Friday 4th 09:00-09:50

1

Cryptographic and Probabilistic Programming

Andrew D. Gordon
Microsoft Research and University of Edinburgh
@AndrewDGordon #fosad2015

Agenda and Goals

- Lecture 1: Problem of Verifying Cryptographic Protocols
- Lecture 2: A Formal Calculus of Refinement Types
- Lecture 3: Verified Cryptographic Programs for Protocols
- Lecture 4: Probabilistic Programming and Security

- My goal in lectures 1-3 is to motivate, explain the basic principles, and give examples, of a line of work on verifying the actual implementation code of cryptographic protocols.
- My goal in the final lecture is to introduce the field of probabilistic programming and discuss various security-related applications.

Credits #fosad2015

- Mihhail Aizatulin, Andrew D. Gordon, Jan Jürjens: Extracting and verifying cryptographic models from C protocol code by symbolic execution. ACM Conference on Computer and Communications Security 2011:331-340
- Mihhail Aizatulin, Andrew D. Gordon, Jan Jürjens: Computational verification of C protocol implementations by symbolic execution. ACM Conference on Computer and Communications Security 2012:712-723
- Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Sergio Maffeis: Refinement types for secure implementations. ACM Trans. Program. Lang. Syst. (TOPLAS) 33(2):8 (2011)
- Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub: Implementing TLS with Verified Cryptographic Security. IEEE Symposium on Security and Privacy 2013:445-459
- Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon: Modular verification of security protocol code by typing. POPL 2010:445-456
- Cédric Fournet, Karthikeyan Bhargavan, Andrew D. Gordon: Cryptographic Verification by Typing for a Sample Protocol Implementation. FOSAD 2011:66-100
- François Dupressoir, Andrew D. Gordon, Jan Jürjens, David A. Naumann: Guiding a general-purpose C verifier to prove cryptographic protocols. Journal of Computer Security (JCS) 22(5):823-866 (2014)
- Andrew D. Gordon, Cédric Fournet: Principles and Applications of Refinement Types. Logics and Languages for Reliability and Security 2010:73-104
- Andrew D. Gordon, Thore Graepel, Nicolas Rolland and, Claudio V. Russo, Johannes Borgström, John Guiver: Tabular: a schema-driven probabilistic programming language. POPL 2014:321-334

Problem of Verifying Cryptographic Protocols

Cryptographic and Probabilistic Programming, Part 1

Cryptographic Protocols

- Principals communicate over an untrusted network
  - Our focus is on Internet protocols, but same principles apply to banking, payment, and telephony protocols

- A range of security and privacy objectives is possible
  - Message confidentiality – against release of contents
  - Identity protection – against release of principal identities
  - Message authentication – against impersonated access
  - Message integrity – against tampering
  - Message correlation – that a response matches a request
  - Message freshness – against replays of old messages

- To achieve these goals, principals rely on applying cryptographic algorithms to parts of messages, but also on including message identifiers, nonces (unpredictable quantities), and timestamps

Cryptographic protocols go wrong

- Historically, one keeps finding simple attacks against protocols
  - even carefully-written, widely-deployed protocols, even a long time after their design & deployment
  - simple = no need to break cryptographic primitives
- Why is it so difficult?
  - breaking functional abstractions
  - concurrency + distribution + cryptography
    - Little control on the runtime environment
  - active attackers
    - hard to test
  - implicit assumptions and goals
    - Authenticity, secrecy

The Needham-Schroeder problem

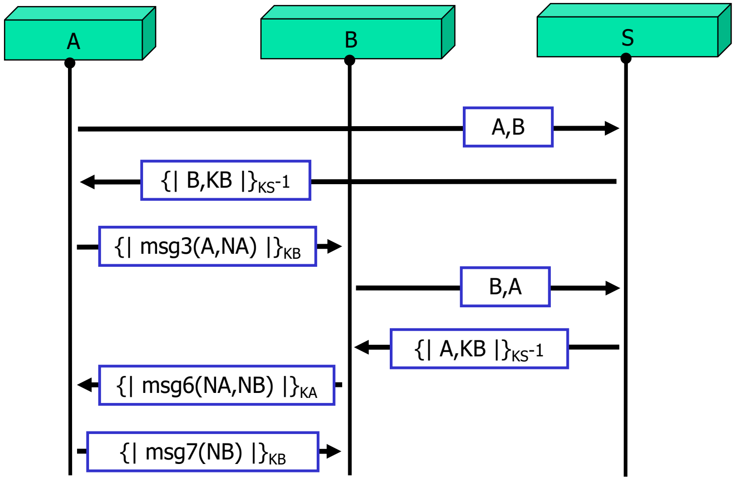
In **Using encryption for authentication in large networks of computers (CACM 1978)**, Needham and Schroeder didn't just initiate a field that led to widely deployed protocols like Kerberos, SSL, SSH, IPSec, etc.

They threw down a gauntlet.

"Protocols such as those developed here are prone to extremely subtle errors that are unlikely to be detected in normal operation.

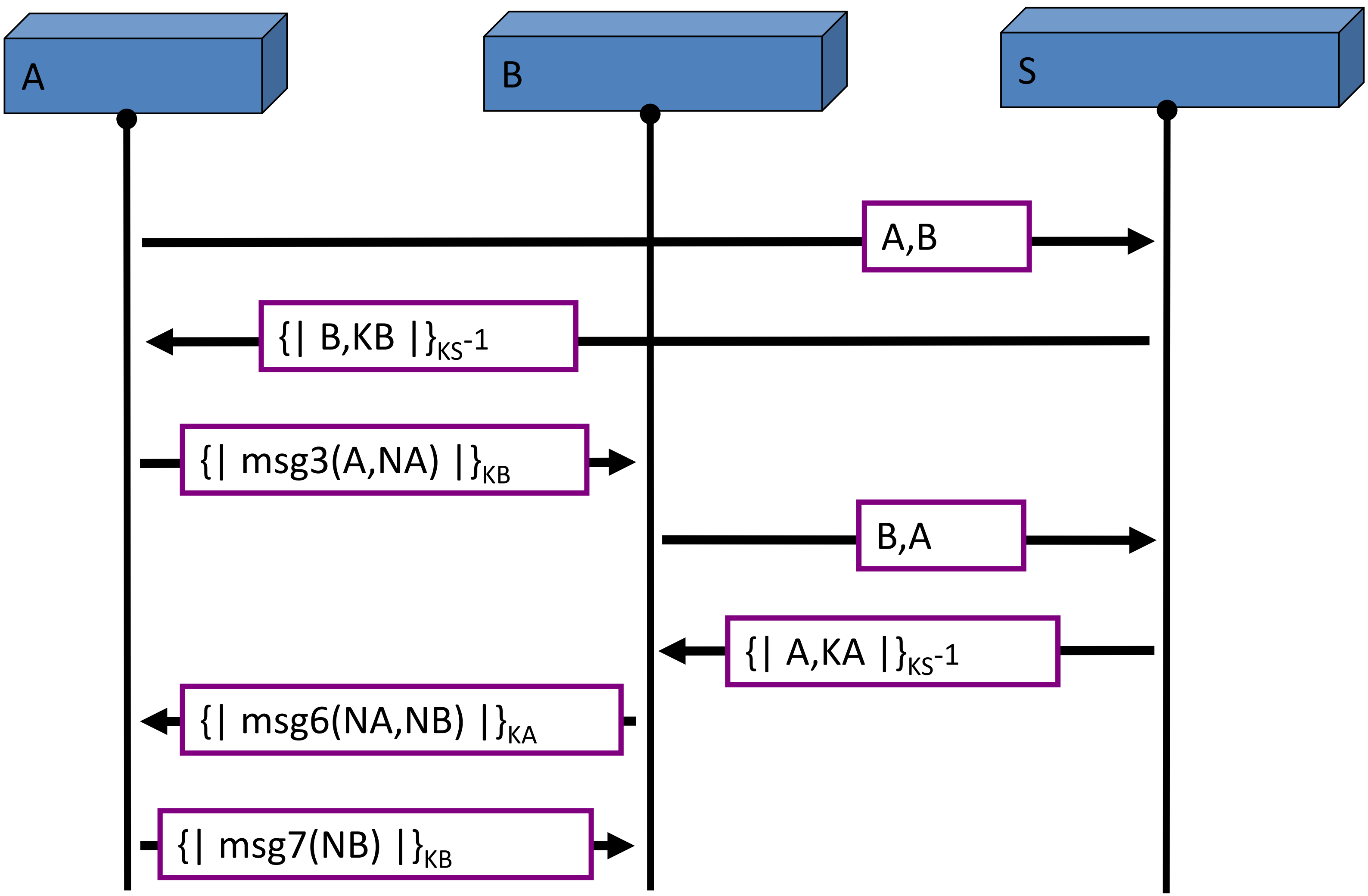
The need for techniques to verify the correctness of such protocols is great, and we encourage those interested in such problems to consider this area."

The Needham-Schroeder public-key authentication protocol (CACM 1978)



Principal A initiates a session with principal B
S is a trusted server returning public-key certificates eg {| A,KA $|\}_{KS}$-1
NA,NB serve as nonces to prove freshness of messages 6 and 7

The Needham-Schroeder public-key authentication protocol (CACM 1978)

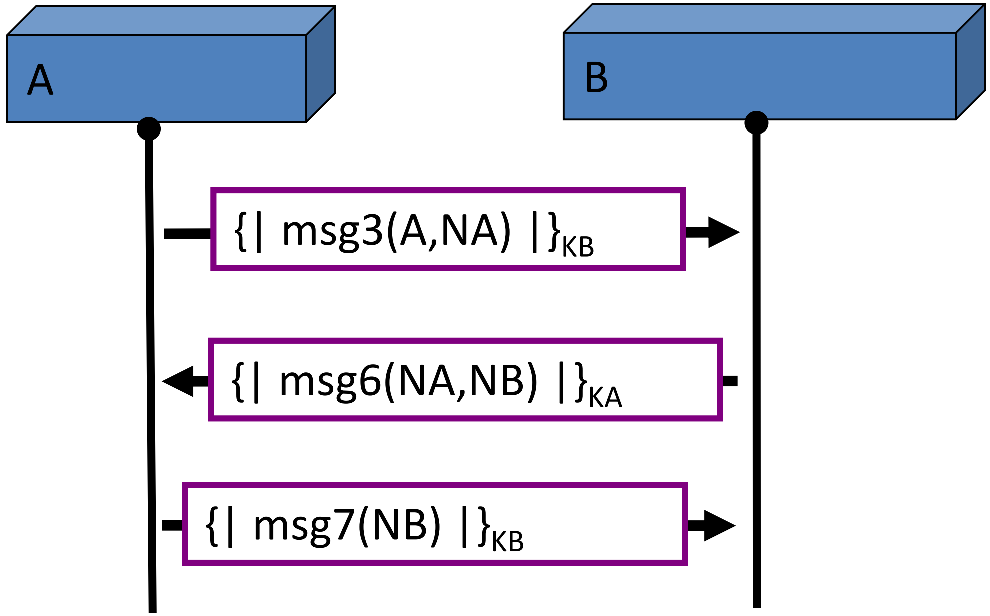


Principal A initiates a session with principal B

S is a trusted server returning public-key certificates eg {| A,KA |}$_{KS}$-1

NA,NB serve as nonces to prove freshness of messages 6 and 7

Assuming A knows KB and B knows KA, we get the core protocol:

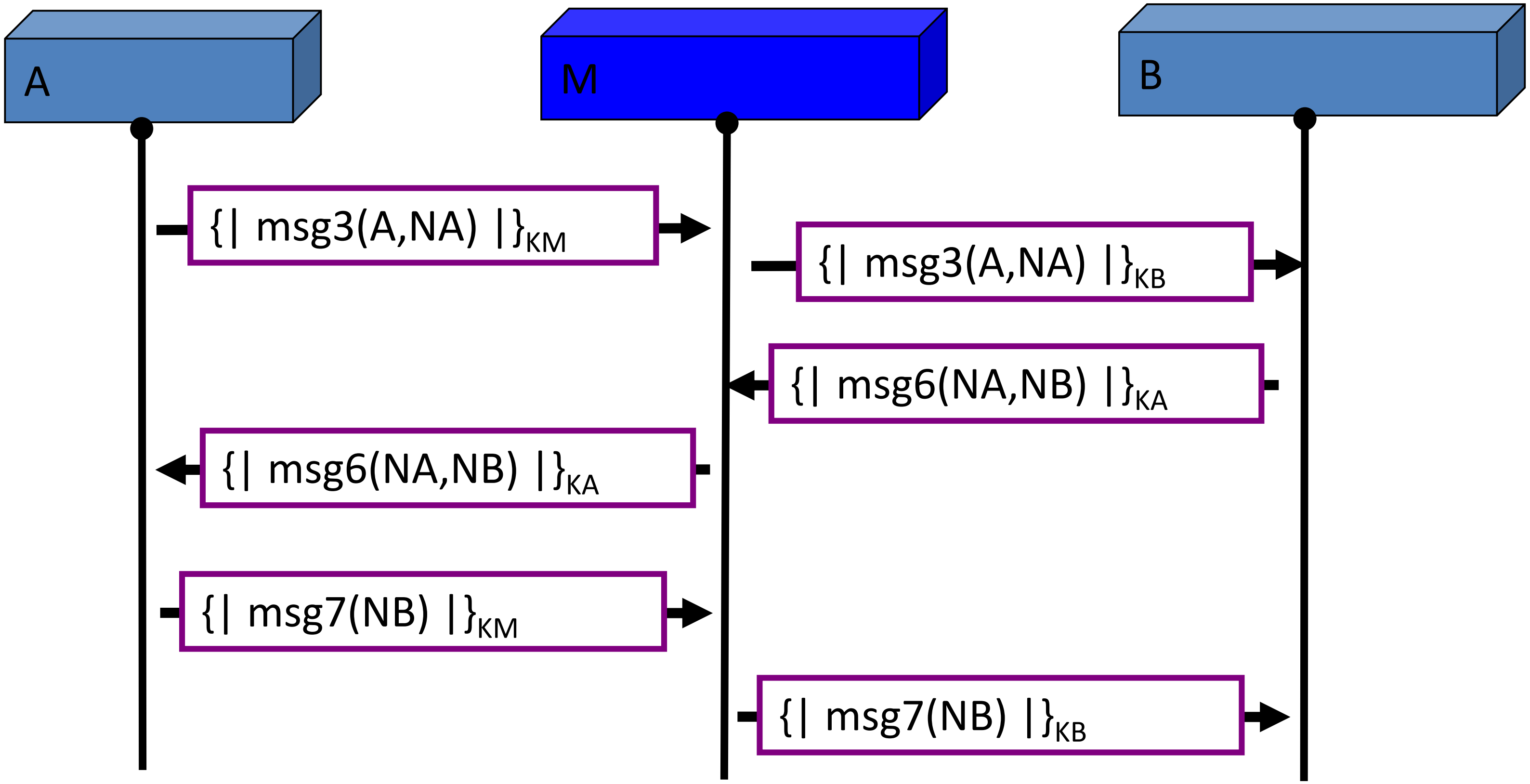


More precisely, the goals of the protocol are:

- After receiving message 6, A believes NA,NB shared just with B
- After receiving message 7, B believes NA,NB shared just with A

If these goals are met, A and B can subsequently rely on keys derived from NA,NB to efficiently secure subsequent messages

A certified user M can play a man-in-the-middle attack (Lowe 1995)

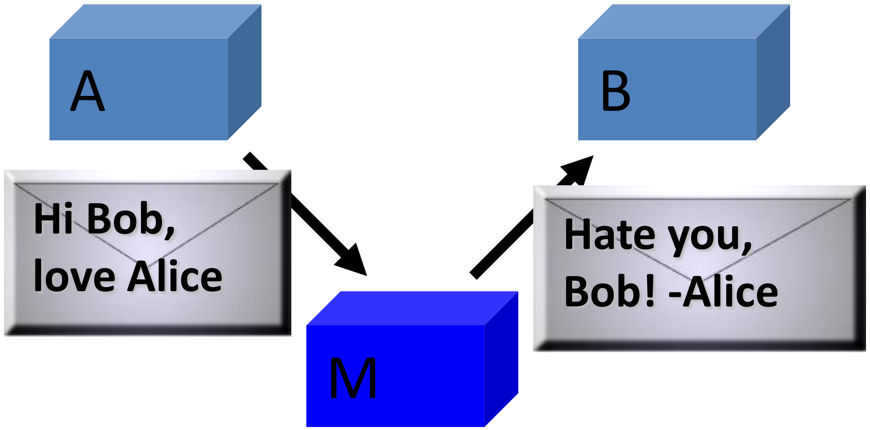


This run shows a certified user M can violate the protocol goals:

- After receiving message 6, A believes NA,NB shared just with M
- After receiving message 7, B believes NA,NB shared just with A

(Writing in the 70s, Needham and Schroeder assumed certified users would not misbehave; we know now they do.)

