

Modular Verification of Security Protocol Code by Typing

Karthikeyan Bhargavan

Cédric Fournet

Andrew D. Gordon

Microsoft Research

<http://research.microsoft.com/f7>

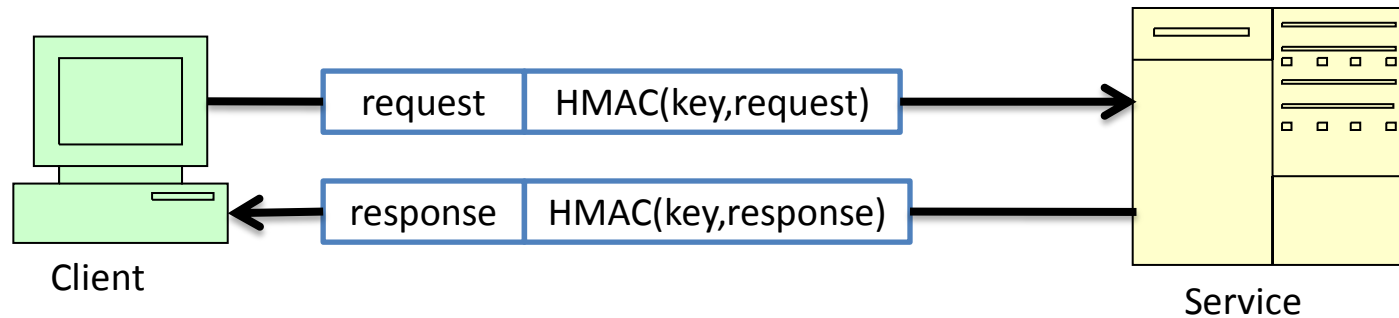
Problem of Verifying Protocol Code

- The problem of vulnerabilities in security protocols is remarkably resistant to the success of formal methods
- Perhaps, tools for verifying the actual protocol code will help
 - Csur (VMCAI'05), fs2pv (CSF'06), F7 (CSF'08), Aspier (CSF'09), etc etc
- Currently, fs2pv most developed, but hitting a wall
 - Translates libraries and protocol code from F#/OCaml to ProVerif, Blanchet's state-of-the-art verifier for crypto protocols
 - ProVerif does whole-program analysis of code versus symbolic attacker
 - Long, unpredictable run times on F# reference implementations of Cardspace (ASIACCS'08), TLS (CCS'08)
- Instead, we have a new scalable analysis for the fs2pv codebase
 - Using **refinement types** based on **first-order formulas**
 - Checked **compositionally** with a **general-purpose** verifier

Our new method:

Invariants for Cryptographic Structures

- (1) We model cryptographic structures as elements of a symbolic algebra, e.g. $MAC(k, M)$.
- (2) We use a “Public” predicate and events keep track of protocols.
 - $Pub(x)$ holds when the value x is known to the adversary.
 - $Request(a, b, x)$ holds when a intends to send message x to b .
- (3) We define logical invariants on cryptographic structures.
 - $Bytes(x)$ holds when the value x appears in the protocol run.
 - $KeyAB(k_{ab}, a, b)$ holds when key k_{ab} is shared between a and b .
 - After verifying the MAC (if no principals are compromised),
 $KeyAB(k_{ab}, a, b) \wedge Bytes(hash\ k_{ab}\ x) \implies Request(a, b, x)$.
- (4) We verify that the protocol code maintains these invariants (by typing)
 - $KeyAB(k_{ab}, a, b) \wedge Request(a, b, x)$ is a precondition for computing $hash\ k_{ab}\ x$



SAMPLE PROTOCOL

AUTHENTICATED RPC

We obtain no guarantee of request/response correlation:

Client sends request1, request2 awaits replies

Service computes and sends response1, response2

Opponent swaps response1, response2

Client successfully checks MACs, and acts on the swapped responses

Informal Description

1. $a \rightarrow b : \text{utf8 } s \mid (\text{hmacsha1 } k_{ab} (\text{request } s))$
2. $b \rightarrow a : \text{utf8 } t \mid (\text{hmacsha1 } k_{ab} (\text{response } s t))$

We design and implement authenticated RPCs over a TCP connection.

