

Refinement Types for Secure Implementations

Jesper Bengtson, Uppsala University

Karthikeyan Bhargavan, Microsoft Research

Cédric Fournet, Microsoft Research

Andrew D. Gordon, Microsoft Research

Sergio Maffeis, Imperial College and UC Santa Cruz

<http://research.microsoft.com/F7>

Refinement types for secure implementations

type payload = string

\rightsquigarrow **type** payload = x: string { Send(a,x) }

1. Context, motivation
 - Verifying protocol implementations
 - Logics for authorization and access control
2. A refinement-typed concurrent lambda-calculus (theory)
3. Cryptography by kinding, subtyping, and sealing
4. Experimental results (F#)

F# demo available on request

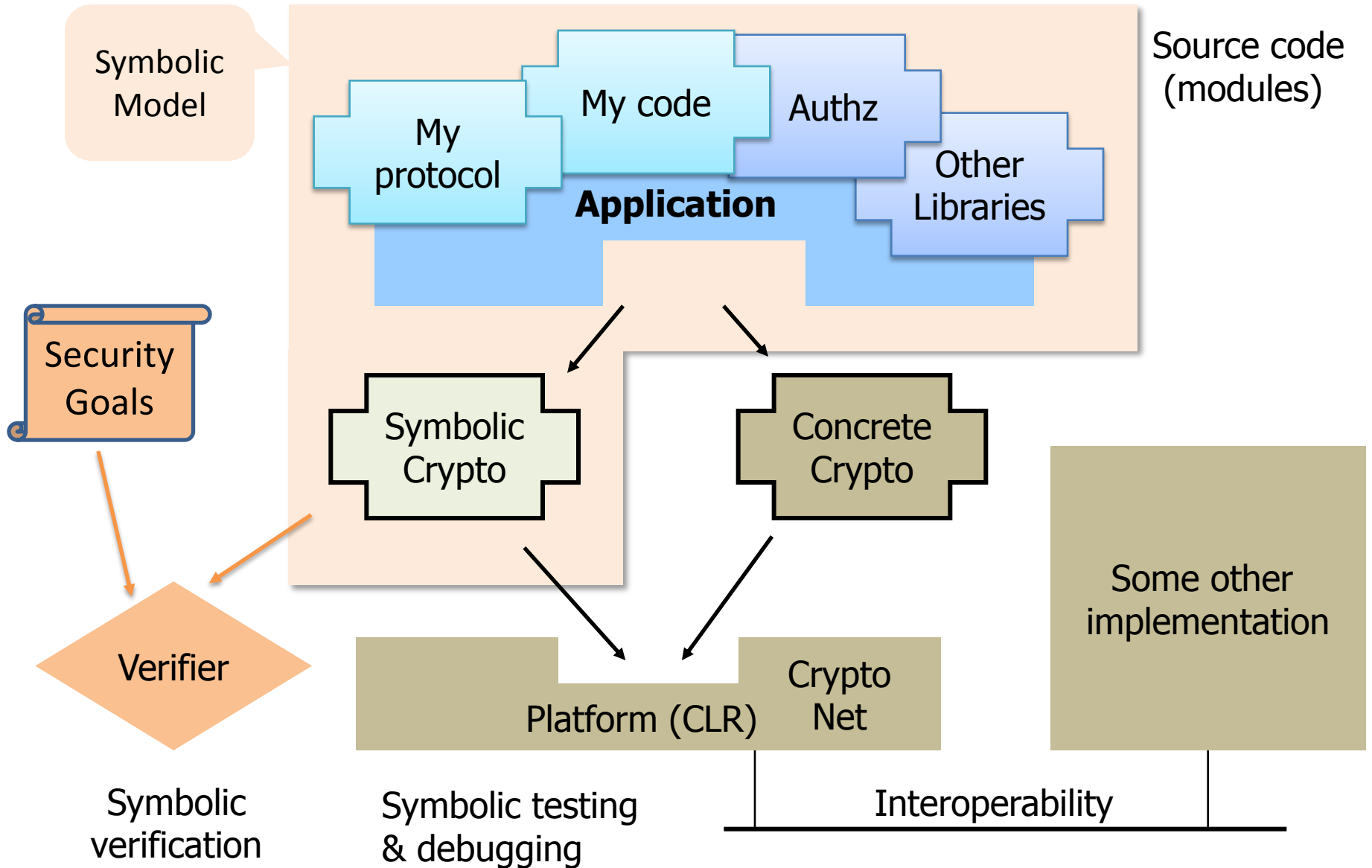
Motivation (1/2)