A brief history: 1978—

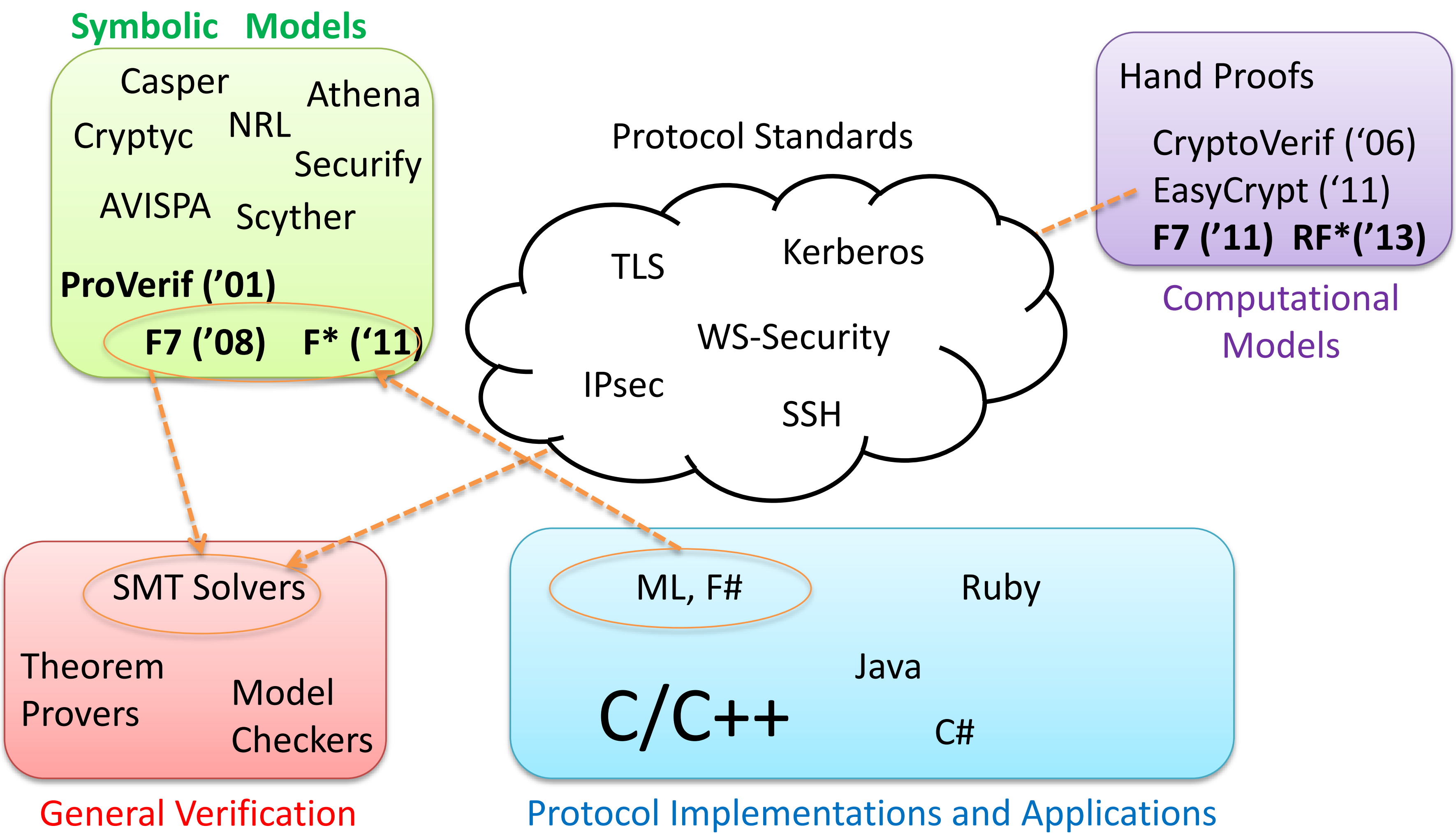


We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols.

Needham and Schroeder CACM (1978)

1978: N&S propose authentication protocols for “large networks of computers”
1981: Denning and Sacco find attack on N&S symmetric key protocol
1983: Dolev and Yao first formalize secrecy properties of NS threat model using formal algebra
1987: Burrows, Abadi, Needham invent authentication logic; incomplete, but useful
1994: Hickman, Elgamal invent SSL; holes in v1, v2, but v3 fixes these, very widely deployed
1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
1995: Abadi, Anderson, Needham, et al propose various informal “robustness principles”
1995: Lowe finds insider attack on N&S asymmetric protocol; rejuvenates interest in FMs
circa 2000: Several FMs for “D&Y problem”: tradeoff between accuracy and approximation
circa 2007: Many FMs developed; several deliver both accuracy and automation
2014: dozens of attacks against mainstream TLS implementations

Specs, code, and formal tools



Models: Formal vs Computational Cryptography

- Two approaches for verifying protocols and programs

  Symbolic models (Needham-Schroeder, Dolev-Yao, ... late 70's)
  - Structural view of protocols, using formal languages and methods
  - Many automated verification tools, scales to large systems

  Computational models (Yao, Goldwasser, Micali, Rivest, ... early 80's)
  - Concrete, algorithmic view, using probabilistic polynomial-time machines
  - New formal tools: CryptoVerif, Certicrypt, EasyCrypt

- Can we get the best of both worlds? Much ongoing work on computational soundness for symbolic cryptography

  (Abadi Rogaway, Backes Pfitzman Wagner, Warinschi,… mid 00's)
  - It works… with many mismatches, restrictions, and technicalities
  - At best, one still needs to verify protocols symbolically
- Can we directly verify real-world protocols ?

Models vs implementations

- 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 (testing & debugging is difficult)
- Specs, models, and implementations drift apart…
  - Even informal synchronization involves painful code reviews
  - How to keep track of implementation changes?

From code to model

- Our approach: we directly verify **reference implementations** treated as “giant” protocol models

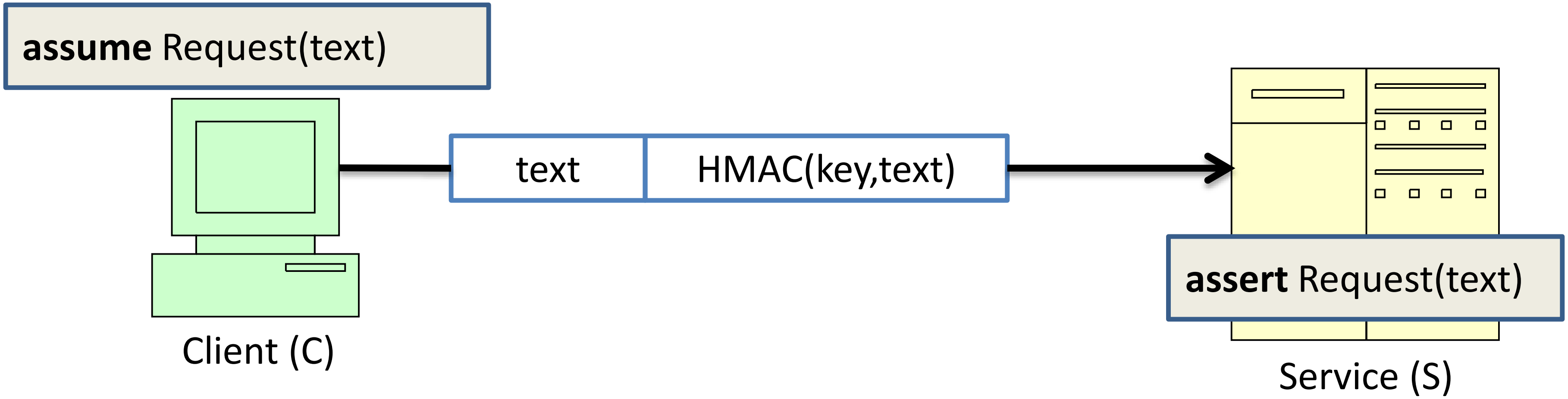
- Executable code is more detailed than models
  - Some functional aspects can be ignored for security
  - Model extraction can safely erase those aspects
- Executable code has better tool support
  - Types, compilers, debuggers, libraries, verification tools

Agenda for Rest of Lecture 1

- How to represent protocols and their correctness within a concurrent functional language (F#/OCaml):
  - Correspondence assertions as assume/assert
  - Message-passing concurrency as in the pi-calculus
  - Crypto modelled using Morris' seal abstraction
  - Protocol roles as functions (we'll see the code in action)
  - Opponent (attacker) is an arbitrary untyped expression
  - Correctness as robust program safety

- Overall, we reduce crypto protocol verification to a program verification problem

Example: Authenticated Message



- Security goal is simply authenticity, but not confidentiality or freshness
- Shows essence of problem, with simplifying assumptions
  - Assume one key, shared between two, fixed principals
  - Assume principals use keys only in compliance with protocol

Assume and Assert

- Suppose there is a global set of formulas, the **log**
- To evaluate **assume** *C*, add *C* to the log, and return ().
- To evaluate **assert** *C*, return ().
  - If *C* logically follows from the logged formulas, we say the assertion **succeeds**; otherwise, we say the assertion **fails**.
  - The log is only for specification purposes; it does not affect execution

- **assume** Foo(); **assert** Bar(); **assume** Foo()$\Rightarrow$Bar(); **assert** Bar()

- Our use of first-order logic predicates (like Foo()) generalizes conventional assertions (like **assert** i>0 in Hoare logic)
  - Such predicates usefully represent security-related concepts like roles, permissions, events, compromises

Symmetric Crypto

type α pickled `(*byte array representation of α*)`
val pickle : $(\alpha \rightarrow \alpha$ pickled$)$
val unpickle : $(\alpha$ pickled $\rightarrow \alpha)$

type α hkey `(*hash key*)`
type hmac `(*keyed hash*)`
val mkHKey : (unit $\rightarrow \alpha$ hkey)
val hmacsha1 : $(\alpha$ hkey $\rightarrow (\alpha$ pickled $\rightarrow$ hmac$))$
val hmacsha1Verify : $(\alpha$ hkey $\rightarrow (\beta$ pickled $\rightarrow$ (hmac $\rightarrow \alpha$ pickled$)))$

type α symkey `(*symmetric encryption key*)`
type enc `(*ciphertext*)`
val mkEncKey : (unit $\rightarrow \alpha$ symkey)
val aesEncrypt : $(\alpha$ symkey $\rightarrow (\alpha$ pickled $\rightarrow$ enc$))$
val aesDecrypt : $(\alpha$ symkey $\rightarrow$ (enc $\rightarrow \alpha$ pickled$))$

Morris' Seal Abstraction

A *seal* k for a type T is a pair of functions:

- the *seal function for* k, of type $T \rightarrow \mathsf{Un}$
- the *unseal function for* k, of type $\mathsf{Un} \rightarrow T$

The type Un consists of untrusted, public bitstrings known to the attacker.

The seal function, applied to M, wraps up its argument as a *sealed value*, written $\{M\}_k$.
There is no other way to construct $\{M\}_k$.

The unseal function, applied to $\{M\}_k$, unwraps its argument and returns M.
There is no other way to retrieve M from $\{M\}_k$.

Sealed values are opaque; in particular, the seal k cannot be retrieved from $\{M\}_k$.

To implement a seal k, we maintain a list of pairs $[(M_1, a_1); \ldots; (M_n, a_n)]$.
The list records all the values M_i that have so far been sealed with k.
Each a_i is a fresh name representing the sealed value $\{M_i\}_k$.

J.H. Morris, Jr, *Protection in Programming Languages*, CACM 1973

Coding Crypto Library with Seals

```
type α hkey = HK of (α pickled) Seal
type hmac = HMAC of Un

let mkHKey ():α hkey = HK (mkSeal "hkey")
let hmacsha1 (HK key) text = HMAC (fst key text)
let hmacsha1Verify (HK key) text (HMAC h) =
  let x:α pickled = snd key h in
    if x = text then x else failwith "hmac verify failed"
```

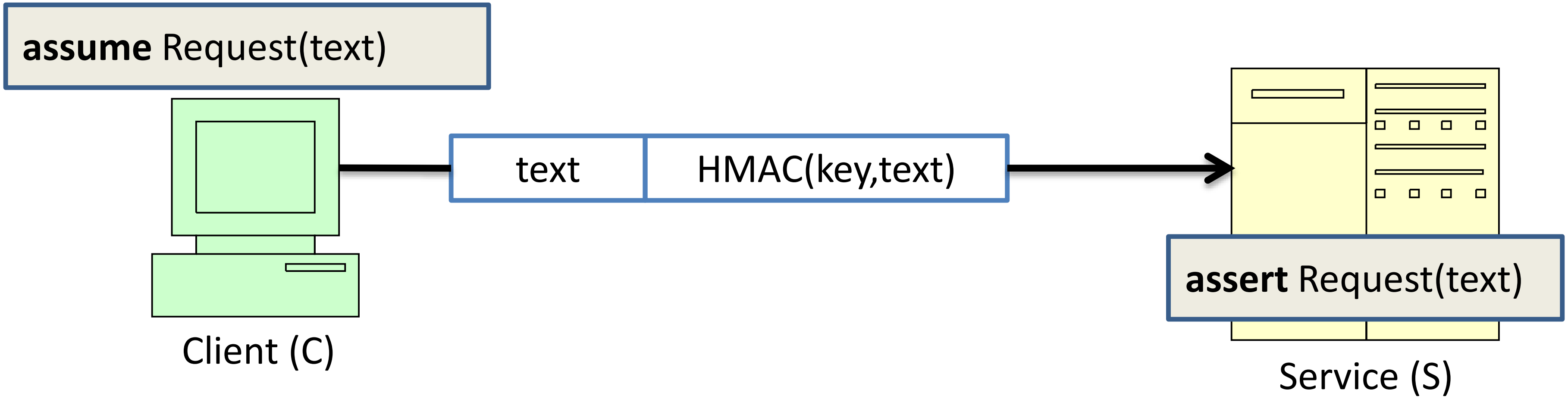
Exercise: Implement shared key encryption, public-key encryption, and digital signatures using seals.

```
type α symkey = Sym of α pickled Seal
type enc = AES of Un
```