We have two roles, client and server, and a population of principals, $a b c \dots$

Our 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.

We use message authentication codes (MACs) computed as keyed hashes, such that each symmetric key k_{ab} is associated with (and known to) the pair of principals a and b .

There are multiple concurrent RPCs between any number of principals.

The adversary controls the network. Keys and principals never compromised.

Logical Specification

1. $a \rightarrow b : \text{utf8 } s \mid (\text{hmacsha1 } k_{ab} (\text{request } s))$
2. $b \rightarrow a : \text{utf8 } t \mid (\text{hmacsha1 } k_{ab} (\text{response } s t))$

Event predicates record the main steps of the protocol:

- $\text{Request}(a,b,s)$ before a sends message 1;
- $\text{Response}(a,b,s,t)$ before b sends message 2;
- $\text{KeyAB}(k,a,b)$ before issuing a key k associated with a and b .

Authentication goals are stated in terms of events:

- $\text{RecvRequest}(a,b,s)$ after b accepts message 1;
- $\text{RecvResponse}(a,b,s,t)$ after a accepts message 2;

where the predicates RecvRequest and RecvResponse are defined by

$$\forall a,b,s. \text{RecvRequest}(a,b,s) \Leftrightarrow \text{Request}(a,b,s)$$

$$\forall a,b,s,t. \text{RecvResponse}(a,b,s,t) \Leftrightarrow (\text{Request}(a,b,s) \wedge \text{Response}(a,b,s,t))$$

F# Implementation

1. $a \rightarrow b : \text{utf8 } s \mid (\text{hmacsha1 } k_{ab} (\text{request } s))$
2. $b \rightarrow a : \text{utf8 } t \mid (\text{hmacsha1 } k_{ab} (\text{response } s t))$

Our F# implementation of the protocol:

```
let mkKeyAB a b = let k = hmac_keygen() in assume (KeyAB(k,a,b)); k
let request s = concat (utf8(str "Request")) (utf8 s)
let response s t = concat (utf8(str "Response")) (concat (utf8 s) (utf8 t))
```

```
let client (a:str) (b:str) (k:keyab) (s:str) =
    assume (Request(a,b,s));
    let c = Net.connect p in
    let mac = hmacsha1 k (request s) in
    Net.send c (concat (utf8 s) mac);
    let (pload',mac') = iconcat (Net.recv c) in
    let t = iutf8 pload' in
    hmacsha1Verify k (response s t) mac';
    assert(RecvResponse(a,b,s,t))
```

```
let server(a:str) (b:str) (k:keyab) : unit =
    let c = Net.listen p in
    let (pload,mac) = iconcat (Net.recv c) in
    let s = iutf8 pload in
    hmacsha1Verify k (request s) mac;
    assert(RecvRequest(a,b,s));
    let t = service s in
    assume (Response(a,b,s,t));
    let mac' = hmacsha1 k (response s t) in
    Net.send c (concat (utf8 t) mac')
```

Test

1. $a \rightarrow b : \text{utf8 } s \mid (\text{hmacsha1 } k_{ab} (\text{request } s))$
2. $b \rightarrow a : \text{utf8 } t \mid (\text{hmacsha1 } k_{ab} (\text{response } s t))$

The messages exchanged over TCP are:

Connecting to localhost:8080

Sending {BgAyICsgMj9mhJa7iDACw3Rrk...} (28 bytes)

Listening at ::1:8080

Received Request 2 + 2?

Sending {AQA0NccjcuL/W0aYS0GGtOtPm...} (23 bytes)

Received Response 4

Opponent as F# Program

Our security goals should hold in the face of an *opponent*

- able to generate arbitrary protocol instances
- able to send, receive, and do crypto on network messages.