VERIFYING IMPLEMENTATIONS

Verifying protocol implementations

- Cryptographic protocols specs, models, and implementations
 - Protocol specifications remain largely informal
 - Formal models are short, abstract, hand-written
 - Specs, models, and implementations drift apart...
- Our current approach is to verify **reference implementations**
 - Executable code has more details than models
 - Executable code has better tool support:
types, compilers, testing, debuggers, libraries, verification

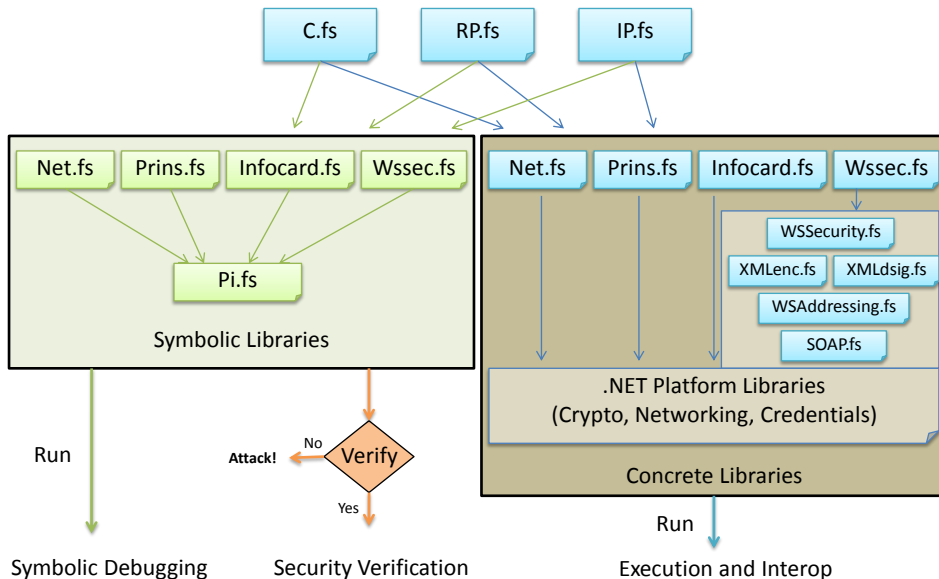
The F# | FS2PV | PV tool chain [CSF'06]



The F# | FS2PV | PV tool chain: scalability issues

- Even with some aggressive abstraction, our tools are hitting long and unpredictable run times
- Can we do better with source-level security types?

A Reference InfoCard Implementation



Safety Results

Name	LOC	Crypto Ops	Auth	Secrecy	Verif Time
SelfIssued-SOAP	1410 (80)	9,3	A1-A3	S1,S2	38s
UserPassword-TLS	1426(96)	0,5,17,6	A1-A3	S1,S2	24m40s
UserPassword-SOAP	1429(99)	9,11,17,6	A1-A3	S1,S2	20m53s
UserCertificate-SOAP	1429(99)	13,7,11,6	A1-A3	S1-S3	66m21s
UserCertificate-SOAP-v	1429(99)	7,5,7,4	A3 Fails!	S1-S3	10s