Limits of Symbolic Models

- Dolev-Yao style **symbolic models** (including seals) have effective proof techniques, but make strong assumptions:
  - Message length is only partially observable
  - No collisions: $\{M\}_K=\{M'\}_{K'}$ implies M=M' and K=K'
  - Non-malleability: from $\{M\}_K$ cannot construct $\{M'\}_K$
  - No partial information: that attacker cannot guess half the bits of a message, or know half in advance
  - Keys are unguessable, even passwords
- Cryptographers rely on probabilistic **computational models**, making fewer assumptions, but with fewer automated reasoning techniques
- Justifying symbolic models via computational models (where possible), or simply developing automation for the latter, is a growing research area

Example: Authenticated Message




```
let addr : (string * hmac, unit) addr = http "http://localhost:7000/pwdmac" "“
let k = mkHKey()
```

```
let client text =
  assume (Request(text));
  let c  = connect addr in
  let mac = hmacsha1 k (pickle s)
  send c (pickle (s,mac))
```

```
let _ = fork (fun _ -> client k "Hel
let _ = server k
```

```
let server =
  let c = listen addr in
  let text,h = unpickle m in
```

- ./msg.exe
- Connecting to localhost:7000
- Sending {BgAyICsgMj9mhJa7iDAcW3Rrk...} (28 bytes)
- Listening at ::1:7000
- Received Request Hello

We assume that an intruder can interpose a computer on all communication paths, and thus can alter or copy parts of messages, replay messages, or emit false material. While this may seem an extreme view, it is the only safe one when designing authentication protocols. Needham and Schroeder CACM (1978)

The problem: can any attacker break any assertion, given:

```
val addr : (content, content) Net.addr
val client : (string -> string)
val server : (unit -> unit)
```

Query

Crypto

Net

Seal

Pi

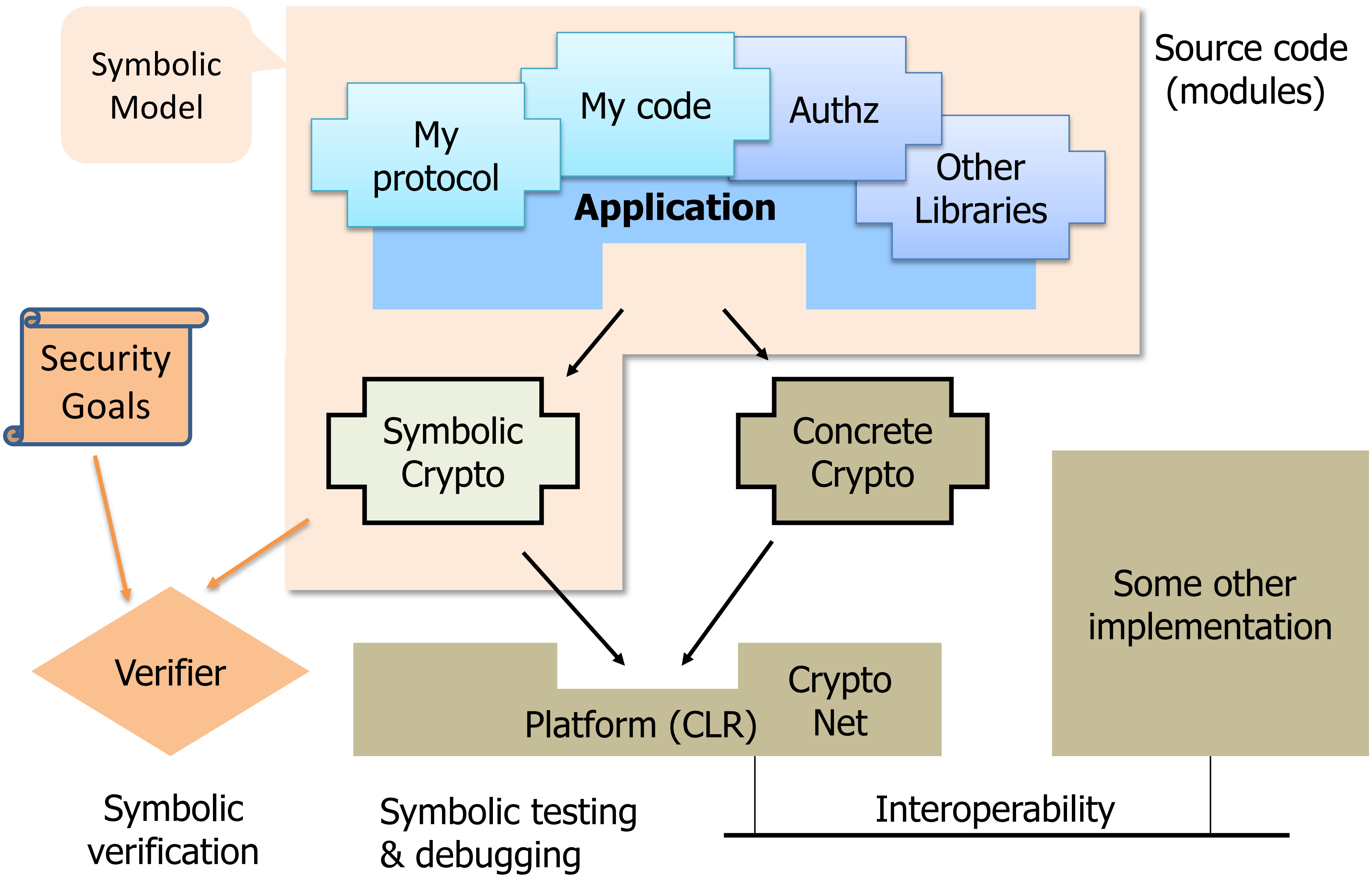
Formal Threat Model: Opponents and Robust Safety

A closed expression O is an *opponent* iff O contains no occurrence of **assert**.

A closed expression A is *robustly safe* iff application $O\ A$ is safe for all opponents O.

Hence, our problem is whether the expression (addr, client, server, . . .) robustly safe.

One Source, Two Tasks



Summary of Lecture 1

- The problem of protocol vulnerabilities remains acute
- Verifying the actual protocol code may help
- We have recast prior work on modelling protocols within process calculi (spi, applied pi) in the setting of ML with concurrency
- Security properties (authenticity, but secrecy too) are expressed using program assertions

- In Lecture 2, we develop RCF – a formal foundation for ML with concurrency – and its system of refinement types
- RCF is the basis for F7, a scalable verifier for protocol code

Friday 4th 10:10-11:00

2

A Formal Calculus for Refinement Types

Cryptographic and Probabilistic Programming, Part 2

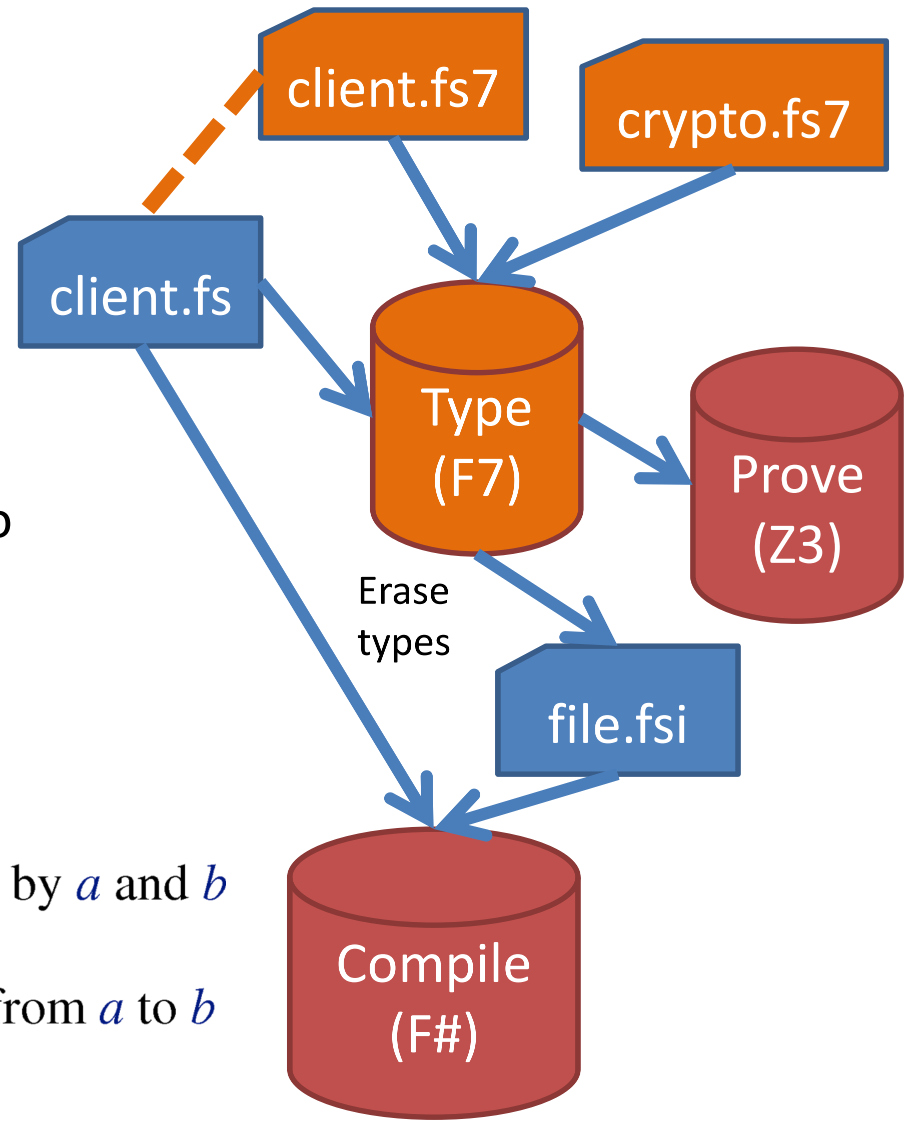
F7: Refinement Types for F#

- We use extended interfaces (.fs7)
  - We typecheck implementations
  - Interfaces include types refined with **first-order formulas**
  - Only libraries security-specific
- F7 supports a large subset of F#
- F7 relies on external SMT solver to discharge proof obligations

$n : \mathrm{int}\{n > 0\}$
is the type of positive integers

$k : \mathrm{bytes}\{KeyAB(k,a,b)\}$
is the type of byte arrays used as keys by a and b

$x : \mathrm{str}\{Request(a,b,x)\}$
is the type of strings sent as requests from a to b



RCF: Refined Concurrent FPC

- supports functional programming a la ML and Haskell,
- has concurrency in the style of process calculus,
- and refinement types, allowing correctness properties to be stated in the style of dependent type theory.

- RCF is the theoretical basis for F7, but there is also a direct implementation (done at Saarbruecken)
- My goal is to explain from first principles how we can show the following RCF example is safe by typechecking:

$$a!42 \upharpoonright (\nu c)((\mathbf{let}\ x = a?\ \mathbf{in}\ \mathbf{assume}\ \mathsf{Sent}(x) \upharpoonright c!x) \upharpoonright (\mathbf{let}\ x = c?\ \mathbf{in}\ \mathbf{assert}\ \mathsf{Sent}(x)))$$

RCF PART 1:
SYNTAX AND SEMANTICS

The Fixpoint Calculus (FPC):

x, y, z	variable
$h ::=$	value constructor
inl	left constructor of sum type
inr	right constructor of sum type
fold	constructor of iso-recursive type
$M, N ::=$	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** $h\ x \rightarrow A$ **else** B	constructor match (scope of x is A)

The Reduction Relation: $A \to A'$

$$(\textbf{fun}\, x \to A)\ N \to A\{N/x\}$$

$$(\textbf{let}\ (x_1, x_2) = (N_1, N_2)\ \textbf{in}\ A) \to A\{N_1/x_1\}\{N_2/x_2\}$$

$$(\textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B) \to \begin{cases} A\{N/x\} & \text{if } M = h\ N \text{ for some } N \\ B & \text{otherwise} \end{cases}$$

$$M = N \to \begin{cases} \mathsf{inl}() & \text{if } M = N \\ \mathsf{inr}() & \text{otherwise} \end{cases}$$

$$\textbf{let}\ x = M\ \textbf{in}\ A \to A\{M/x\}$$

$$A \to A' \Rightarrow \textbf{let}\ x = A\ \textbf{in}\ B \to \textbf{let}\ x = A'\ \textbf{in}\ B$$

Example: Booleans and Conditional Branching:

$\textbf{false} \triangleq \mathsf{inl}\ ()$
$\textbf{true} \triangleq \mathsf{inr}\ ()$
$\textbf{if}\ A\ \textbf{then}\ B\ \textbf{else}\ B' \triangleq$
$\quad \textbf{let}\ x = A\ \textbf{in match}\ x\ \textbf{with}\ \mathsf{inr}(\_) \ \to B\ \textbf{else match}\ x\ \textbf{with}\ \mathsf{inl}(\_) \ \to B'$

Exercise: Derive arithmetic, that is, value zero, functions succ, pred, and iszero.

Exercise: What is the reduction of: $\textbf{if true then}\ B\ \textbf{else}\ B'$

Exercise: Derive list processing, that is, value nil, functions cons, hd, tl, and null.

Exercise: Write down an expression Ω that diverges, that is, $\Omega \to A_1 \to A_2 \to \ldots$.

Exercise: Derive a fixpoint function fix so that we can define recursive function definitions as follows: $\textbf{let rec}\ f x = A \triangleq \textbf{let}\ f = \mathsf{fix}\ (\textbf{fun}\ f \to \textbf{fun}\ x \to A)$.

The Heating Relation $A \Rrightarrow A'$:

Axioms $A \equiv A'$ are read as both $A \Rrightarrow A'$ and $A' \Rrightarrow A$.

$A \Rrightarrow A$

$A \Rrightarrow A''$ if $A \Rrightarrow A'$ and $A' \Rrightarrow A''$

$A \Rrightarrow A' \Rightarrow \textbf{let } x = A \textbf{ in } B \Rrightarrow \textbf{let } x = A' \textbf{ in } B$

$A \rightarrow A'$ if $A \Rrightarrow B, B \rightarrow B', B' \Rrightarrow A'$

Heating is an auxiliary relation; its purpose is to enable reductions, and to place every expression in a normal form, known as a *structure*.
(Process calculi often use a symmetric version, called *structural equivalence*.)

