y_m)].

High-level specification

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

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

zkdef daa =

matched = $[y_{vki} : VK\text{<}T$

returned = $[y_m : Un$

Witnesses, never revealed
(but prover knows them)

secret = $[x_f : T_{vki}, x_{cert} : Un]$

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

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

High-level specification

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

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

zkdef daa =

matched = $[y_{vki} : \mathrm{VK}<T_{vki}>]$

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

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

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

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

Statement of the proof
(positive Boolean formula)

High-level specification

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

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

zkdef daa =

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

returned = [y_m : Un]

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

statement = [x_f = verify<T_{vki}> y_{vki} x_{cert} x_f]

promise = [Send(x_f,y_m)].

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

Generated implementation

T_{vki} = {x_f : Private | OkTPM(x_f)}

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

Generated implementation

$T_{vki} = \{x_f : Private \mid OkTPM(x_f)\}$

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

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

Generated implementation

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

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

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

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

Generated implementation

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

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

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

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

$\text{zk-public}_{daa} = \lambda z{:}\mathrm{Un}.\ \mathbf{case}\ w' = (\mathrm{snd}\ k_{daa})\ z : T_{daa}\vee\mathrm{Un}\ \mathbf{of} \qquad : \mathrm{Un} \to \mathrm{Un}$

$\qquad \mathbf{let}\ (y_{vki}, y_m, s) = w'\ \mathbf{in}\ (y_{vki}, y_m)$

Generated implementation

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

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

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

$\text{zk-create}_{daa} = \lambda w{:}T_{daa} \vee \mathrm{Un}.$ (… $\mathrm{Un} \rightarrow \mathrm{Un}$

Elimination construct for union types

$\text{zk-public}_{daa} = \lambda z{:}\mathrm{Un}.\ \mathbf{case}\ w' = (\mathrm{snd}\ k_{daa})\ z : T_{daa} \vee \mathrm{Un}\ \mathbf{of}$ $\quad : \mathrm{Un} \rightarrow \mathrm{Un}$

$\qquad \mathbf{let}\ (y_{vki}, y_m, s) = w'\ \mathbf{in}\ (y_{vki}, y_m)$

Generated implementation

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

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

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

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

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

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

Generated implementation

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

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

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

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

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

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

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

Case #1: honest verifier, honest prover

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

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

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

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

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

Case #2: honest verifier, dishonest prover

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

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

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

$\lambda z{:}\text{Un}.\ \lambda y_{vki}' : \text{VK}<T_{vki}>;\ \text{Un}.$ — $y_{vki}' : \text{VK}<T_{vki}>$

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 $x_f = \text{verify}<T_{vki}>\ y_{vki}''\ x_{cert}\ x_f$ **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"

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

Cases #3 & #4: dishonest verifier

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

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

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

$\lambda z{:}\text{Un}.\ \lambda y_{vki}' : \text{VK}\langle T_{vki}\rangle;\ \text{Un}.$ — $y_{vki}' : \text{Un}$ (#3) — $y_{vki}' : \text{VK}\langle T_{vki}\rangle$ (#4)

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** — $x_f : \text{Un}$

if $y_{vki} = y_{vki}'$ **as** y_{vki}'' **then** — $y_{vki}'' : \text{Un} \wedge \ldots$

   if $x_f = \text{verify}\langle T_{vki}\rangle\ y_{vki}''\ x_{cert}\ x_f$ **then** (y_m) — $y_m : \text{Un}$

   else failwith "statement not valid"

else failwith "y_{vki} does not match"

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

Cases #3 & #4: dishonest verifier

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

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

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

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

$\lambda z{:}\text{Un}.\ \lambda y_{vki}' : \text{VK}\langle T_{vki}\rangle;\ \text{Un}.$	$y_{vki}' : \text{Un}$ (#3)	$y_{vki}' : \text{VK}\langle T_{vki}\rangle$ (#4)
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**		$x_f : \text{Un}$
if $y_{vki} = y_{vki}'$ **as** y_{vki}'' **then**		$y_{vki}'' : \text{Un} \wedge \ldots$
if $x_f = \text{verify}\langle T_{vki}\rangle\ y_{vki}''\ x_{cert}\ x_f$ **then** (y_m)		$y_m : \text{Un}$
else failwith "statement not valid"		
else failwith "y_{vki} does not match"		

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

$\mathrm{RCF}^{\forall}_{\wedge\vee}$

design choices & technical difficulties

Intrinsic vs extrinsic typing

- Church-style ($RCF^{\forall}_{\wedge\vee}$) vs. Curry-style (RCF)
- Our reasons for going intrinsically typed
    - Type-checking and type inference decoupled
        - Type-checking for RCF already undecidable (FOL)
        - Type inference for refinement types alone is hot research topic [Liquid Types; Rondon, Kawaguchi & Jhala, PLDI 08']
        - Type inference for "System *D*" is equivalent to strong normalizability of untyped λ-calculus terms (undecidable)
        - For now we move type inference burden to programmer
    - Wanted to encode type Private that is disjoint from Un
    - Seemed to help in the proofs (stronger inversion principles)