Protocol verification vs program verification

- Cryptographic tools (ProVerif, CryptoVerif, AVISPA) are great but...
 - They use global analyses, not suitable for “giant” protocols
 - Security applications combine both crypto protocols and ordinary code
- General-purpose verification techniques are also making rapid progress
 - They can deal with much larger programs
 - They don’t directly support protocol verification with cryptographic primitive, active adversaries, etc
- Using dependent types, we can integrate cryptographic protocol verification as a part of program verification

Motivation (2/2)

AUTHORIZATION POLICIES & PARTIALLY-TRUSTED CODE

Authorization Logics

- Authorization policies are complex and changing
 - How to express policies? → use some logic [Datalog, DCC, SecPAL]
 - How to **statically** check whether a system implements a policy?
- Authorization by typing [ESOP'05, CSF'07]:
 - Represent code in a process calculus
 - Represent policies as logical formulas within types
 - Generalize type and effect systems for formal cryptography
- This work:
 - Typecheck implementations (in F#) against policies (in FOL)
 - Simplify the theory: we now derive formal cryptography

Specification: assume, then assert

- Suppose there is a global set of formulas, the **log**
- To evaluate **assume** C , add C to the log.
- To evaluate **assert** C ,
 - If C logically follows from the logged formulas, the assertion **succeeds**; otherwise, the assertion **fails**.
 - The log is only for specification purposes; it does not affect execution
- Our use of first-order logic predicates generalizes conventional assertions (like **assert** $i > 0$ in eg Spec#)
 - Such predicates can also represent security-related concepts like roles, permissions, events, compromises, access rights,...

Example: access control for files

- **Untrusted** code may call a **trusted** library
- Trusted code expresses security policy with **assumes** and **asserts**

```
type facts = CanRead of string | CanWrite of string
```

```
let read file = assert(CanRead(file)); ...
```

```
let delete file = assert(CanWrite(file)); ...
```

```
let pwd = "C:/etc/password"
```

```
let tmp = "C:/temp/tempfile"
```

```
assume CanWrite(tmp)
```

```
assume  $\forall x. \text{CanWrite}(x) \rightarrow \text{CanRead}(x)$ 
```

- Each policy violation causes an assertion failure
- We **statically** prevent any assertion failures by typing

```
let untrusted() =
```

```
  let v1 = read tmp in // ok, by policy
```

```
  let v2 = read pwd in // assertion fails
```

Typechecking failed at `acls.fs(39,9)–(39,12)`
Error: Cannot establish formula `CanRead(pwd)`

Logging dynamic events

- Security policies often stated in terms of dynamic events such as role activations or data checks
- We mark such events by adding formulas to the log with **assume**

```
type facts = ... | PublicFile of string
let read file = assert(CanRead(file)); ...
let readme = "C:/public/README"

// Dynamic validation:
let publicfile f =
  if f = "C:/public/README" || ...
  then assume (PublicFile(f))
  else failwith "not a public file"

assume  $\forall x. \text{PublicFile}(x) \rightarrow \text{CanRead}(x)$ 
```

```
let untrusted() =
  let v2 = read readme in // assertion fails
  publicfile readme; // validate the filename
  let v3 = read readme in () // now, ok
```

Access control with refinement types

```
val read: file:string{ CanRead(file)} → string  
val delete: file:string{ CanDelete(file)} → unit  
val publicfile: file:string → unit{ PublicFile(file)}
```

- Preconditions express access control requirements
- Postconditions express results of validation
- We typecheck partially trusted code to guarantee that all preconditions (and hence all asserts) hold at runtime

- Related work: eg types for stack inspection (Pottier, Skalka, Smith), Aura (Zdancevic et al)

a concurrent call-by-value lambda-calculus
with refinement types

“RCF”