Parallel Composition:

$A, B ::=$	expression
$\quad \ldots$	$\quad$ as before
$\quad A \upharpoonright B$	$\quad$ fork

$$() \upharpoonright A \equiv A$$
$$(A \upharpoonright A') \upharpoonright A'' \equiv A \upharpoonright (A' \upharpoonright A'')$$
$$(A \upharpoonright A') \upharpoonright A'' \Rrightarrow (A' \upharpoonright A) \upharpoonright A''$$
$$\textbf{let}\ x = (A \upharpoonright A')\ \textbf{in}\ B \equiv A \upharpoonright (\textbf{let}\ x = A'\ \textbf{in}\ B)$$

$$A \Rrightarrow A' \Rightarrow (A \upharpoonright B) \Rrightarrow (A' \upharpoonright B)$$
$$A \Rrightarrow A' \Rightarrow (B \upharpoonright A) \Rrightarrow (B \upharpoonright A')$$

$$A \longrightarrow A' \Rightarrow (A \upharpoonright B) \longrightarrow (A' \upharpoonright B)$$
$$B \longrightarrow B' \Rightarrow (A \upharpoonright B) \longrightarrow (A \upharpoonright B')$$

Exercise: Which parameter is passed to the function F by the following expression:
$\textbf{let}\ x = (1 \upharpoonright (2 \upharpoonright 3))\ \textbf{in}\ F x$

Input and Output:

$A, B ::=$	expression
$\ldots$	as before
$a!M$	transmission of M on channel a
$a?$	receive message off channel

$$a!M \Rrightarrow a!M \Rsh ()$$

$$a!M \Rsh a? \rightarrow M$$

Exercise: What are the reductions of the expression: $a!3 \Rsh a? \Rsh a!5$

Exercise: What are the reductions of the expression: $a!3 \Rsh \textbf{let } x = a? \textbf{ in } F\ x$

Exercise: What are the reductions of the expression: $a!\textbf{true} \Rsh a!\textbf{false}$

Name Generation:

$A, B ::=$		expression
	$\ldots$	as before
	$(\nu a)A$	fork

$A \Rrightarrow A' \Rightarrow (\nu a)A \Rrightarrow (\nu a)A'$
$a \notin fn(A') \Rightarrow A' \upharpoonleft ((\nu a)A) \Rrightarrow (\nu a)(A' \upharpoonleft A)$
$a \notin fn(A') \Rightarrow ((\nu a)A) \upharpoonright A' \Rrightarrow (\nu a)(A \upharpoonright A')$
$a \notin fn(B) \Rightarrow$ **let** $x = (\nu a)A$ **in** $B \Rrightarrow (\nu a)$**let** $x = A$ **in** B

$A \longrightarrow A' \Rightarrow (\nu a)A \longrightarrow (\nu a)A'$

Exercise: What are the reductions of the following expression:
let $x = (\nu a)a \upharpoonright (\nu b)b$ **in** $F\ x$

Origins of this Calculus

- RCF is an assembly of standard parts, generalizing some 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 is derivable by coding up **seals** (Morris 1973, Sumii and Pierce 2002), not primitive as in eg spi calculus(Abadi and Gordon, 1997)
  - Security specs via **assume/assert** (Floyd, Hoare, Dijkstra 1970s), generalizing eg correspondences (Woo and Lam 1992)
  - To check assertions statically, rely on dependent functions and pairs with subtyping (Cardelli 1988) and **refinement types** (Pfenning 1992, ...) aka **predicate subtyping** (as in PVS, and more recently Russell)
  - **Public/tainted kinds** to track data that may flow to or from the opponent, as in Cryptyc (Gordon, Jeffrey 2002)

Example: Concurrent ML:

$(T)\mathsf{chan} \triangleq (T \to \mathsf{unit}) * (\mathsf{unit} \to T)$

$\mathsf{chan} \triangleq \mathbf{fun}\ \_ \to (\nu a)(\mathbf{fun}\ x \to a!x, \mathbf{fun}\ \_ \to a?)$	
$\mathsf{send} \triangleq \mathbf{fun}\ c\ x \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ s\ x$	send x on c
$\mathsf{recv} \triangleq \mathbf{fun}\ c \to \mathbf{let}\ (s,r) = c\ \mathbf{in}\ r\ ()$	block for x on c
$\mathsf{fork} \triangleq \mathbf{fun}\ f \to (f() \upharpoonright ())$	run f in parallel

Example: Mutable State:

$(T)\mathbf{ref} \triangleq (T)\mathsf{chan}$

$\mathbf{ref}\ M \triangleq \mathbf{let}\ r = \mathsf{chan}()\ \mathbf{in}\ \ \mathsf{send}\ r\ M; r$	new reference to M
$\mathsf{deref}\ M \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ x; x$	dereference M
$M := N \triangleq \mathbf{let}\ x = \mathsf{recv}\ M\ \mathbf{in}\ \mathsf{send}\ M\ N$	update M with N

Exercise: What are the reductions of the expression: $\mathbf{let}\ x = \mathbf{ref}\ 5\ \mathbf{in}\ x := 7$

Exercise: Encode IMP programs within RCF.

Consider a global set of formulas, the *log*, drawn from some logic.

A General Class of Logics:

$C ::= p(M_1, \ldots, M_n) \mid M = M' \mid \ldots$

$\{C_1, \ldots, C_n\} \vdash C$ deducibility relation

To evaluate **assume** C, add C to the log, and return $()$.
To evaluate **assert** C, return $()$. If C logically follows from the logged formulas, we say the assertion *succeeds*; otherwise, we say the assertion *fails*.

Assume and Assert:

$\textbf{assume}\ C \Rrightarrow \textbf{assume}\ C \upharpoonright ()$

$\textbf{assert}\ C \rightarrow ()$

Exercise: What are the reductions of our running example:

$$a!42 \upharpoonright (\nu c)((\textbf{let}\ x = a?\ \textbf{in}\ \textbf{assume}\ \mathsf{Sent}(x) \upharpoonright c!x) \upharpoonright (\textbf{let}\ x = c?\ \textbf{in}\ \textbf{assert}\ \mathsf{Sent}(x)))$$

Structures and Static Safety:

$e ::= M \mid MN \mid M = N \mid \textbf{let } (x,y) = M \textbf{ in } B \mid$
$\quad \textbf{match } M \textbf{ with } h\, x \to A \textbf{ else } B \mid M? \mid \textbf{assert } C$

$\prod_{i \in 1..n} A_i \triangleq () \upharpoonright A_1 \upharpoonright \ldots \upharpoonright A_n$

$\mathscr{L} ::= \{\} \mid (\textbf{let } x = \mathscr{L} \textbf{ in } B)$

$$\mathbf{S} ::= (\nu a_1)\ldots(\nu a_\ell)\left((\prod_{i \in 1..m} \textbf{assume } C_i) \upharpoonright (\prod_{j \in 1..n} c_j!M_j) \upharpoonright (\prod_{k \in 1..o} \mathscr{L}_k\{e_k\}) \right)$$

Let structure **S** be *statically safe* if and only if,
for all $k \in 1..o$ and C, if $e_k = \textbf{assert } C$ then $\{C_1, \ldots, C_m\} \vdash C$.

Lemma For every expression A, there is a structure **S** such that $A \Rrightarrow \mathbf{S}$.

Expression Safety:

Let expression A be *safe* if and only if,
for all A' and **S**, if $A \to^* A'$ and $A' \Rrightarrow \mathbf{S}$, then **S** is statically safe.

RCF PART 2:
TYPES FOR SAFETY

Starting Point: The Type System for FPC:

$$\frac{E \vdash \diamond \quad (x:T) \in E}{E \vdash x : T} \qquad \frac{E \vdash A : T \quad E, x : T \vdash B : U}{E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : U}$$

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathsf{unit} + \mathsf{unit}}$$

$$\frac{E, x : T \vdash A : U}{E \vdash \textbf{fun}\, x \to A : (T \to U)} \qquad \frac{E \vdash M : (T \to U) \quad E \vdash N : T}{E \vdash M\ N : U}$$

$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash (M, N) : (T \times U)} \qquad \frac{E \vdash M : (T \times U) \quad E, x : T, y : U \vdash A : V}{E \vdash \textbf{let}\ (x, y) = M\ \textbf{in}\ A : V}$$

$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U} \qquad \frac{E \vdash M : T \quad h : (H, T) \quad E, x : H \vdash A : U \quad E \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \to A\ \textbf{else}\ B : U}$$

$\mathsf{inl}{:}(T, T+U) \quad \mathsf{inr}{:}(U, T+U) \quad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

Exercise: Write types of Booleans, numbers, and lists.
Exercise: Write a well-typed fixpoint combinator.

Three Steps Toward Safety by Typing

1. We include **refinement types** {*x* : *T* | *C*}, whose values are those of *T* that satisfy *C*
2. To exploit refinements, we add a judgment *E* |- *C*, meaning that *C* follows from the refinement types in *E*
3. To manage refinement formulas, we need (1) dependent versions of the function and pair types, and (2) subtyping

- A value of $\Pi x : T.\ U$ is a function M such that if N has type T, then $M\ N$ has type $U\{N/x\}$.
- A value of $\Sigma x : T.\ U$ is a pair (M, N) such that M has type T and N has type $U\{M/x\}$.
- If $A : T$ and $T <: U$ then $A : U$.

Syntax of RCF Types:

$H, T, U, V ::=$	type
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)
α	iso-recursive type variable
$\{x : T \mid C\}$	refinement type (scope of x is C)

$\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$	ok-type
$\mathsf{bool} \triangleq \mathsf{unit} + \mathsf{unit}$	Boolean type

Starting Point: The Type System for FPC:

$$\frac{E \vdash \diamond \quad (x:T) \in E}{E \vdash x : T} \qquad \frac{E \vdash A : T \quad E, x : T \vdash B : U}{E \vdash \textbf{let}\ x = A\ \textbf{in}\ B : U}$$

$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{unit}} \qquad \frac{E \vdash M : T \quad E \vdash N : U}{E \vdash M = N : \mathsf{unit} + \mathsf{unit}}$$

$$\frac{E, x : T \vdash A : U}{E \vdash \textbf{fun}\, x \rightarrow A : (T \rightarrow U)} \qquad \frac{E \vdash M : (T \rightarrow U) \quad E \vdash N : T}{E \vdash M\ N : U}$$

$$\frac{E \vdash M : T \quad E \vdash N : U}{E \vdash (M, N) : (T \times U)} \qquad \frac{E \vdash M : (T \times U) \quad E, x : T, y : U \vdash A : V}{E \vdash \textbf{let}\ (x, y) = M\ \textbf{in}\ A : V}$$

$$\frac{h : (T, U) \quad E \vdash M : T \quad E \vdash U}{E \vdash h\ M : U} \qquad \frac{E \vdash M : T \quad h : (H, T) \quad E, x : H \vdash A : U \quad E \vdash B : U}{E \vdash \textbf{match}\ M\ \textbf{with}\ h\ x \rightarrow A\ \textbf{else}\ B : U}$$

$\mathsf{inl}{:}(T, T+U) \quad \mathsf{inr}{:}(U, T+U) \quad \mathsf{fold}{:}(T\{\mu\alpha.T/\alpha\}, \mu\alpha.T)$

Exercise: Write types of Booleans, numbers, and lists.

Exercise: Write a well-typed fixpoint combinator.

Rules for Formula Derivation:

$$\mathsf{forms}(E) \triangleq \begin{cases} \{C\{y/x\}\} \cup \mathsf{forms}(y:T) & \text{if } E = (y:\{x:T \mid C\}) \\ \mathsf{forms}(E_1) \cup \mathsf{forms}(E_2) & \text{if } E = (E_1, E_2) \\ \varnothing & \text{otherwise} \end{cases}$$

$$\frac{E \vdash \diamond \quad \mathit{fnfv}(C) \subseteq \mathit{dom}(E) \quad \mathsf{forms}(E) \vdash C}{E \vdash C}$$

Exercise: What is $\mathsf{forms}(E)$ if $E = x_1 : \{y_1 : \mathsf{int} \mid \mathsf{Even}(y_1)\}, x_2 : \{y_2 : \mathsf{int} \mid \mathsf{Odd}(x_1)\}$?

Exercise: A handy abbreviation is $\{C\} \triangleq \{\_ : \mathsf{unit} \mid C\}$, where $\_$ is fresh. What is $\mathsf{forms}(x : \{C\})$?

We write $E \vdash C$ to mean that C follows from the refinement formulas in C.
For example, $x : \{x : \mathsf{int} \mid x > 0\}, b : \{b : \mathsf{bool} \mid x < 2\} \vdash x = 1$.
(In F7, did we try to implement this directly?)

Rules for Assume and Assert:

$$\frac{E \vdash \diamond \quad fnfv(C) \subseteq dom(E)}{E \vdash \textbf{assume } C : \{\_ : \mathsf{unit} \mid C\}} \qquad \frac{E \vdash C}{E \vdash \textbf{assert } C : \mathsf{unit}}$$

Subtyping Rules for Refinement Types:

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

Exercise: How would we derive $\vdash \{x : \mathsf{int} \mid x > 0\} <: \mathsf{int}$.
Exercise: Derive the following subtyping rules:

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

Standard Rules of (Dependent) Subtyping:

$$\frac{E \vdash A : T \quad E \vdash T <: T'}{E \vdash A : T'}$$

$$\frac{E \vdash \diamond}{E \vdash \mathsf{unit} <: \mathsf{unit}} \qquad \frac{E \vdash T' <: T \quad E, x : T' \vdash U <: U'}{E \vdash (\Pi x : T.\ U) <: (\Pi x : T'.\ U')}$$

$$\frac{E \vdash T <: T' \quad E, x : T \vdash U <: U'}{E \vdash (\Sigma x : T.\ U) <: (\Sigma x : T'.\ U')} \qquad \frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash (T + U) <: (T' + U')}$$

$$\frac{E \vdash \diamond \quad (\alpha <: \alpha') \in E}{E \vdash \alpha <: \alpha'} \qquad \frac{E, \alpha <: \alpha' \vdash T <: T' \quad \alpha \notin fnfv(T') \quad \alpha' \notin fnfv(T)}{E \vdash (\mu \alpha . T) <: (\mu \alpha' . T')}$$

Exercise: Understand why:

$\vdash \{x : \mathsf{int} \mid x > 0\} <: \mathsf{int}$

$\vdash (\Pi x : \mathsf{int}.\ \mathsf{bool}) <: (\Pi x : \{x : \mathsf{int} \mid x > 0\}.\ \mathsf{bool})$

but not:

$\vdash (\Pi x : \{x : \mathsf{int} \mid x > 0\}.\ \mathsf{bool}) <: (\Pi x : \mathsf{int}.\ \mathsf{bool})$

Exercise: Prove that $E \vdash T <: T'$ is decidable, assuming an oracle for $E \vdash C$.

Exercise: (Hard.) Prove that $E \vdash T <: T'$ is transitive.

Rules for Restriction, I/O, and Parallel Composition:

$$\frac{E, a \updownarrow T \vdash A : U \quad a \notin fn(U)}{E \vdash (\nu a)A : U} \qquad \frac{E \vdash M : T \quad (a \updownarrow T) \in E}{E \vdash a!M : \mathsf{unit}} \qquad \frac{E \vdash \diamond \quad (a \updownarrow T) \in E}{E \vdash a? : T}$$

$$\frac{E, \_ : \{\overline{A_2}\} \vdash A_1 : T_1 \qquad E, \_ : \{\overline{A_1}\} \vdash A_2 : T_2}{E \vdash (A_1 \upharpoonright A_2) : T_2}$$

$$\overline{(\nu a)A} = (\exists a.\overline{A})$$
$$\overline{A_1 \upharpoonright A_2} = (\overline{A_1} \wedge \overline{A_2})$$
$$\overline{\textbf{let } x = A_1 \textbf{ in } A_2} = \overline{A_1}$$
$$\overline{\textbf{assume } C} = C$$
$$\overline{A} = \mathsf{True} \quad \text{if } A \text{ matches no other rule}$$

Exercise: Find types to typecheck the following code:

$$a!42 \upharpoonright (\nu c)((\textbf{let } x = a? \textbf{ in assume } \mathsf{Sent}(x) \upharpoonright c!x) \upharpoonright (\textbf{let } x = c? \textbf{ in assert } \mathsf{Sent}(x)))$$

Type System and Theorem

$E ::= x_1 : T_1, \ldots, x_n : T_n$ environment

$E \vdash \diamond$	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
$E \vdash T <: U$	in E, type T is a subtype of type U
$E \vdash A : T$	in E, expression A has type T

Lemma If $\varnothing \vdash \mathbf{S} : T$ then $\mathbf{S}$ is statically safe.
Lemma If $E \vdash A : T$ and $A \Rrightarrow A'$ then $E \vdash A' : T$.
Lemma If $E \vdash A : T$ and $A \rightarrow A'$ then $E \vdash A' : T$.