Introduction of intersection types

- Because of type annotations need an explicit construct
- $\lambda x{:}T_1;\ T_2.\ M$ works but is quite restrictive [Reynolds '96]
  - can only introduce intersections between function types
  - 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
  - no way to refer to the type of argument in function body
- Type alternation: for α in T; U do A [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 $\lambda x{:}\alpha.\lambda x{:}\alpha.M : (T_1 \to T_1 \to U_1) \wedge (T_2 \to T_2 \to U_2)$
  - for α in T_1;T_2 do $\lambda x{:}\alpha.\ \mathrm{enc}\langle\alpha\rangle\ k\ x$

Type refinements vs. alternation

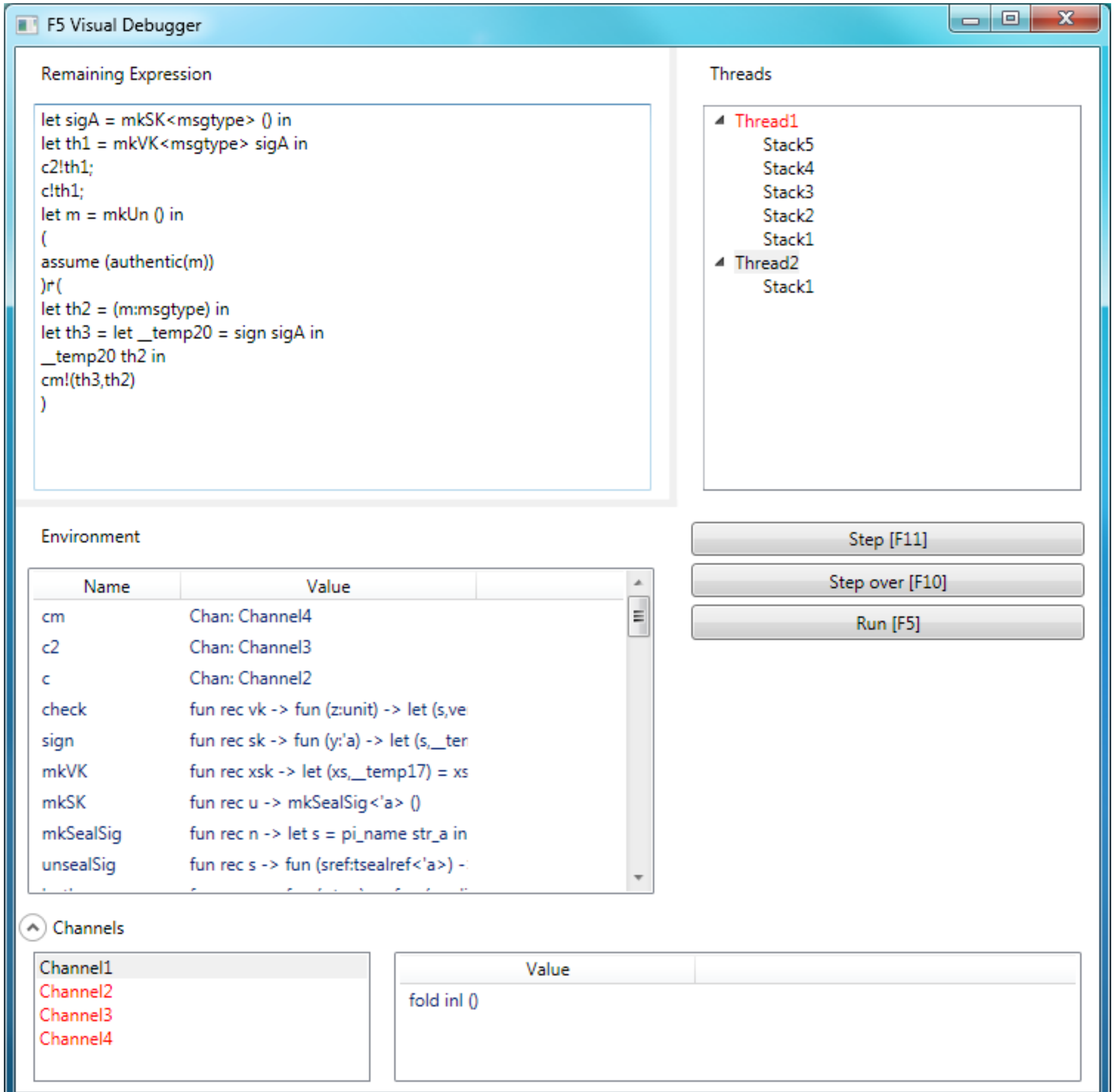
- Refinement: if $\Gamma \vdash M:T$ and $\Gamma \vdash C\{M/x\}$ then $\Gamma \vdash M:\{x:T \mid C\}$
- Alternation: if $\Gamma \vdash A\{T_1/\alpha\} : T$ or $\Gamma \vdash A\{T_2/\alpha\} : T$
  then $\Gamma \vdash$ for α in T_1; T_2 do $A : T$
- Counterexample:
  Let $\vdash M\{T_1/\alpha\}:T$, we have $\vdash M\{T_1/\alpha\}=M\{T_1/\alpha\}$ so also
  $\vdash M\{T_1/\alpha\} : \{x:T \mid x=M\{T_1/\alpha\}\}$ so
  $\vdash$ for α in T_1; T_2 do $M : \{x:T \mid x=M\{T_1/\alpha\}\}$,
  which is wrong(!) since $\vdash$ for α in T_1; T_2 do $M \neq M\{T_1/\alpha\}$
- Our current solution for this is complicated and nasty
- Type alternation construct breaks other things as well
  - Doesn't work properly for functions with side-effects