We model these abilities as an interface to some abstract types:

Opponent Interface (excerpts):

val *send*: **conn** → **bytespub** → **unit**

val *recv*: **conn** → **bytespub**

val *hmacsha1* : **keypub** → **bytespub** → **bytespub**

val *hmacsha1Verify* : **keypub** → **bytespub** → **bytespub** → **unit**

val *setup*: **strpub** → **strpub** → (**strpub** → **unit**) * (**unit** → **unit**)

The implementation

```
let setup (a:str) (b:str) =  
  let k = mkKeyAB a b in  
  (fun s → client a b k s),  
  (fun _ → server a b k)
```

Security Theorem

Let I_L be the opponent interface for our library.

Let I_R be the opponent interface for our protocol (the *setup* function).

Let an *opponent* be any O with $I_L, I_R \vdash O : \mathbf{unit}$.

An expression is *semantically safe* when every executed assertion logically follows from previous assumptions.

Let X be the composition of our library and protocol code.

Theorem 1 (Authentication for the RPC Protocol)

For any opponent O , the composition $X[O]$ is semantically safe.

F7: Refinement Types for F#

- We use extended interfaces (.fs7)
 - We typecheck implementations
 - Interfaces include types refined with **first-order formulas**
 - Only libraries security-specific
- We support a large subset of F#
 - Algebraic types, records, patterns, refs
 - Concurrency