Theorem If $\varnothing \vdash A : T$ then A is safe.
(For any A' and $\mathbf{S}$ such that $A \rightarrow^* A'$ and $A' \Rrightarrow \mathbf{S}$ we need that $\mathbf{S}$ is statically safe.)

RCF III: TYPES FOR ROBUST SAFETY

Safety Versus an Untyped Adversary

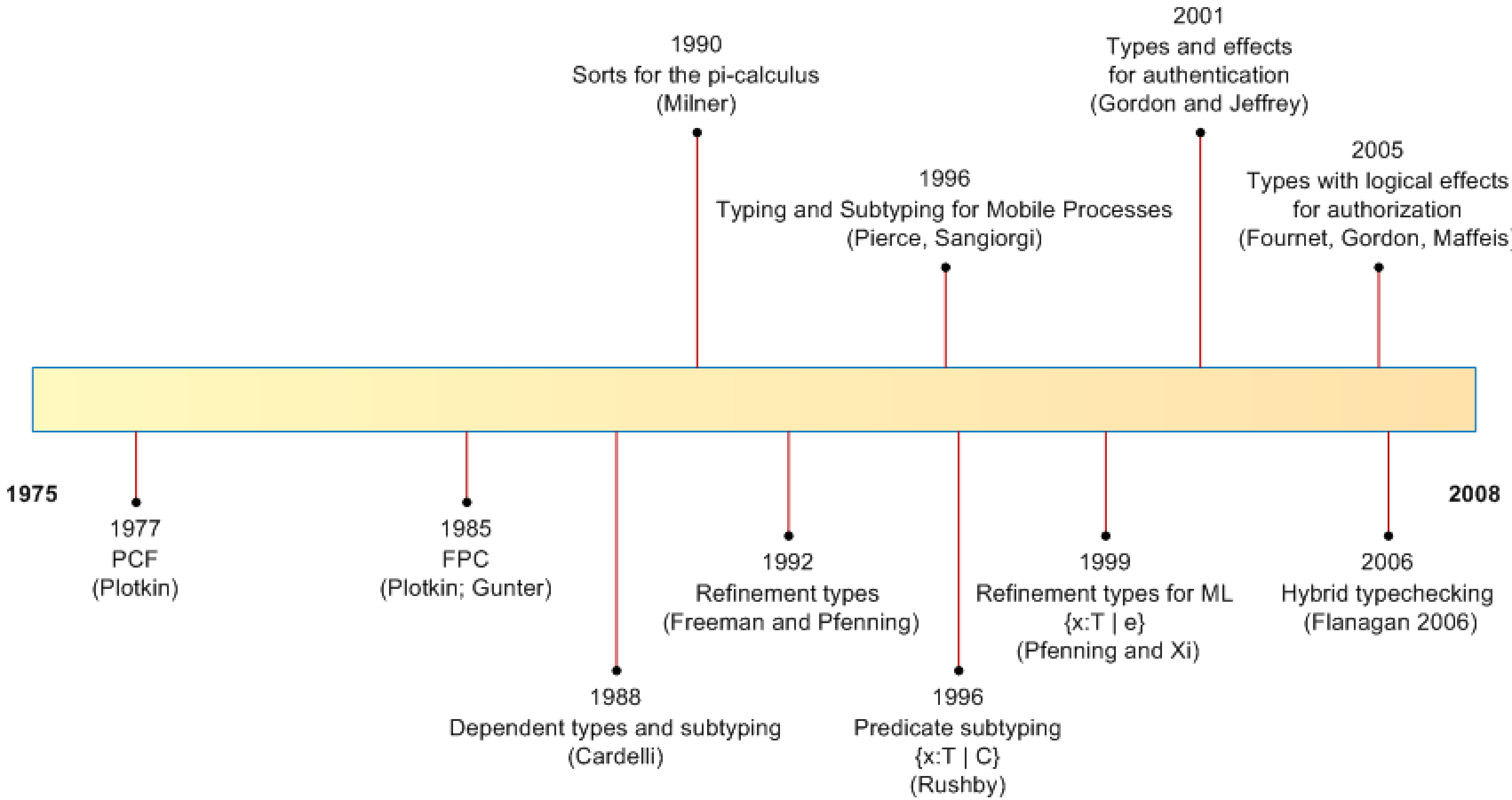
Closed expression A is *robustly safe* iff the application $O\ A$ is safe, for all opponents O.
Well-typed expressions are safe, but not in general robustly safe.
Consider $\mathbf{fun}\, x : \mathsf{pos} \rightarrow (\mathbf{assert}\ x > 0)$ where $\mathsf{pos} \triangleq \{x : \mathsf{int} \mid x > 0\}$.
Type T is *public* iff all refinements occur positively.

- pos
- $\mathsf{int} \rightarrow \mathsf{pos}$
- $\mathsf{pos} \rightarrow \mathsf{int}$
- $(\mathsf{pos} \rightarrow \mathsf{int}) \rightarrow \mathsf{int}$

We extend the type system with a type Un and public/tainted rules to get:

Lemma 1 (Opponent Typability) *If O is any opponent then $\varnothing \vdash O : \mathsf{Un}$.*

Theorem 1 (Robust Safety) *If $\varnothing \vdash A : T$ and T is public then A is robustly safe.*



TYPE THEORIES BEHIND RCF

Summary of Lecture 2

- RCF is an assembly of standard parts, generalizing some ad hoc constructions in language-based security
- It underpins F7, a scalable verifier for security code
- In the next lecture, we consider applications of F7, its successor F*, and adaptations of this work to programs in C

- http://research.microsoft.com/F7

Friday 4th 17:00-18:00

3

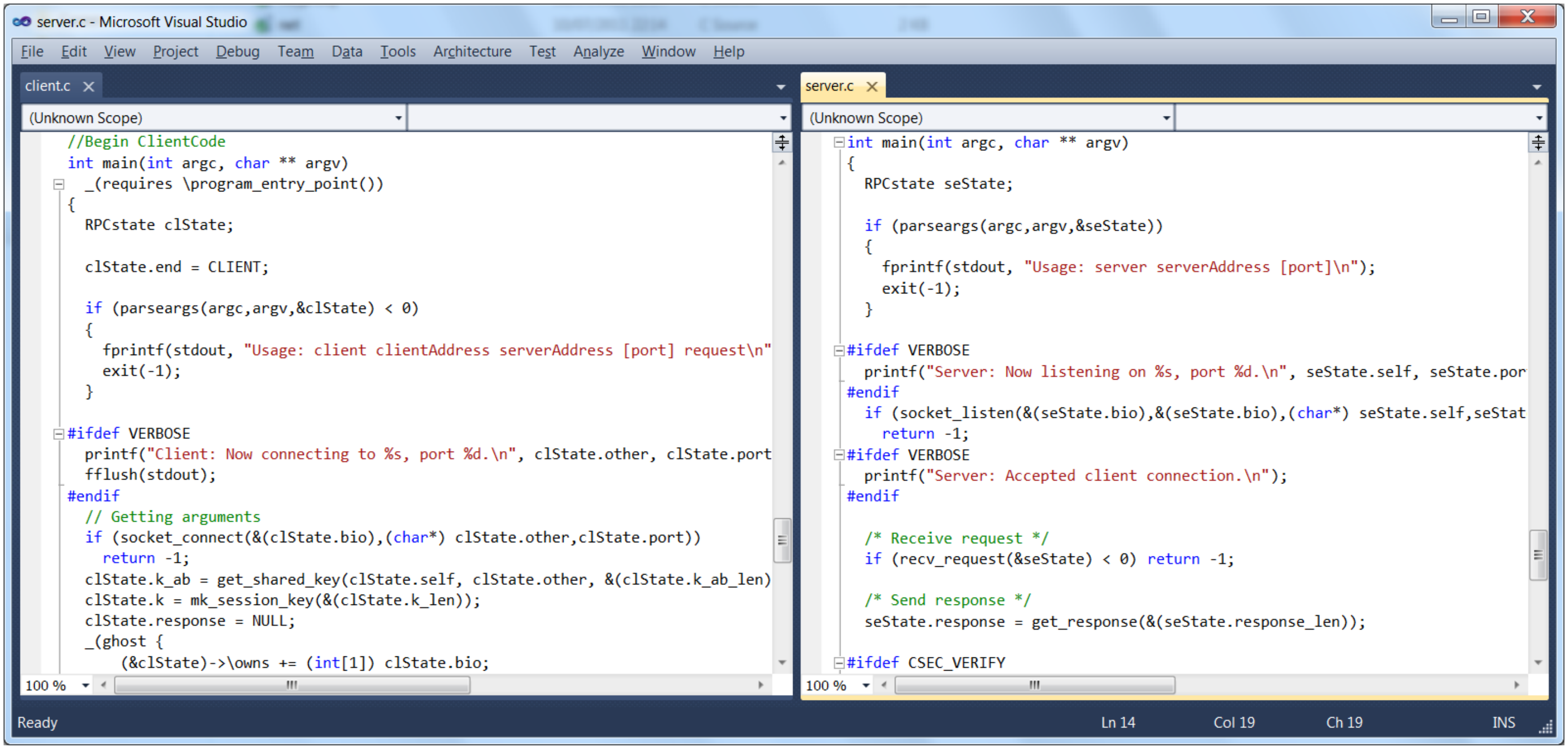
Verified Cryptographic Programs for Protocols

Cryptographic and Probabilistic Programming, Part 3

From a Statement for Panel on "CSF: The Next Twenty Years" at CSF20, Venice, 2007

The Rise of Code Verification

- Re security protocols and the Needham-Schroeder problem:
  - The first 20 years of CSF has seen the **Rise of Model Verification**
  - The next 20 years of CSF will see the **Rise of Code Verification**
- If we can verify code in the languages implementors actually use, we can find and fix security properties as soon as protocols are first implemented
- We may well do better to teach existing software verification tools about the attacker, than to build from scratch
- Into the bargain, we'll detect other security bugs, eg, overruns, using the same tools


server.c - Microsoft Visual Studio
File Edit View Project Debug Team Data Tools Architecture Test Analyze Window Help
client.c
(Unknown Scope)
//Begin ClientCode
int main(int argc, char ** argv)
_(requires \program_entry_point())
{
RPCstate clState;
clState.end = CLIENT;
if (parseargs(argc,argv,&clState) < 0)
{
fprintf(stdout, "Usage: client clientAddress serverAddress [port] request\n"
exit(-1);
}
#ifdef VERBOSE
printf("Client: Now connecting to %s, port %d.\n", clState.other, clState.port
fflush(stdout);
#endif
// Getting arguments
if (socket_connect(&(clState.bio),(char*) clState.other,clState.port))
return -1;
clState.k_ab = get_shared_key(clState.self, clState.other, &(clState.k_ab_len)
clState.k = mk_session_key(&(clState.k_len));
clState.response = NULL;
_(ghost {
(&clState)->\owns += (int[1]) clState.bio;
100 %
server.c
(Unknown Scope)
int main(int argc, char ** argv)
{
RPCstate seState;
if (parseargs(argc,argv,&seState))
{
fprintf(stdout, "Usage: server serverAddress [port]\n");
exit(-1);
}
#ifdef VERBOSE
printf("Server: Now listening on %s, port %d.\n", seState.self, seState.por
#endif
if (socket_listen(&(seState.bio),&(seState.bio),(char*) seState.self,seStat
return -1;
#ifdef VERBOSE
printf("Server: Accepted client connection.\n");
#endif
/* Receive request */
if (recv_request(&seState) < 0) return -1;
/* Send response */
seState.response = get_response(&(seState.response_len));
#ifdef CSEC_VERIFY
100 %
Ready
Ln 14
Col 19
Ch 19
INS

An Example Protocol

Client: Now connecting to localhost, port 4433

Server: Now listening on localhost, port 4433

Authenticated RPC: RPC-enc

$$A \to B\colon A, \{request, k_{req}\}_{k_{AB}}$$

$$B \to A\colon \{response\}_{k_{req}}$$

Client: Received encrypted message:
6a64b21d6d93a65aead74fa820d7049fd661bd2a
9495deaef59c528b51e4042cb10a47d507e42c1c
132a8855b5d8081c46197131

Client: Received and authenticated response:
Look out the window.

Server: Sending encrypted message:
6a64b21d6d93a65aead74fa820d7049fd661bd2a
9495deaef59c528b51e4042cb10a47d507e42c1c
132a8855b5d8081c46197131

```
$ proverif -in pi pvmodel.out | grep RESULT
RESULT not ev:client_accept(x_23,y_24) is false.
RESULT ev:server_reply(x_219,y_220) ==> ev:client_begin(x_219) is true.
RESULT ev:client_accept(x_346,y_347) ==> ev:server_reply(x_346,y_347) is true
$
```

Authenticated RPC: RPC-enc

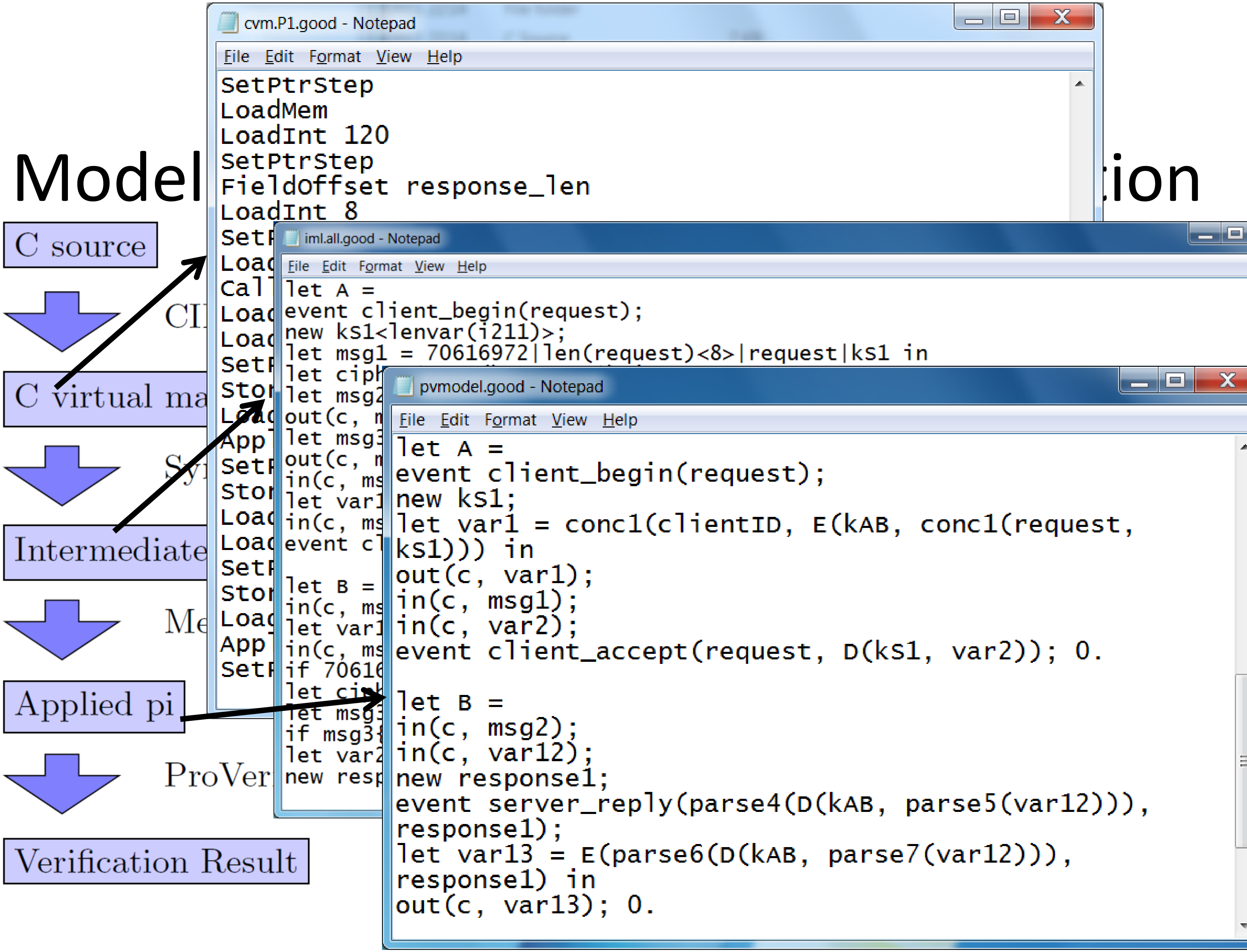
$A \rightarrow B$: $A, \{request, k_{req}\}_{k_{AB}}$
$B \rightarrow A$: $\{response\}_{k_{req}}$

```
let A =  event client_begin(request);
new kS1;
let var1 = conc1(clientID, E(kAB, conc1(request, kS1))) in
out(c, var1);
in(c, msg1);
in(c, var2);
event client_accept(request, D(kS1, var2)); 0.
```

```
let B =
in(c, msg2);
in(c, var12);
new response1;
event server_reply(fst(D(kAB, snd(var12))), response1);
let var13 = E(snd(D(kAB, snd(var12))), response1) in
out(c, var13); 0.

process  ! new kAB; (!A | !B)
```