Syntax for values and expressions

- A concurrent call-by-value lambda-calculus
- A formal core for F#
- Specifications expressed by **assume** and **assert** over logic formulas

a	name
x	variable
$h ::= \text{inl} \mid \text{inr} \mid \text{fold}$	value constructor
$M, N ::=$	value
x	variable
$()$	unit
fun $x \rightarrow A$	function (scope of x is A)
(M, N)	pair
$h M$	construction
$A, B ::=$	expression
M	value
$M N$	application
$M = N$	syntactic equality
let $x = A$ in B	let (scope of x is B)
let $(x, y) = M$ in A	pair split (scope of x, y is A)
match M with	constructor match
$h x \rightarrow A$ else B	(scope of x is A)
$(\nu a)A$	restriction (scope of a is A)
$A \dot{\uparrow} B$	fork
$a!M$	transmission of M on channel a
$a?$	receive message off channel
assume C	assumption of formula C
assert C	assertion of formula C

Semantics: expression safety

- We use a standard small-step reduction semantics; runtime configurations are expressions of the form

$$\mathbf{S} ::= (va_1) \dots (va_\ell) \left(\left(\prod_{i \in 1..m} \mathbf{assume} C_i \right) \overrightarrow{\vdash} \left(\prod_{j \in 1..n} c_j!M_j \right) \overrightarrow{\vdash} \left(\prod_{k \in 1..o} \mathcal{L}_k\{e_k\} \right) \right)$$

active	pending	running
assumptions	messages	threads

- An expression is **safe** when,
for all runs of A, **all assertions succeed**

Refinement types

- An assembly of standard components

$H, T, U ::=$	type
α	type variable
unit	unit type
$\Pi x : T. U$	dependent function type (scope of x is U)
$\Sigma x : T. U$	dependent pair type (scope of x is U)
$T + U$	disjoint sum type
$\mu \alpha. T$	iso-recursive type (scope of α is T)
$\{x : T \mid C\}$	refinement type (scope of x is C)

- For example, **type** filename = x:string{ CanRead(x) } declares a type of strings for filename with the read access right

Safety by typing

$E \vdash \diamond$	E is syntactically well-formed
$E \vdash T$	in E , type T is syntactically well-formed
$E \vdash C$	formula C is derivable from E
$E \vdash T :: v$	in E , type T has kind $v \in \{\mathbf{pub}, \mathbf{tnt}\}$
$E \vdash T <: U$	in E , type T is a subtype of type U
$E \vdash A : T$	in E , expression A has type T

$\mu ::=$	environment entry
$\alpha :: v$	kinding
$\alpha <: \alpha'$	subtyping
$a : T \updownarrow$	name (of channel type)
$x : T$	variable (of any type)
$E ::= \mu_1, \dots, \mu_n$	environment

An expression A is *safe* if and only if,
in all evaluations of A , all assertions succeed.

Theorem 1 (Safety by Typing) *If $\emptyset \vdash A : T$ then A is safe.*

Rules for refinements

We can refine any type
with any formula
that follows from E

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

$$\frac{E \vdash T <: T'}{E \vdash \{x : T \mid C\} <: T'} \quad \frac{E \vdash T <: T' \quad E, x : T \vdash C}{E \vdash T <: \{x : T' \mid C\}}$$

Rules for assume and assert

$$\frac{E \vdash \diamond \quad \text{fnfv}(C) \subseteq \text{dom}(E)}{E \vdash \mathbf{assume} C : \{- : \mathbf{unit} \mid C\}}$$

We can assume
any formula

$$\frac{E \vdash C}{E \vdash \mathbf{assert} C : \mathbf{unit}}$$

We can assert
any formula that
follows from E

Robust safety

- Active opponents can intercept all communications, rewrite messages, inject new messages, but not break cryptography [Needham-Shroeder'76, Dolev-Yao'83]
- We represent opponents as top-level programs, with access to selected libraries and functions **without asserts, but not necessarily well-typed**

Robust safety by typing

An expression A is *robustly safe* iff
the application $O A$ is safe for all opponents O .

Let a type T be *public* if and only if $T <: \mathbf{Un}$.