$n : \text{int}\{n > 0\}$

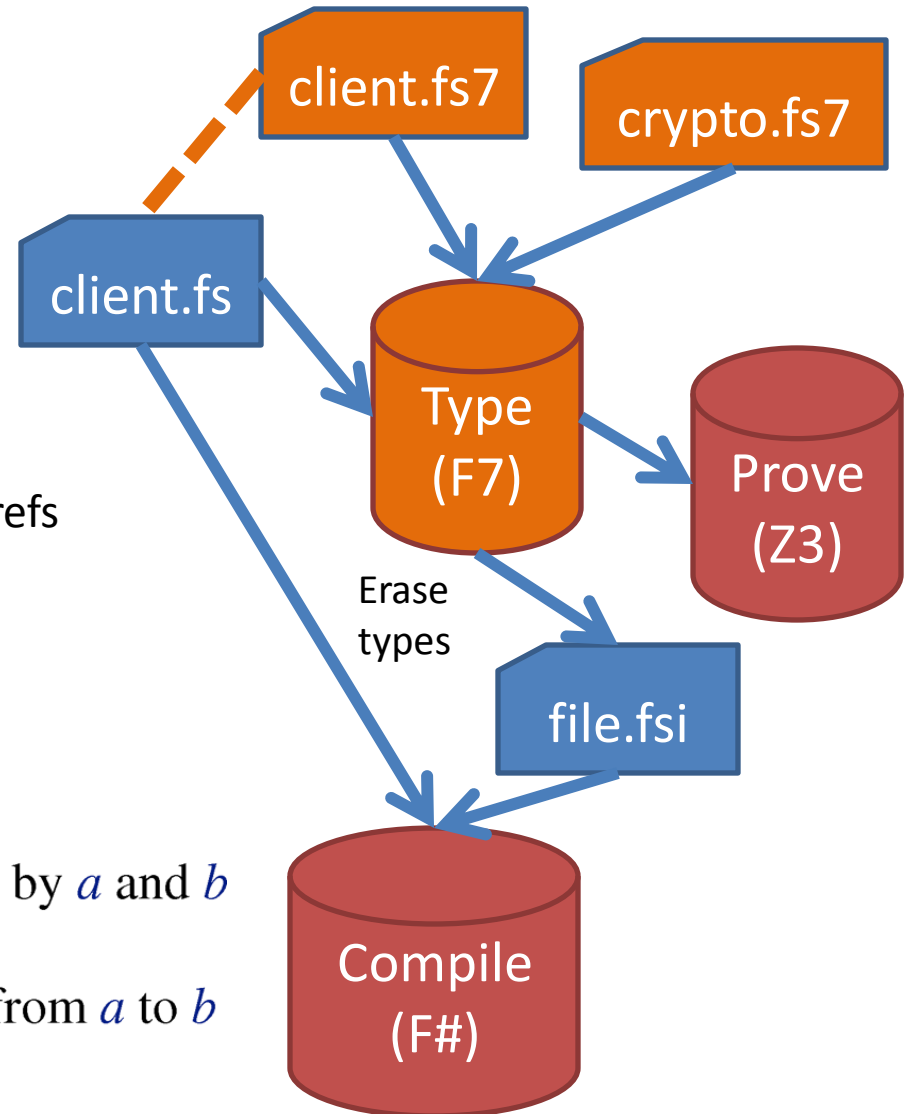
is the type of positive integers

$k : \text{bytes}\{KeyAB(k, a, b)\}$

is the type of byte arrays used as keys by a and b

$x : \text{str}\{Request(a, b, x)\}$

is the type of strings sent as requests from a to b



Security by Typing

To prove the theorem, we type code against the library interface.

Correctness of the code relies on properties of the library exported as logical pre- and post-conditions expressed with refinement types

$MKey(k)$ means k is a key for forming MACs

$MACSays(k, b)$ is the logical payload of a MAC of bytes b with key k

$Pub(b)$ means the opponent may know b

$IsMAC(h, k, b)$ means h is indeed a hash of b with k

Some of the Refinement Types: MACs (Protocol)

```
private val hmac_keygen: unit → k:key {MKey(k)}
```

```
val hmacsha1:
```

```
  k:key →
```

```
  b:bytes { (MKey(k) ∧ MACSays(k,b)) } →
```

```
  h:bytes { IsMAC(h,k,b) }
```

```
val hmacsha1Verify:
```

```
  k:key {MKey(k)} → b:bytes → h:bytes → unit {IsMAC(h,k,b)}
```

Security by Typing

To prove the theorem, we type code against the library interface.

Correctness of the code relies on properties of the library exported as logical pre- and post-conditions expressed with refinement types

$MKey(k)$ means k is a key for forming MACs

$MACSays(k, b)$ is the logical payload of a MAC of bytes b with key k

$Pub(b)$ means the opponent may know b

$IsMAC(h, k, b)$ means h is indeed a hash of b with k

Some of the Refinement Types: MACs (Protocol and Opponent)

```
private val hmac_keygen: unit → k:key {MKey(k)}
```

```
val hmacsha1:
```

```
  k:key →
```

```
  b:bytes { (MKey(k) ∧ MACSays(k,b)) ∨ (Pub(k) ∧ Pub(b)) } →
```

```
  h:bytes { IsMAC(h,k,b) ∧ (Pub(b) ⇒ Pub(h)) }
```

```
val hmacsha1Verify:
```

```
  k:key {MKey(k) ∨ Pub(k)} → b:bytes → h:bytes → unit {IsMAC(h,k,b)}
```

Opponent's abstract type $\text{bytespub} \triangleq x : \text{bytes} \{Pub(x)\}$

Some of the Formulas Assumed during Typechecking:

(KeyAB MACSays)

$$\forall a,b,k,m. \text{KeyAB}(k,a,b) \Rightarrow (\text{MACSays}(k,m) \Leftrightarrow \\ ((\exists s. \text{Request}(a,b,s) \wedge m = \text{request}(s)) \vee \\ (\exists s,t. \text{Response}(a,b,s,t) \wedge m = \text{response}(s,t))))$$

(IsMAC MACSays)

$$\forall h,k,b. \text{IsMAC}(h,k,b) \wedge \text{MKey}(k) \Rightarrow \text{MACSays}(k,b)$$