PhD work of Mihhail Aizatulin, papers at CCS 2011-2012

SOLUTION VIA SYMBOLIC EXECUTION


Model
tion
C source
C virtual ma
Intermediate
Applied pi
ProVer
Verification Result
cvm.P1.good - Notepad
File Edit Format View Help
SetPtrStep
LoadMem
LoadInt 120
SetPtrStep
Fieldoffset response_len
LoadInt 8
iml.all.good - Notepad
File Edit Format View Help
let A =
event client_begin(request);
new ks1<lenvar(i211)>;
let msg1 = 70616972|len(request)<8>|request|ks1 in
let B =
pvmodel.good - Notepad
File Edit Format View Help
let A =
event client_begin(request);
new ks1;
let var1 = conc1(clientID, E(kAB, conc1(request, ks1))) in
out(c, var1);
in(c, msg1);
in(c, var2);
event client_accept(request, D(ks1, var2)); 0.
let B =
in(c, msg2);
in(c, var12);
new response1;
event server_reply(parse4(D(kAB, parse5(var12))), response1);
let var13 = E(parse6(D(kAB, parse7(var12))), response1) in
out(c, var13); 0.

C line	symbolic execution steps
`int send_request(RPCstate * ctx){`	

1. `uint32_t m1_len, m1_e_len, full_len;`
`unsigned char * m1, * p, * m1_e;`
`m1_len = 1 + ctx→k_s_len`
`+ sizeof(ctx→request_len)`
`+ ctx→request_len;`
	stack $m1\_len \Rightarrow 1 + \mathrm{len}(k_S) + 4 + \mathrm{len}(request)$

2. `p = m1 = malloc(m1_len);`	stack $p \Rightarrow \mathrm{ptr}(\mathrm{heap}\,6,\ 0)$ stack $m1 \Rightarrow \mathrm{ptr}(\mathrm{heap}\,6,\ 0)$
3. `memcpy(p, "p", 1);`	heap $6 \Rightarrow$ 'p'
4. `p += 1;`	stack $p \Rightarrow \mathrm{ptr}(\mathrm{heap}\,6,\ 1)$
5. `* (uint32_t *) p = ctx→request_len;`	heap $6 \Rightarrow$ 'p'$\|\mathrm{len}(request)$
6. `p += sizeof(ctx→request_len);`	stack $p \Rightarrow \mathrm{ptr}(\mathrm{heap}\,6,\ 5)$
7. `memcpy(p, ctx→request, ctx→request_len);`	heap $6 \Rightarrow$ 'p'$\|\mathrm{len}(request)\|request$
8. `p += ctx→request_len;`	stack $p \Rightarrow \mathrm{ptr}(\mathrm{heap}\,6,\ 5 + \mathrm{len}(request))$
9. `memcpy(p, ctx→k_s, ctx→k_s_len);`	heap $6 \Rightarrow$ 'p'$\|\mathrm{len}(request)\|request\|k_S$

10. `full_len = 1 + sizeof(ctx→self_len)`
`+ ctx→self_len`
`+ encrypt_len(ctx→k_ab, ctx→k_ab_len,`
`m1, m1_len);`
	stack $full\_len \Rightarrow 5 + \mathrm{len}(clientID)$
	$+ encrypt\_len(msg1)$

11. `p = m1_e = malloc(full_len);`	stack $p \Rightarrow$ heap 7 stack $m1\_e \Rightarrow$ heap 7
12. `memcpy(p, "p", 1);`	heap $7 \Rightarrow$ 'p'
13. `p += 1;`	stack $p \Rightarrow \mathrm{ptr}(\mathrm{heap}\,7,\ 1)$
14. `* (uint32_t *) p = ctx→self_len;`	heap $7 \Rightarrow$ 'p'$\|\mathrm{len}(clientID)$
15. `p += sizeof(ctx→self_len);`	stack $p \Rightarrow \mathrm{ptr}(\mathrm{heap}\,7,\ 5)$
16. `memcpy(p, ctx→self, ctx→self_len);`	heap $7 \Rightarrow$ 'p'$\|\mathrm{len}(clientID)\|clientID$
17. `p += ctx→self_len;`	stack $p \Rightarrow \mathrm{ptr}(\mathrm{heap}\,7,\ 5 + \mathrm{len}(clientID))$

18. `m1_e_len`
`= encrypt(ctx→k_ab, ctx→k_ab_len,`
`m1, m1_len, p);`
	heap $7 \Rightarrow$ 'p'$\|\mathrm{len}(clientID)\|clientID\|cipher1$
	stack $m1\_e\_len \Rightarrow \mathrm{len}(cipher1)$
	new fact: $\mathrm{len}(cipher1) \le encrypt\_len(msg1)$
	$cipher1 = E(key(clientID, serverID), msg1)$
	$msg1 =$ 'p'$\|\mathrm{len}(request)\|request\|k_S$

19. `full_len = 1 + sizeof(ctx→self_len)` `+ ctx→self_len + m1_e_len;`	stack $full\_len \Rightarrow 5 + \mathrm{len}(clientID)$ $+ \mathrm{len}(cipher1)$
20. `send(&(ctx→bio),` `&full_len, sizeof(full_len));`	*generate IML:* **out**$(c,\ 5 + \mathrm{len}(cipher1) + \mathrm{len}(cipher1))$;
21. `send(&(ctx→bio), m1_e, full_len);}`	*generate IML:* **out**$(c,$ 'p'$\|\mathrm{len}(clientID)\|clientID\|cipher1)$;

	C LOC	IML LOC	outcome	result type	time
simple mac	∼ 250	12	verified	symbolic	4s
RPC	∼ 600	35	verified	symbolic	5s
NSL	∼ 450	40	verified	computat.	5s
CSur	∼ 600	20	flaw: fig. 11	—	5s
minexplib	∼ 1000	51	flaw: fig. 12	—	15s

Figure 10: Summary of analysed implementations.

```
read(conn_fd, temp, 128);
// BN_hex2bn expects zero-terminated string
temp[128] = 0;
BN_hex2bn(&cipher_2, temp);
// decrypt and parse cipher_2
// to obtain message fields
```

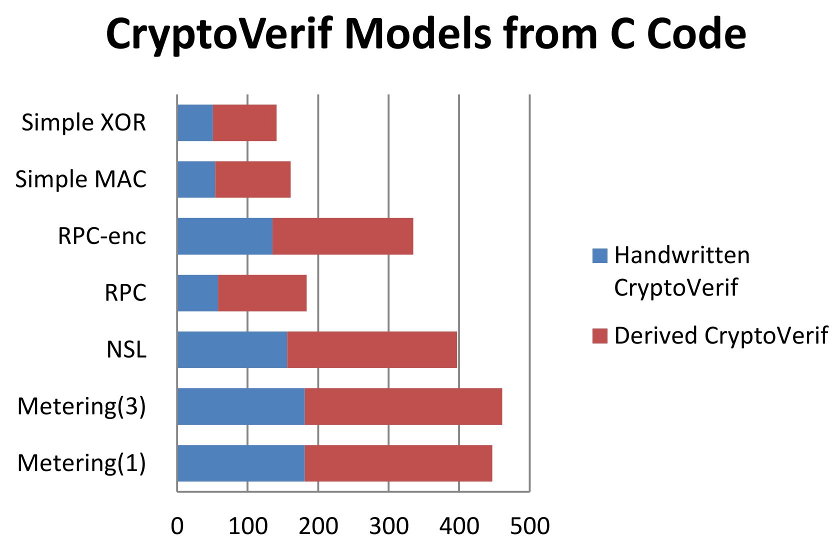
Figure 11: A flaw in the CSur example: input may be too short.

```
unsigned char session_key[256 / 8];
...
// Use the 4 first bytes as a pad
// to encrypt the reading
encrypted_reading =
  ((unsigned int) *session_key) ^ *reading;
```

Figure 12: A flaw in the minexplib code: only one byte of the pad is used.

Computational Verification

- First security analysis of C code to target a verifier for the probabilistic computational model
(ie, not “perfect” symbolic crypto)
- Builds on Blanchet’s CryptoVerif
- Verify over 3000 LOC, more than any prior work on cryptographic code in C



	C LOC	CV LOC	Time	Primitives
Simple MAC	~ 250	109	4s	UF-CMA MAC
Simple XOR	~ 100	68	3s	XOR
NSL	~ 450	262	86s	IND-CCA2 PKE
RPC	~ 600	145	13s	UF-CMA MAC
RPC-enc	~ 700	234	9s	IND-CPA INT-CTXT AE
Metering	~ 1000	299	33s	UF-CMA sig, CR/PRF hash

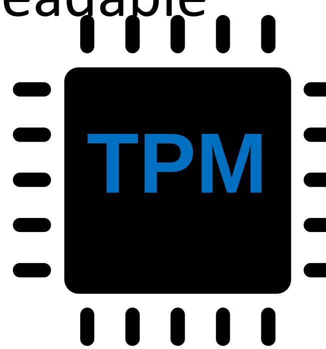
Model Extraction

- Allows automatic extraction of protocol model from code
  - Assumes protocol follows a single correct run, and any deviation should terminate immediately
  - Tools allows protocol designer to write π-calculus in C
  - Verification shows the model is correct, but not the code, as it may follow other paths

- Future directions?
  - Backpatch the code to terminate if it deviates from normal path
  - Scale to more examples eg PolarSSL handshake

Towards Full Verification

- Proves memory safety and symbolic security of C code
  - PhD work of Francois Dupressoir, paper
  - Full verification based on the MSR VCC tool, but needs much more interactive effort than symbolic execution
- Strategy: port theory of crypto from F7 to VCC
  - Not preventing timing, power consumption, physical attacks
- Future challenge
  - Work with Trusted Computing Group on TPM 2.0 chip – using stylized ANSI-C as a normative "Machine+Human-Readable Specification"



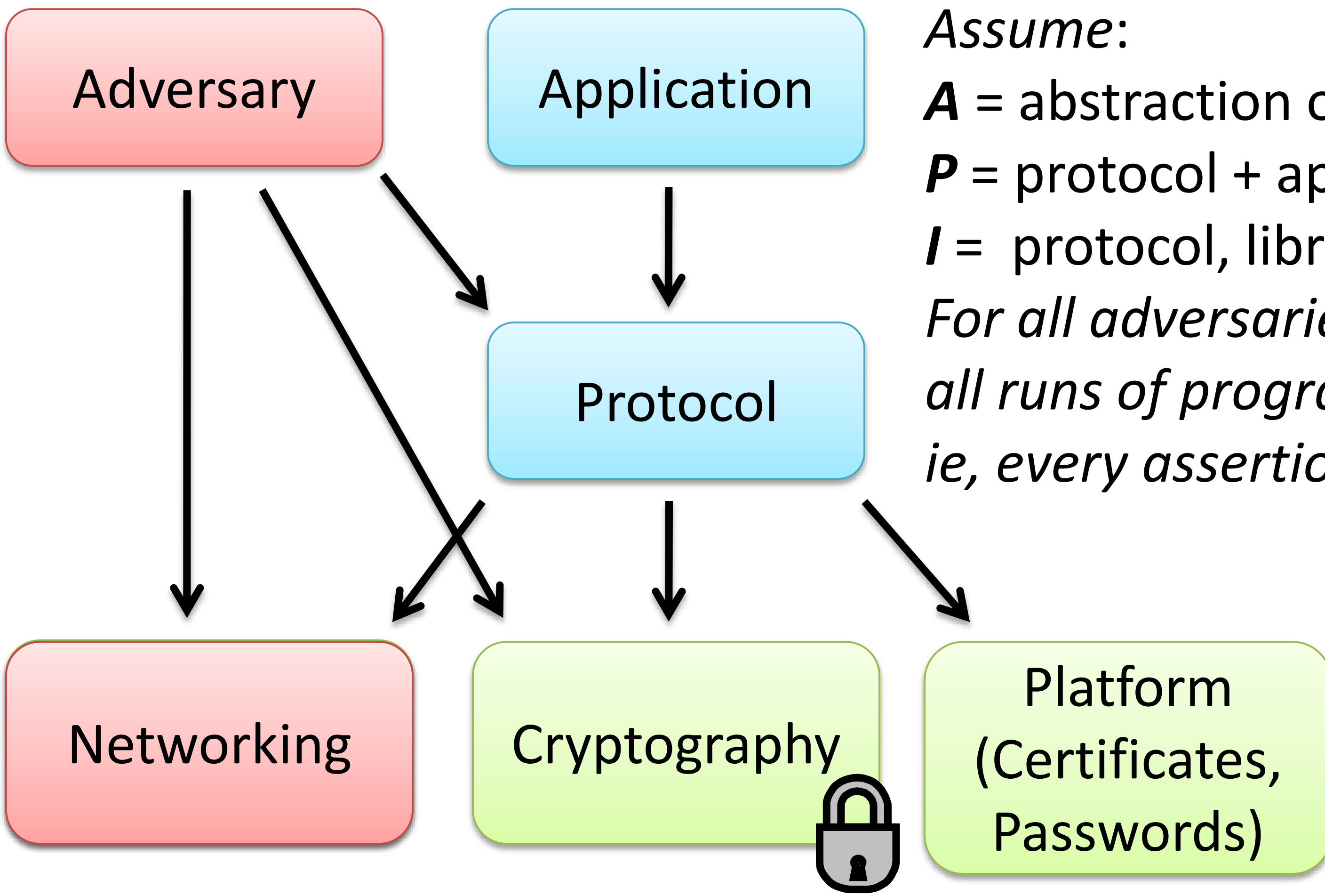
Main Lines of Related Work on C

- **Csur** [Goubault-Larrecq and Parrennes 2005] analyzes C code for secrecy properties via a custom abstract interpretation.
- **Pistachio** [Udrea et al 2006] verifies compliance of C code with a rule-based specification of the communication steps of a protocol, but doesn't show security of the specification.
- **ASPIER** [Chaki and Datta 2009] relies on security-specific software model-checking techniques, obtaining good results on the main loop of OpenSSL.
- Corin and Manzano [2011] extend the **KLEE** symbolic execution engine to represent the outcome of cryptographic algorithms symbolically.
- Cade and Blanchet [2013] compile the CryptoVerif input language to Ocaml and obtain computational guarantees; an application is to the SSH Transport Layer
- Almeida et al [2014] show correctness of implementations of secure and verifiable computation over encrypted data using EasyCrypt.