Let a type T be *tainted* if and only if $\mathbf{Un} <: T$.

Theorem 2 (Robust Safety by Typing)

If $\emptyset \vdash A : \mathbf{Un}$ then A is robustly safe.

Lemma 1 (Universal Type)

There is a type \mathbf{Un} such that $E \vdash \diamond$ implies $E \vdash \mathbf{Un} <:> T$

where T ranges over *unit*, $\Pi x : \mathbf{Un}. \mathbf{Un}$, $\Sigma x : \mathbf{Un}. \mathbf{Un}$, $\mathbf{Un} + \mathbf{Un}$, $\mu \alpha. \mathbf{Un}$, and *ChanTypeUn*.

Lemma 2 (Opponent Typability)

If O is an opponent and $E \vdash u : \mathbf{Un}$ for each $u \in \text{fnfv}(O)$, then $E \vdash O : \mathbf{Un}$.

Typed functional encoding of cryptography

- In prior (pi calculus) work, we included a selection of cryptographic primitives and typing rules
- We now derive typed cryptographic functions from **seals** [Morris'73]

```
type  $\alpha$  Seal = ( $\alpha \rightarrow \text{Un}$ ) * ( $\text{Un} \rightarrow \alpha$ )  
val mkSeal: unit  $\rightarrow$   $\alpha$  Seal
```

- We rely on functions and fresh names
- We obtain a symbolic model, similar to oracles
- We then code typed symbolic implementations for standard primitives

```
type  $\alpha$  SealChan = (( $\alpha * \text{Un}$ ) list) Pi.chan  
let seal:  $\alpha$  SealChan  $\rightarrow$   $\alpha \rightarrow \text{Un}$  = fun s M  $\rightarrow$   
  let state = recv s in match first (left M) state with  
  | Some(a)  $\rightarrow$  send s state; a  
  | None  $\rightarrow$   
    let a: Un = Pi.name "a" in  
    send s ((M,a)::state); a  
let unseal:  $\alpha$  SealChan  $\rightarrow$  Un  $\rightarrow$   $\alpha$  = fun s a  $\rightarrow$   
  let state = recv s in match first (right a) state with  
  | Some(M)  $\rightarrow$  send s state; M  
  | None  $\rightarrow$  failwith "not a sealed value"  
let mkSeal () :  $\alpha$  Seal =  
  let s:  $\alpha$  SealChan = chan "seal" in  
  send s []; (seal s, unseal s)
```

Example: symbolic implementation for MACs

Message authentication codes (MAC) provide integrity using joint hashes of a shared key and authenticated message

1. We declare an abstract interface for MACs, eg HMACSHA1:

```
type  $\alpha$  hkey  
type hmac  
val mkHKey: unit  $\rightarrow$   $\alpha$  hkey  
val hmacsha1:  $\alpha$  hkey  $\rightarrow$   $\alpha$  pickled  $\rightarrow$  hmac  
val hmacsha1Verify:  $\alpha$  hkey  $\rightarrow$  Un  $\rightarrow$  hmac  $\rightarrow$   $\alpha$  pickled
```

