

Secure Distributed Components



Gilles Barthe, Evmorfia-Iro Bartzia, Karthikeyan Bhargavan, Cédric Fournet, Benjamin Grégoire, James J. Leifer, Jean-Jacques Lévy, Francesco Zappa Nardelli, Alfredo Pironti, Jérémy Planul, Tamara Rezk, Pierre-Yves Strub

Cryptographic Protocols Go Wrong

Both designs and implementations have bugs

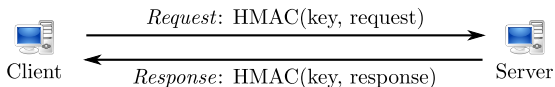
- Most standards got it wrong first (SSL, SSH, IPSEC, 802.11)
- Implementation details matter!

Secure Distributed Computations and their Proofs

- Protocol specifications remain largely informal
 - They focus on message formats and interoperability,
 - not on local enforcement of security properties
- Models are short, abstract, hand-written
 - They ignore large functional parts of implementations
 - Their formulation is driven by verification techniques
 - It is easy to write models that are safe but dysfunctional
- Specs, models, and implementations drift apart
 - Informal synchronization involves painful code reviews
 - How to keep track of implementation changes?

- Our approach:
 - We automatically extract models from protocol code
 - We develop models as executable code too (reference implementations)
- Executable code is more detailed than models
 - Some functional aspects can be ignored for security
 - Model extraction can safely erase those aspects
- Executable code has better tool support
 - Types, compilers, debuggers, libraries, testing, verification tools

The RPC Protocol



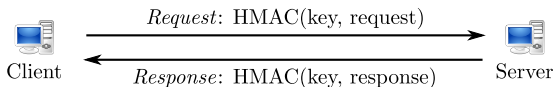
A simple RPC protocol:

- two roles: *client* and *server*
- a population of principals: *a*, *b*, *c*, ...

Security goals:

- if *b* accepts a request *s* from *a*,
then *a* has indeed sent this request to *b*
- if *a* accepts a response *t* from *b*,
then *b* has indeed sent *t* in response to *a*'s request

The RPC Protocol

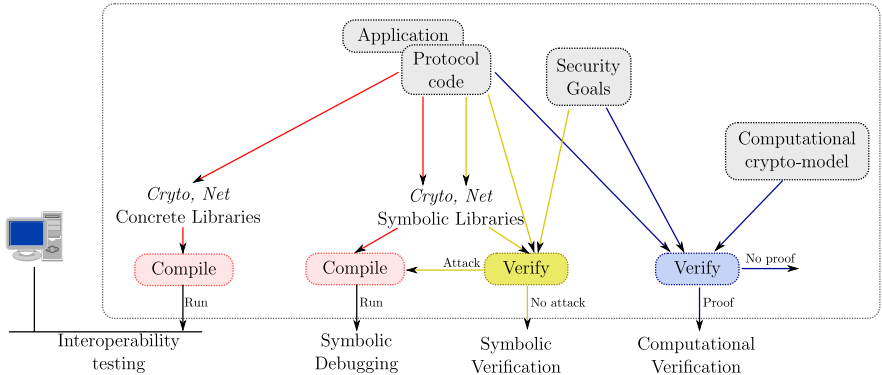


- Protocol uses message authentication codes (MACs)
- Multiple concurrent instances of the protocols
- Keys and principals may get compromised
- The adversary controls the network

Is this program secure ?

- Can the opponent cause our authentication goals to fail?
- What if the opponent sent a forged message to the server?
- What if the MAC key is compromised?

Verifying Protocol Code



Programming Language: F#

"Combining the strong typing, scripting and productivity of ML with the efficiency, stability, libraries, cross-language working and tools of .NET."



- Interoperability with production code
- Great for research & prototyping
- Clean strongly-typed semantics
- Modular programming based on strong interfaces
- Algebraic data types with pattern matching
 - useful for cryptography & message formats

Example: access control for files

Untrusted code may call a **trusted** library

Trusted code expresses security policy with **assumes** and **asserts**

Each policy violation causes an assertion failure.

We **statically** prevent any assertion failures by typing.

```
type facts =  
  CanRead of string | CanWrite of string  
  
let read file = assert (CanRead(file)); ...  
  
let delete file = assert (CanWrite(file)); ...  
  
let pwd = "C:/etc/passwd"  
let tmp = "C:/tmp/tempfile"  
  
assume CanWrite(tmp)  
assume  $\forall x, \text{CanWrite}(x) \rightarrow \text{CanRead}(x)$ 
```

```
let untrusted () =  
  let v1 = read tmp in // ok, by policy  
  let v2 = read pwd in // assertion fails  
  ...
```

```
Typechecking failed at acls.fs  
Error: cannot establish formula CanRead(pwd)
```

Example: access control for files

```
val read: file:string{CanRead(file)} -> string
val delete: file:string{CanWrite(file)} -> string
val publicfile: file:string{CanRead(file)} -> unit{PublicFile(file)}
```

- **Preconditions** express access control requirements
- **Postconditions** express results of validation
- We **typecheck** partially trusted code
 - guarantee that all preconditions hold at runtime