F7: AN IMPLEMENTATION OF RCF

http://research.microsoft.com/F7

What Does F7 Prove By Typing?



***Verification Goal**: Robust Safety*

Assume:

A = abstraction of libraries

P = protocol + application

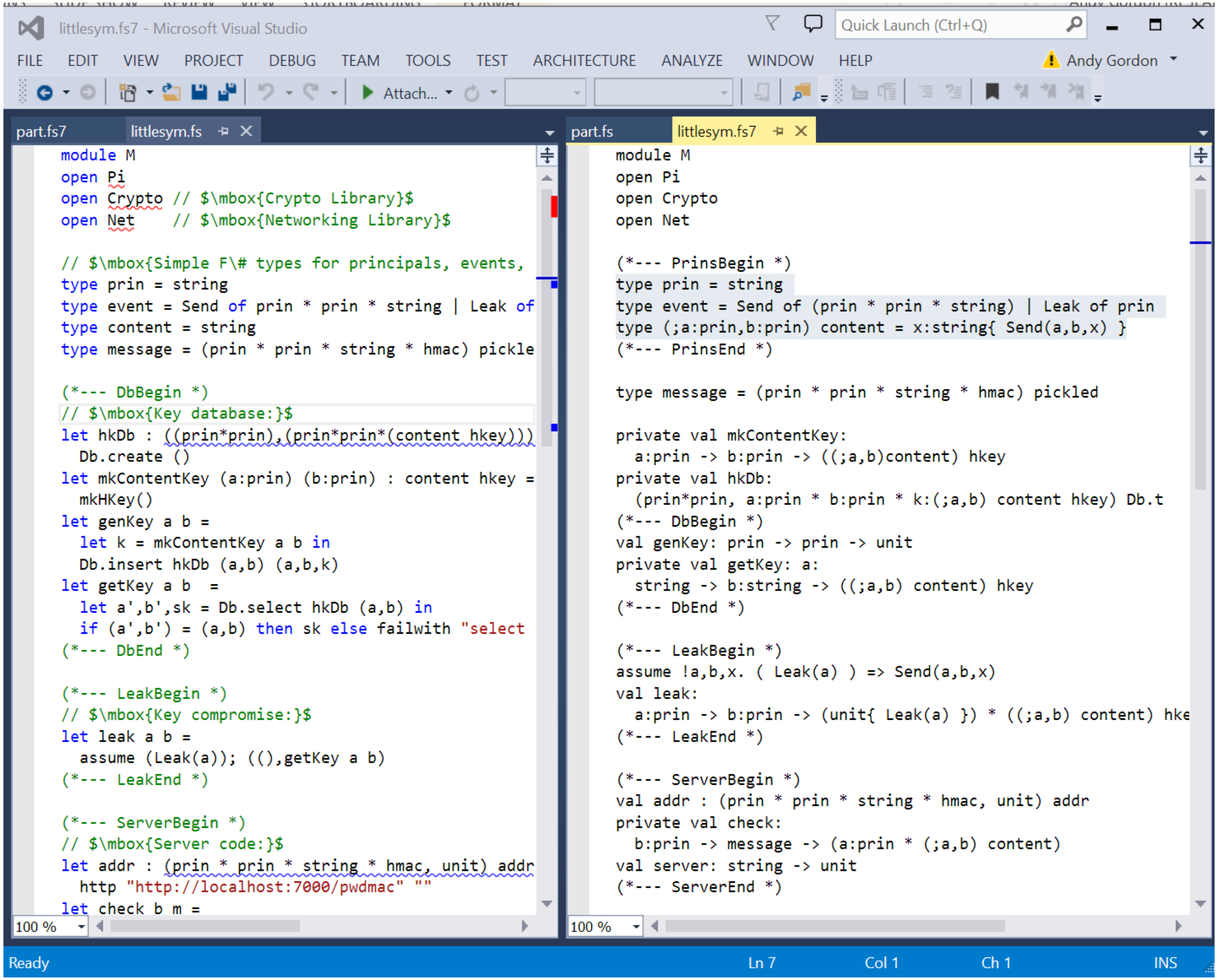
I = protocol, library interface

*For all adversaries **O** that use **I**,*

all runs of program **A P O** *are safe,*

ie, every assertion succeeds

F7 on Example from Lecture 1



Implementing TLS with Verified Cryptographic Security

Karthikeyan Bhargavan
Cédric Fournet
Markulf Kohlweiss
Alfredo Pironti
Pierre-Yves Strub

INRIA, Microsoft Research and IMDEA

Transport Layer Security (1995—)

The most widely deployed cryptographic protocol?

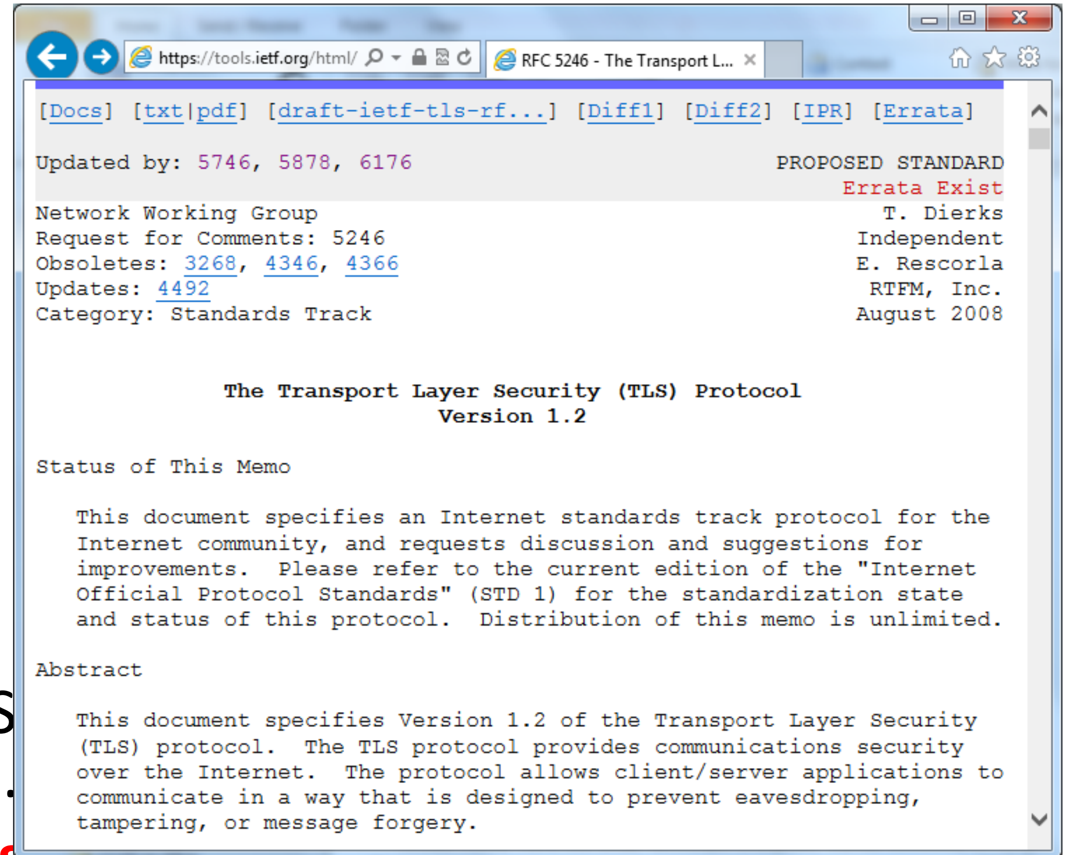
HTTPS, 802.1x (EAP), FTPS, VPN, mail, VoIP, …

18 years of attacks, fixes, and extensions

1995 – Netscape's Secure Sockets Layer
1995 – SSL2
1996 – SSL3
1999 – TLS1.0 (RFC2246, ≈SSL3)
2006 – TLS1.1 (RFC4346)
2008 – TLS1.2 (RFC5246)

Many implementations

- SChannel, OpenSSL, NSS, GnuTLS, JSSE, PolarSSL, …
- **Several security patches every year**



Many papers

- **Well-understood, detailed specs**
- **Security theorems… mostly for small simple models of TLS**

What can still possibly go wrong?

Application
protocol configuration

Infrastructure
certificate management

Protocol Logic
e.g. ambiguous messages

- cause servers to attribute secrets to wrong clients

TLS DESIGN

Cryptography
e.g. no fresh IV

- write applet to realize adaptive attack (BEAST)

Implementation Errors
many critical bugs

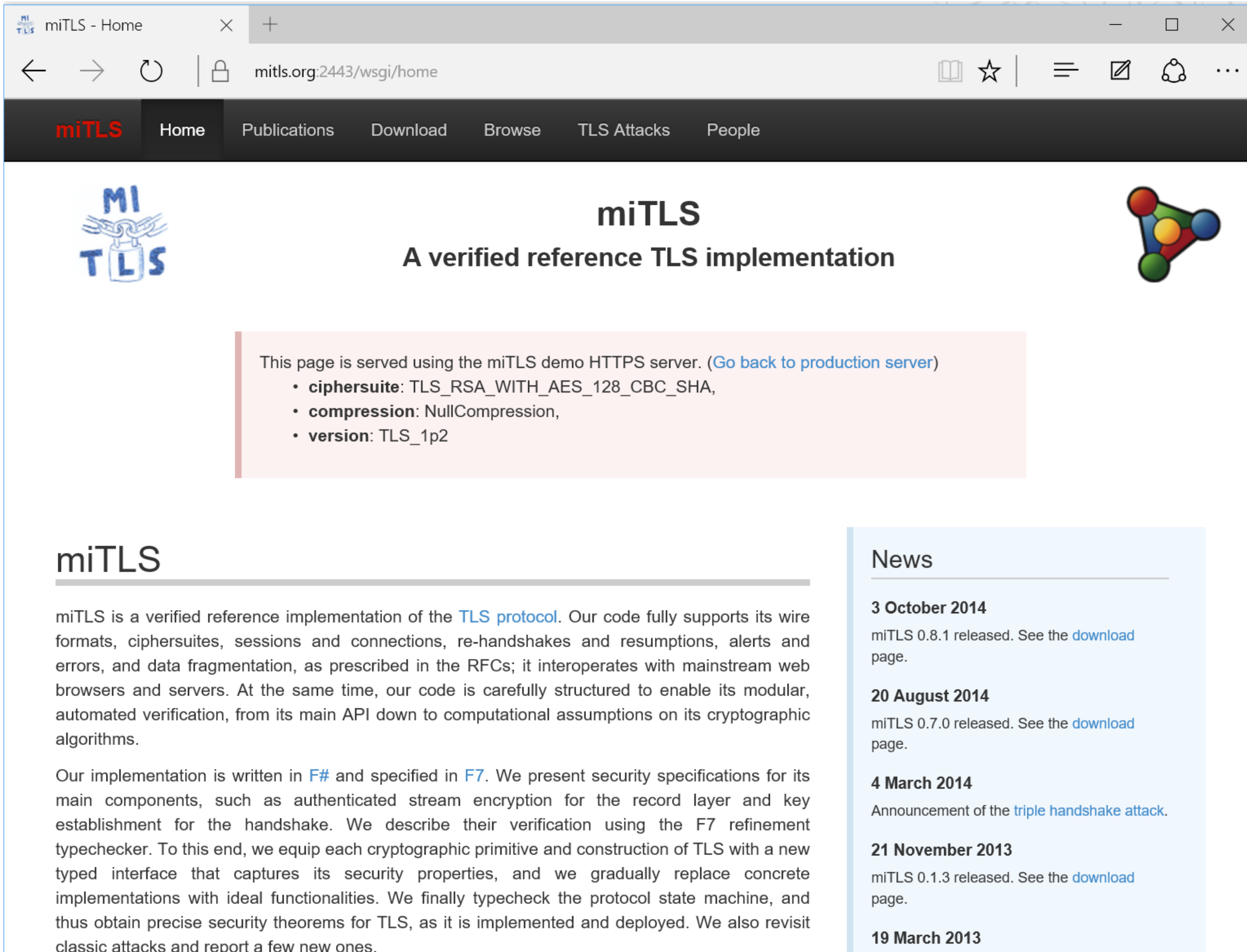
Weak Algorithms
MD5, PKCS1, RC4, …

TLS in F# & F7: miTLS

We develop and verify a **reference implementation** for SSL 3.0—TLS 1.2

1. **Standard compliance**: we closely follow the RFCs
   - concrete message formats
   - support for multiple ciphersuites, sessions and connections, re-handshakes and resumptions, alerts, message fragmentation,…
   - interop with other implementations such as web browsers and servers
2. **Verified security**: we structure our code to enable its modular verification, from its main API down to concrete assumptions on its base cryptography (e.g. RSA)
   - formal computational security theorems for a 5000-line functionality (automation required)
3. **Experimental platform:** for testing corner cases, trying out attacks, analysing new extensions and patches, …

https://www.mitls.org


miTLS - Home

mitls.org:2443/wsgi/home

miTLS | Home | Publications | Download | Browse | TLS Attacks | People

miTLS

A verified reference TLS implementation

This page is served using the miTLS demo HTTPS server. (Go back to production server)

- **ciphersuite**: TLS_RSA_WITH_AES_128_CBC_SHA,
- **compression**: NullCompression,
- **version**: TLS_1p2

miTLS

miTLS is a verified reference implementation of the TLS protocol. Our code fully supports its wire formats, ciphersuites, sessions and connections, re-handshakes and resumptions, alerts and errors, and data fragmentation, as prescribed in the RFCs; it interoperates with mainstream web browsers and servers. At the same time, our code is carefully structured to enable its modular, automated verification, from its main API down to computational assumptions on its cryptographic algorithms.

Our implementation is written in F# and specified in F7. We present security specifications for its main components, such as authenticated stream encryption for the record layer and key establishment for the handshake. We describe their verification using the F7 refinement typechecker. To this end, we equip each cryptographic primitive and construction of TLS with a new typed interface that captures its security properties, and we gradually replace concrete implementations with ideal functionalities. We finally typecheck the protocol state machine, and thus obtain precise security theorems for TLS, as it is implemented and deployed. We also revisit classic attacks and report a few new ones.

News

3 October 2014

miTLS 0.8.1 released. See the download page.

20 August 2014

miTLS 0.7.0 released. See the download page.

4 March 2014

Announcement of the triple handshake attack.