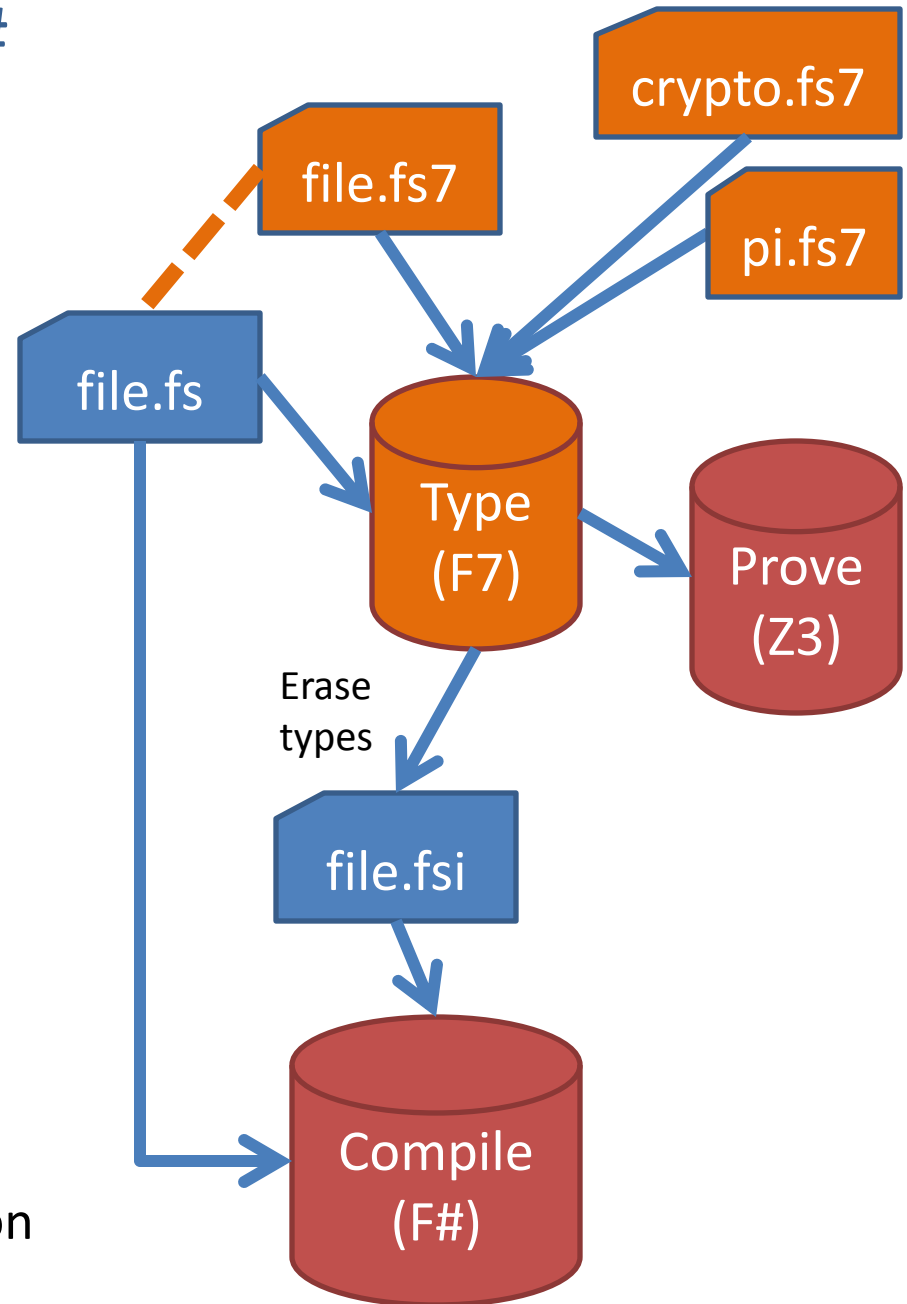
2. We implement (and typecheck) HMACSHA1 symbolically using seals

```
type  $\alpha$  hkey = HK of ( $\alpha$  pickled) Seal  
type hmac = HMAC of Un  
let mkHKey ():  $\alpha$  hkey = mkSeal()  
let hmacsha1 (HK(seal,unseal) text = HMAC(seal text)  
let hmacsha1Verify (HK(seal,unseal)) text (HMAC h) =  
  if unseal h = text then text else failwith "hmac verify failed"
```

EXPERIMENTAL RESULTS

Implementation for F#

- We use extended interfaces
 - We typecheck implementations
 - We kindcheck interfaces (all values must be public)
 - We generate .fsi interfaces by erasure from .fs7
- We support a large subset of F#
 - ADTs, records, patterns, refs
 - Value- and type-polymorphism
- We do some type inference
 - Plain F# types as usual
 - Refinements require annotations
- We call Z3, an SMT prover, on each non-trivial proof obligation