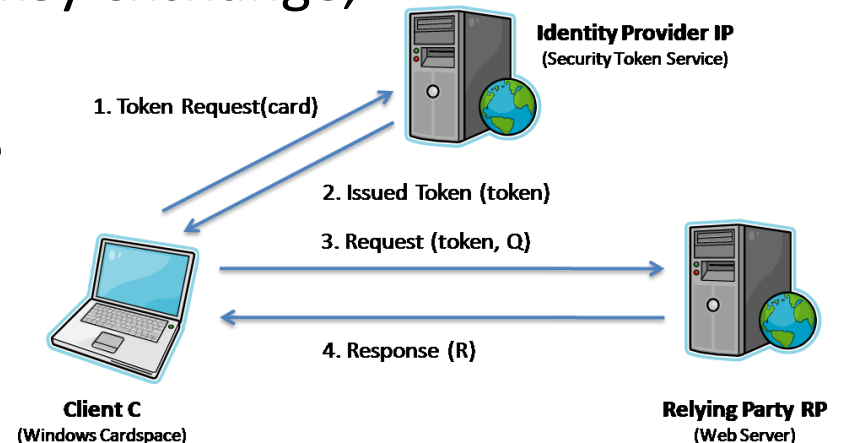
(KeyAB MACSays) is a *definition* for the library predicate *MACSays*.
It states the intended usage of keys in this protocol.

(IsMAC MACSays) is a *theorem*: existence of a MAC with key k
implies the logical payload *MACSays*(k, b).

Using these assumptions, F7 typechecks our protocol code.
This automatically completes our protocol verification.

What Else Is In The Paper?

- Theory of *refined modules* to justify independent type-checking of F# libraries
 - Inductive definitions expressed as Horn clauses in F7 interface files
- Discussion of F7 and improvements since CSF'08
- Comprehensive libraries for standard crypto
 - Key management, authenticated encryption, hybrid encryption, derived keys, endorsing signatures
- Example protocols: Otway-Rees key exchange, secure conversations
- Case study: Windows CardSpace



Experimental Evaluation

Example	F# Program		F7 Typechecking		Fs2PV Verification	
	Modules	Lines of Code	Interface	Checking Time	Queries	Verifying Time
Cryptographic Patterns	1	158 lines	100 lines	17.1s	4	3.8s
Basic Protocol (Section 2)	1	76 lines	141 lines	8s	4	4.1s
Otway-Rees (Section 4.2)	1	265 lines	233 lines	1m.29.9s	10	8m 2.2s
Otway-Rees (No MACs)	1	265 lines	-	(Type Incorrect)	10	2m 19.2s
Secure Conversations (Section 4.3)	1	123 lines	111 lines	29.64s	-	(Not Verified)
Web Services Security Library	5	1702	475	48.81s	(Not Verified Separately)	
X.509-based Client Auth (Section 5.1)	+ 1	+ 88 lines	+ 22 lines	+ 10.8s	2	20.2s
Password-X.509 Mutual Auth(Section 5.2)	+ 1	+ 129 lines	+ 44 lines	+ 12s	15	44m
X.509-based Mutual Auth	+ 1	+ 111 lines	+ 53 lines	+ 10.9s	18	51m
Windows CardSpace (Section 5.3)	1	1429 lines	309 lines	6m3s	6	66m 21s

Table 1. Verification Times and Comparison with ProVerif

- Above a threshold, typing faster than whole program analysis, and errors in typing much easier to localize
- But, F7 needs hand-written logical theories, tedious to check
 - Candidate for mechanized proof with interactive theorem prover
- But, F7 relies entirely on Z3 for deduction during typechecking
 - Some appeal to tactics, cf PVS, HOL/Boogie, etc, would probably help

Some Conclusions

- F7 achieves flexible and scalable verification of security code
 - can beat sophisticated whole-program analysis on realistic examples
- The only domain-specific part of our theory is the predicate library (Pub, IsMAC, etc), so we hope it will easily port
 - to other refinement-type checkers for functional code
 - and even to verifiers for imperative languages
- Our community is making real progress on verified software
 - “Verified Software Initiative” aka “The War on Error”
 - Our work contributes a hybrid approach to verifying code in existing functional languages: (1) prove logical theory by hand or with proof assistant, then (2) link to code with refinement types

<http://research.microsoft.com/f7>

QUESTIONS?