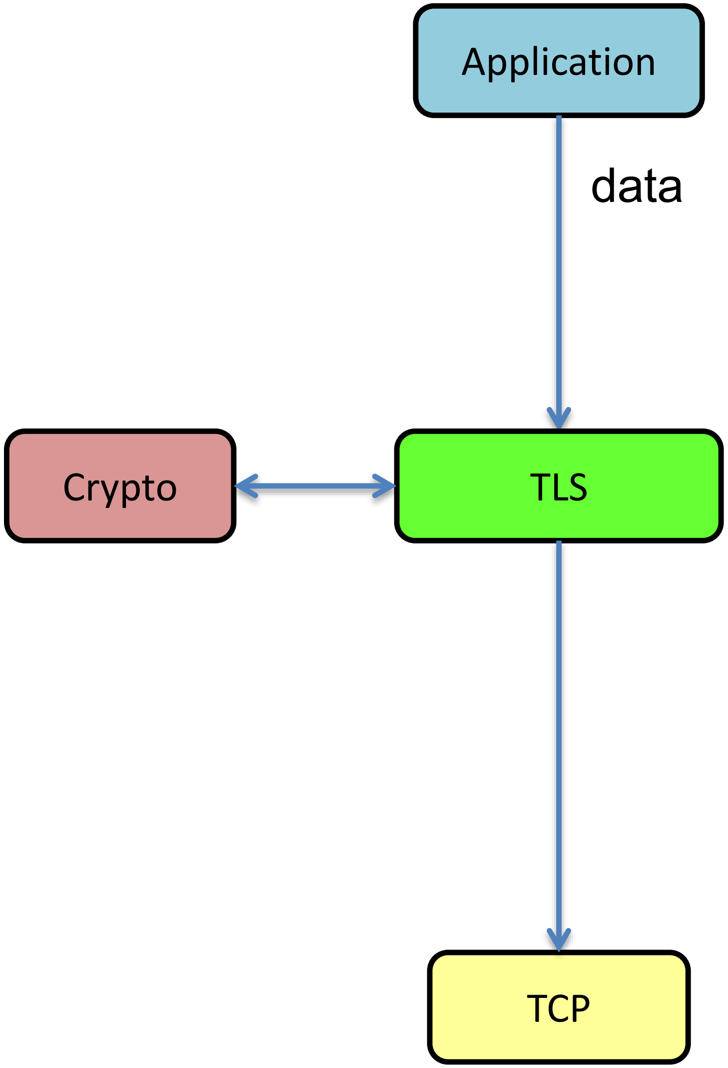
21 November 2013

miTLS 0.1.3 released. See the download page.

19 March 2013

TLS Security Goals, Informally

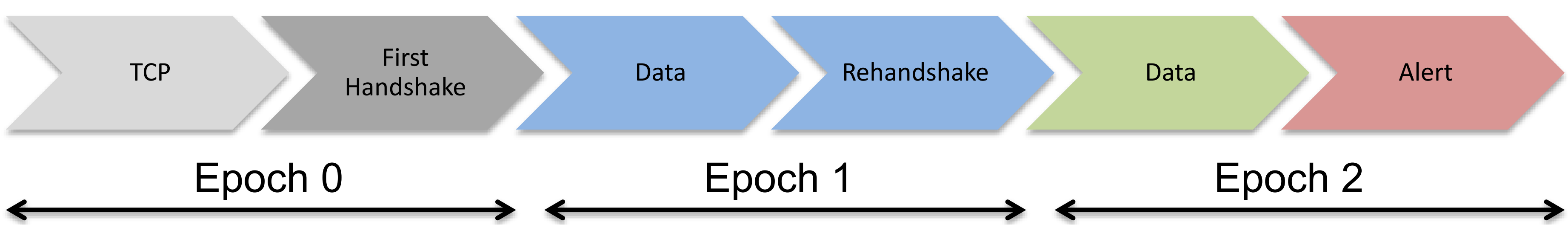
- Goals
  - Plaintext confidentiality
  - Server (and client) authentication
  - Stream integrity



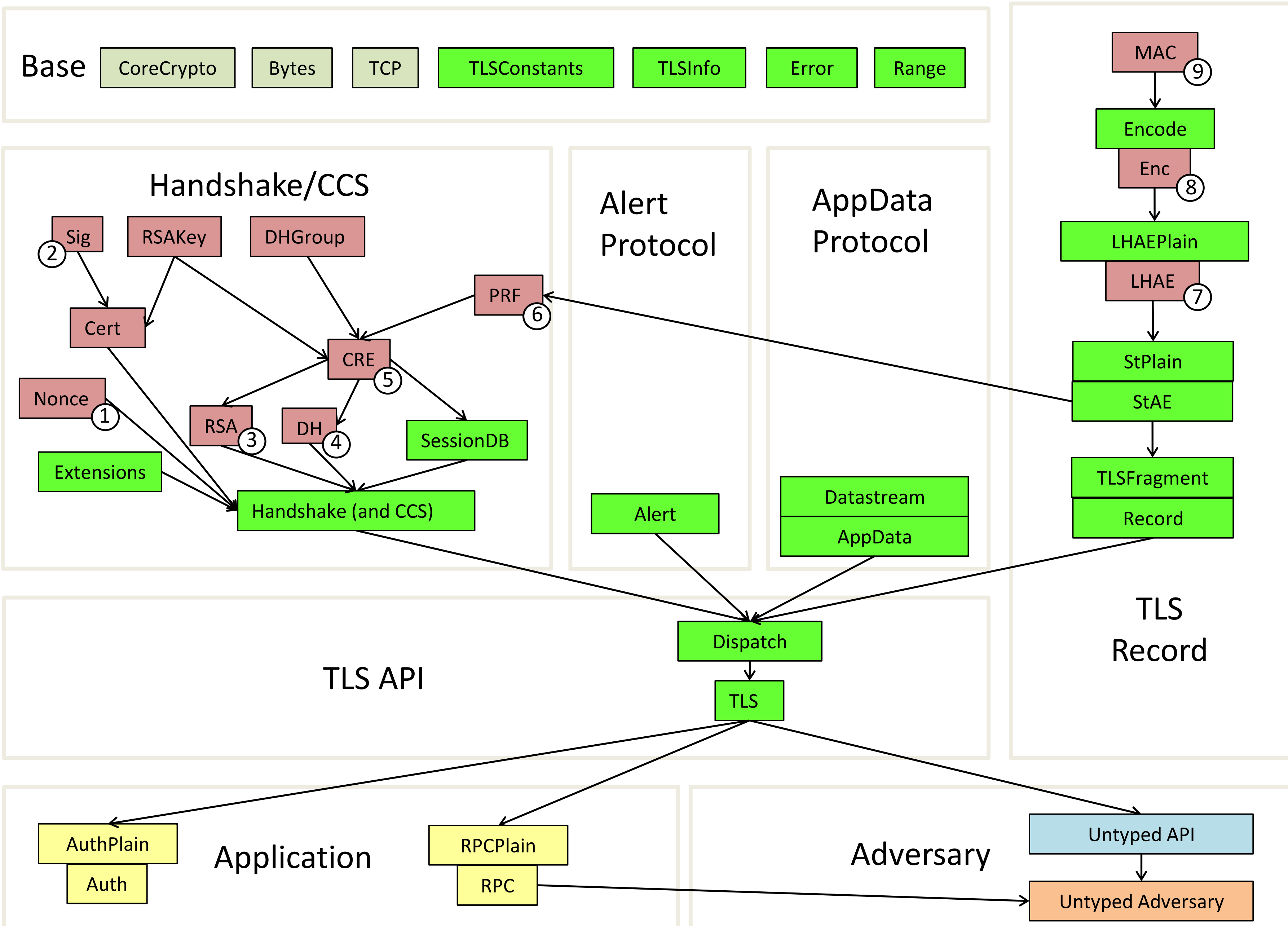
- Given a TLS connection with
  - Honest parties
  - Strong crypto algorithms
  - Recent protocol versions and extensions

Challenges

- Cryptographic agility
  - Ciphersuites, protocol versions
  - Some are weaker than others
  - Prove security for the negotiated parameters
- Complex state machines
  - Multiple epochs: initial handshake; resumption; renegotiation
  - Fragmentation
  - Specify and prove security invariants



Modular Architecture for miTLS



our main TLS API (outline)

Each application creates and runs session & connections in parallel

- Parameters select ciphersuites and certificates
- Results provide detailed information on the protocol state

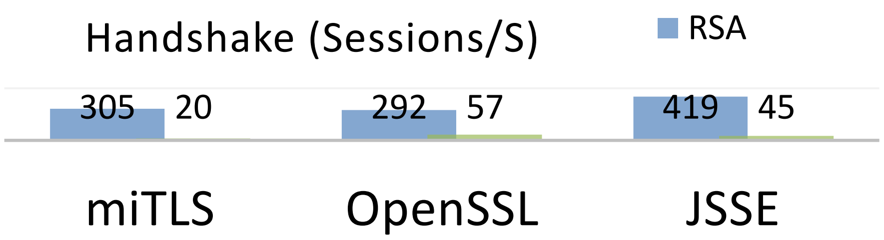
```
type cn // for each local instance of the protocol

// creating new client and server instances
val connect: TcpStream -> params -> (;Client) nullCn Result
val accept:  TcpStream -> params -> (;Server) nullCn Result

// triggering new handshakes, and closing connections
val rehandshake: c:cn{Role(c)=Client} -> cn Result
val request:     c:cn{Role(c)=Server} -> cn Result
val shutdown:    c:cn -> TcpStream Result

// writing data
type (;c:cn,d:(;c,OutStream(c)) data) ioresult_o =
| WriteComplete of c':cn
| WritePartial  of c':cn * rest:(;c',OutStream(c')) data
| MustRead      of c':cn
val write: c:cn -> d:(;c,OutStream(c)) data -> (;c,d) ioresult_o

// reading data
type (;c:cn) ioresult_i =
| Read      of c':cn * d:(;c,InStream(c)) data
| CertQuery of c':cn
| Handshake of c':cn
| Close     of TcpStream
| Warning   of c':cn * a:alertDescription
| Fatal     of a:alertDescription
val read : c:cn -> (;c) ioresult_i
```

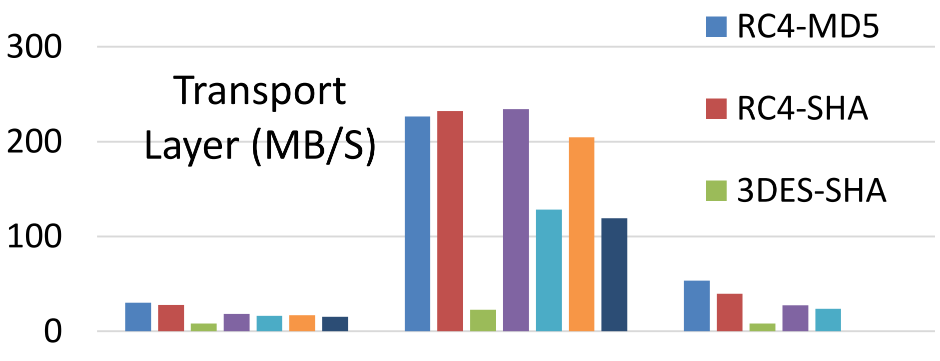
Interoperability & Performance

reference code vs production code

Sufficient for simple applications.

We miss system engineering:
custom memory manager,
crypto hardware acceleration,
low-level countermeasures…





miTLS: A Verified Reference Implementation for TLS

We get strong, usable, conditional application security

We trust...

1. verification tools: F7, Z3, EasyCrypt
   - now: mechanized theory using Coq/SSReflect
   - next: certified F* tools and SMT solver
2. cryptographic assumptions
   - now: concrete reductions using Easycrypt
   - next: mechanized proofs using relational probabilistic logic
3. the F# compiler and runtime: Windows and .NET
   - next: minimal TCB running e.g. on isolated core (SGX)
4. core cryptographic providers
   - next: correctness for selected algorithms (elliptic curves)

Milestone in **verified software**: cf Leroy's CompCert (2009) or Klein et al's L4.verified (2010)

Triple handshake attack

A Few Thoughts on Cryptographic Engineering

Some random thoughts about crypto. Notes from a course I teach. Pictures of my dachshunds.

Thursday, April 24, 2014

Attack of the Week: Triple Handshakes (3Shake)

The other day Apple released a major security update that fixes a number of terrifying things that can happen to your OS/X and iOS devices. You should install it. Not only does this fix a possible remote code execution vulnerability in the JPEG parser (!), it also patches a TLS/SSL protocol bug known as the "Triple Handshake" vulnerability. And this is great timing, since Triple Handshakes are something I've been meaning (and failing) to write about for over a month now.

3Shake logo designed by @R

But before we get there: a few points of order.

First, if Heartbleed taught us one thing, it's that when it comes to TLS vulnerabilities, *branding is key*. Henceforth, and with apologies to Bhargavan, Delign Lavaud, Pironti, Fournet and Strub (who actually discovered the attack*), for the rest of this be referring to the vulnerability simply as "3Shake". I've also taken the liberty of commissior logo. I hope you like it.

On a more serious note, 3Shake is not Heartbleed. That's both good and bad. It's good because Heartbleed was nasty and 3Shake really isn't anywhere near as dangerous. It's *bad* since, awful as it was, Heartbleed was only an implementation vulnerability -- and one in a single TLS library to boot. 3Shake represents a novel and fundamental bug in the TLS protocol.

The final thing you should know about 3Shake is that, according to the cryptographic literature, it shouldn't exist.

About Me

Matthew Green

I'm a cryptographer and research professor at

View my complete profile

Popular Posts

On the NSA

Let me tell you the story

Triple Handshakes and Cookie Cutters: Breaking and Fixing Authentication over TLS

Karthikeyan Bhargavan*, Antoine Delignat-Lavaud*, Cédric Fournet†, Alfredo Pironti* and Pierre-Yves Strub‡
*INRIA Paris-Rocquencourt †Microsoft Research ‡IMDEA Software Institute

***Abstract*—TLS was designed as a transparent channel abstraction to allow developers with no cryptographic expertise to protect their application against attackers that may control some clients, some servers, and may have the capability to tamper with network connections. However, the security guarantees of TLS fall short of those of a secure channel, leading to a variety of attacks.**

We show how some widespread false beliefs about these guarantees can be exploited to attack popular applications and defeat several standard authentication methods that rely too naively on TLS. We present new client impersonation attacks against TLS renegotiations, wireless networks, challenge-response protocols, and channel-bound cookies. Our attacks exploit combinations of RSA and Diffie-Hellman key exchange, session resumption, and renegotiation to bypass many recent countermeasures. We also demonstrate new ways to exploit known weaknesses of HTTP over TLS. We investigate the root causes for these attacks and propose new countermeasures. At the protocol level, we design and implement two new TLS extensions that strengthen the authentication guarantees of the handshake. At the application level, we develop an exemplary HTTPS client library that implements several mitigations, on top of a previously verified TLS implementation, and verify that their composition provides strong, simple application security.

sessions, validating certificates, etc. Meanwhile, TLS applications continue to rely on URLs, passwords, and cookies; they mix secure and insecure transports; and they often ignore lower-level signals such as handshake completion, session resumption, and truncated connections.

Many persistent problems can be blamed on a mismatch between the authentication guarantees expected by the application and those actually provided by TLS. To illustrate our point, we list below a few myths about those guarantees, which we debunk in this paper. Once a connection is established:

1) the principal at the other end cannot change;
2) the master secret is shared only between the two peers, so it can be used to derive fresh application-level keys;
3) the `tls-unique` channel binding [6] uniquely identifies the connection;
4) the connection authenticates the whole data stream, so it is safe to start processing application data as it arrives.

The first is widely believed to be ensured by the TLS renegotiation extension [49]. The second and third are used for man-in-the-middle protections in tunneled protocols like PEAP and some authentication modes in SASL and GSS-API. The fourth

F* - Latest in an Evolution of Languages

- Symbolic and computational models for cryptography (F7)
- A type-preserving compiler to .NET bytecode (Fine)
- Security of an implementation of the TLS 1.2 standard (F7)
- Self-certification: Certifying F* using F* and Coq
- A fully abstract compiler from F* to JavaScript
- TS*: An embedded, secure subset of TypeScript
- RF*: Probabilistic relational logic for verified cryptography
- F* v1.0:
  Open source, programmed entirely in F*, bootstrapped in OCaml and F#.
  More streamlined, expressive, and efficient than prior versions.

Summary of Lecture 3

- We consider applications of F7, its successor F*, and adaptations of this work to programs in C
- Plenty of scope to adapt these techniques to other applications of cryptographic programming!

#fosad2015