Libraries

- We annotate (and retype) some libraries
- We also provide concrete implementations for some system libraries (without extended typing)

```
open System.Security.Cryptography
type  $\alpha$  hkey = bytes
type hmac = bytes
let mkHKey () = mkNonce()
let hmacsha1 (k: $\alpha$  hkey) (x:bytes) =
  (new HMACSHA1 (k)).ComputeHash x
let hmacsha1Verify (k: $\alpha$  hkey) (x:bytes) (h:bytes) =
  let hh = (new HMACSHA1 (k)).ComputeHash x in
  if h = hh then x else failwith "hmac verify failed"
```

Evaluation so far

Sample	.fs	.fs7	time (S)	Z3 proofs	time (S) / proof	proofs / loc
Logs and queries	37	16	2.80	6	0.47	0.16
MAC protocols	40	12	2.50	3	0.83	0.08
Principals & partial Compromise	48	26	3.10	12	0.26	0.25
Certificate chains	61	21	3.65	19	0.19	0.31
File access control	104	34	8.30	16	0.52	0.15
Flexible signatures	167	52	14.60	28	0.52	0.17
Typed libraries	440	146	12.10	12	1.01	0.03

- We verify non-trivial code and properties
- Ask for a demo!

Sample: flexible digital signatures

- Signing keys are often shared among several protocols, e.g. XML digital signatures provide extreme, programmable flexibility
- How to avoid ambiguous signatures?
 - When signing, we must ensure that
 - (1) the signed content authenticates the protocol message; and
 - (2) the signed content cannot be interpreted otherwise
- We type signed contents with a refinement that specifies the different possible interpretations of the signature
 - For signing either XML requests or XML responses, we use

```
type verified = x:signinfo{
  (∀id, b.(Mem(IdHdr(id),x) ∧ Mem(RequestBody(b),x))
    ⇒ Request(id,b) )
  ∧ (∀id, req, b.(Mem(IdHdr(id),x) ∧ Mem(ResponseBody(b),x)
    ∧ Mem(InReplyTo(req),x)) ⇒ Response(id,req,b) ) }
```

Context, discussion

- RCF is an assembly of standard parts, generalizing ad hoc constructions in language-based security
 - **FPC** (Plotkin 1985, Gunter 1992) – core of ML and Haskell
 - Concurrency in style of the **pi-calculus** (Milner, Parrow, Walker 1989) but for a lambda-calculus (like 80s languages PFL, Poly/ML, CML)
 - Formal crypto derived by coding up **seals** (Morris 1973)
 - Security specs via logical **assume/assert** (Floyd, Hoare, Dijkstra 1970s), generalizing eg correspondences (Woo and Lam 1992)
 - Typing with dependent functions, pairs, subtyping (Cardelli 1988), and **refinement types** (Pfenning 1992, ...) aka **predicate subtyping**
 - **Public/tainted kinds** to track data that may flow to or from the opponent, as in Cryptyc (Gordon, Jeffrey 2002)
- Our experimental approach is to target existing languages & tools
 - Checker for existing language (F#) with codebase of security-critical code
 - Refinement types based on FOL, not bespoke authorization logic or type-and-effect system, to benefit from general-purpose verification tools

Summary

- We use formulas as computational effects to integrate program logics and type systems, with applications to security.
 - We embed formulas using refinement types
 - We represent active opponents as Un-typed contexts using subtyping
 - We encode symbolic cryptography using typed seals
- We obtain a first tool for verifying implementations of security policies and protocols by typing source code
 - More scalable and flexible than our prior translation to ProVerif
- We are experimenting with larger applications and examples
 - Concrete language design
 - Type inference, or type compilation
- <http://research.microsoft.com/F7>