Implementation (F5) & case studies

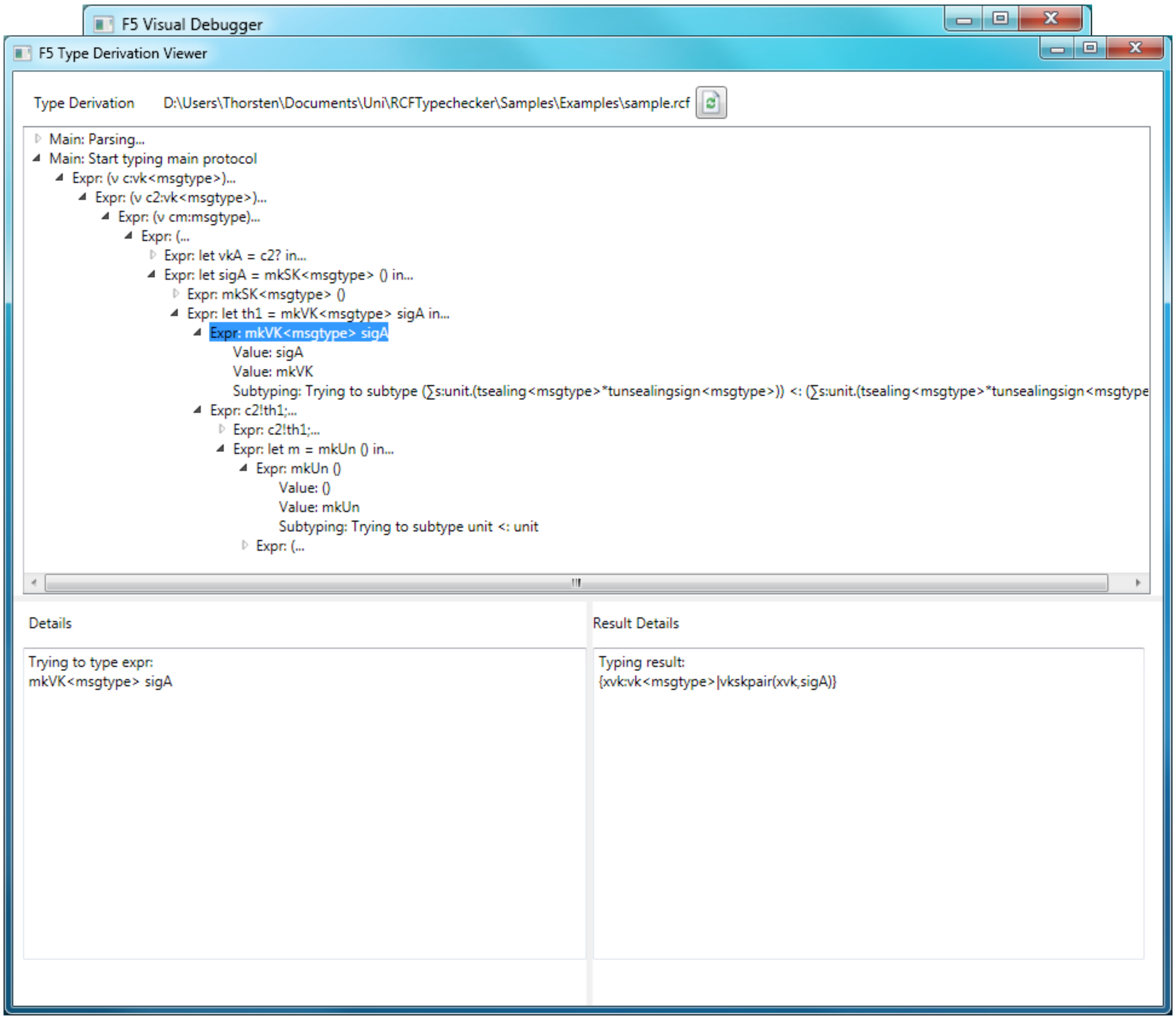
F5: tool-chain for $RCF^{\forall}_{\wedge\vee}$