F7: refinement typechecking for F#

We write extended interfaces

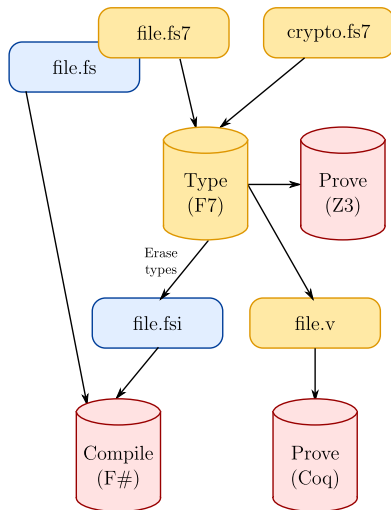
- We typecheck implementations
- We generate `.fsi` interfaces by erasure from `.fs7`

We do some type inference

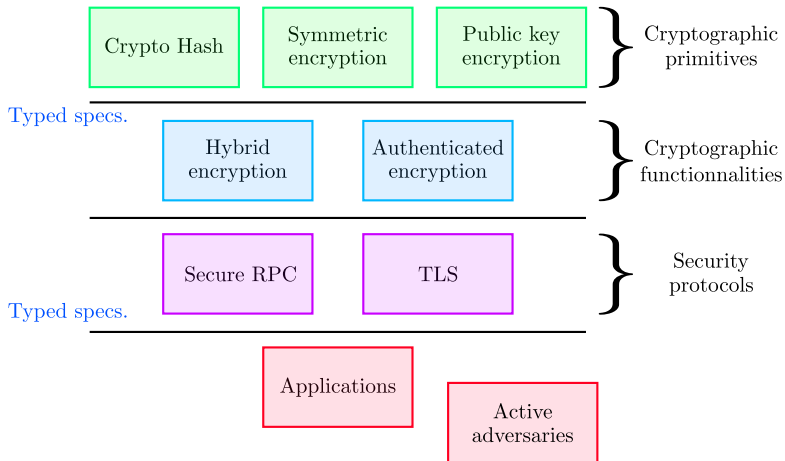
- Plain F# types as usual
- Refinements require annotations

We call Z3, an SMT prover, on each proof obligation

We can also generate Coq proof obligations Selected interactive proofs



Modular Symbolic Verification



Computational F7

```
type key
type bytes = string
type text = bytes
type mac = bytes

predicate authentic = Msg of text

val genkey : unit -> key
val mac : key -> t:text{Msg(t)} -> mac
val verify :
  key -> text -> mac -> b:bool{b => Msg(t)}
```

All verified messages
are authentic

```
let rng = new RNGCryptoServiceProvider ()

let randomBytes n =
  let b = Bytearray.make n in
  rng.GetBytes b; Key b

let genkey () = randomBytes 32 (* 256 bits *)

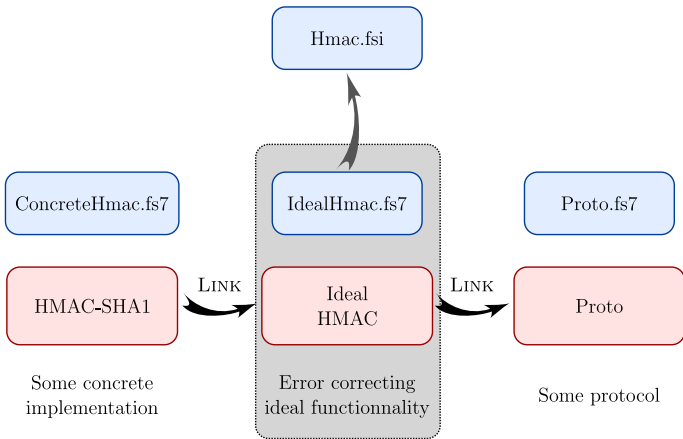
let mac (Key k) (r : text) =
  base64 ((new HMACSHA1(k)).ComputeHash (utf8 r))

let verify k t sv = (mac k t = sv)
```

Many collisions

Cannot match
given interface

MACs: interfaces and implementations

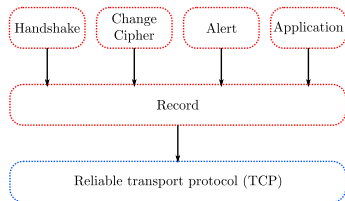


- Ideal fonctionnality is *indistinguishable* from concrete one

We are formally proving our tools in the Coq proof assistant:

- Formal study of the F7 meta-theory
- Formal study of the computational crypto-proofs
 - Formalization of the framework logic
 - Proof of the crypto-primitives in the framework
- Certification of the type-checker
 - Self-certification technique
 - The type-checker generates a Coq proof of its own correctness

Use Case: Transport Layer Security



- 1994 Netscape's Secure Sockets Layer
- 1994 SSL2 (known attacks)
- 1995 SSL3 (fixed them)
- 1999 IETF's TLS1.0 (RFC2246)
- 2006 TLS1.1 (RFC4346)
- 2008 TLS1.2 (RFC5246)

Between TCP and Application
Itself a layered protocol:

Record protocol: provides a private and reliable connection

Handshake protocol: authenticates one or both parties, negotiates security parameters, establishes connection keys for the *record protocol*

Resumption protocol: abbreviated version of Handshake: establishes connection keys from previous handshake

Our TLS implementation is interoperable

Demo