- 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
- Automatic code generator for zero-knowledge
- Interpreter/debugger
- Spi2RCF automatic code generator
- Experimental RCF2F# automatic code generator
- First release coming soon

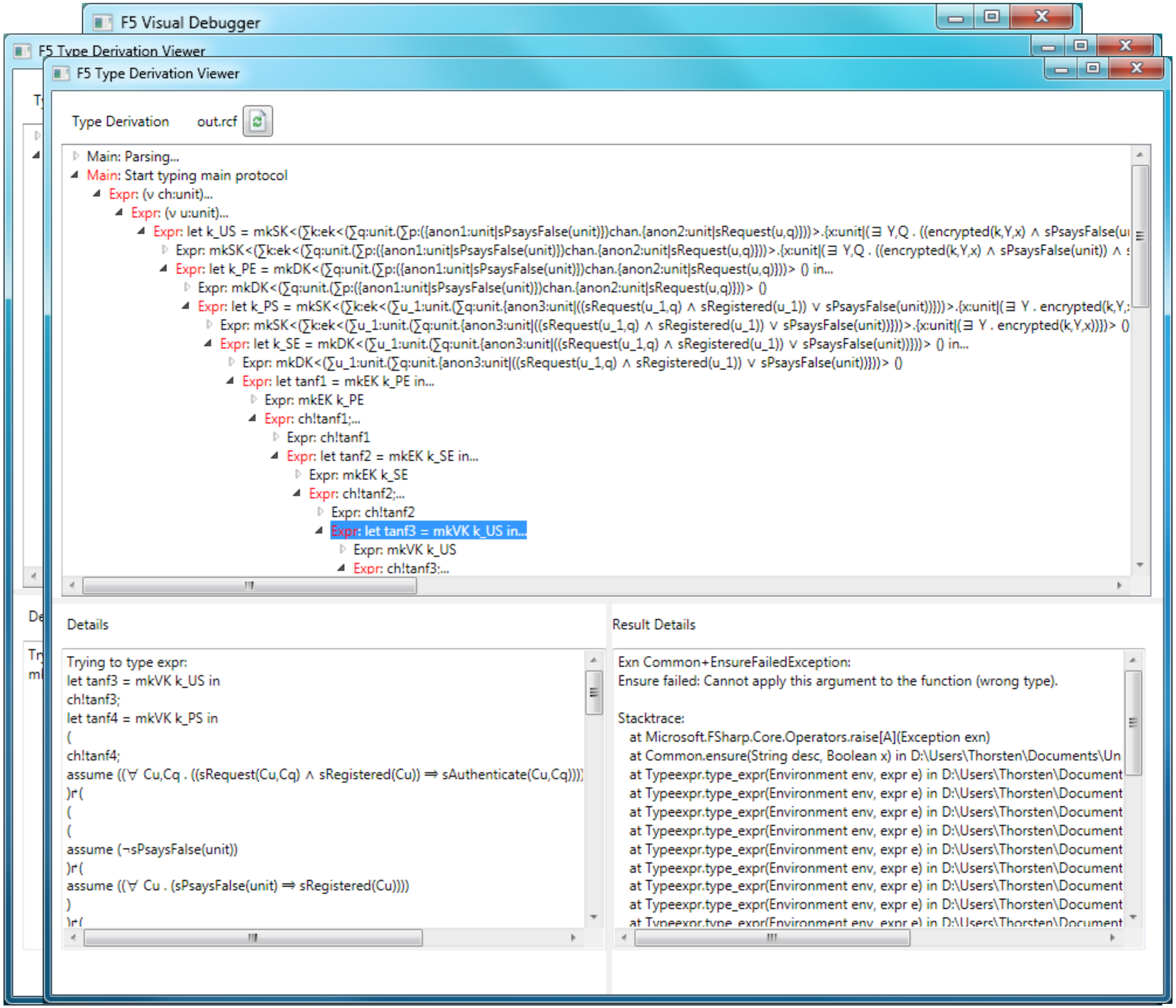
Screenshots



Screenshots



Screenshots



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, S&P '08]
   - Work in progress (Matteo Maffei & Fabienne Eigner)
   - Other complex primitives: distributed encryption with re-encryption and plaintext equivalence testing (PET)

Thoughts for the future

- Study type inference, maybe in restricted setting
- Prove semantic properties of ZK encoding
- 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 cryptographic implementation hard to do by hand
  - efficiency is a big challenge

Thank you!