Verifying Security Protocols and their Implementations

Information Security and Cryptography Reading Group

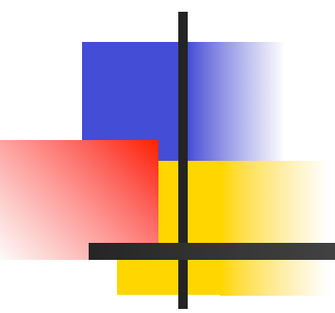
Presenter: Cătălin Hriţcu

References

- Verified Interoperable Implementations of Security Protocols. Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, Stephen Tse, CSFW 2006. (and accompanying technical report with proofs)
- Most of the slides are from
  - Protecting Alice from Malice: Protocols, Process Calculi, Proved Programs. Andy Gordon, Marktoberdorf, 2006.
  - Vérification de Protocoles Cryptographiques et de leurs Implémentations. Cédric Fournet, Colloquium INRIA, 2007.
  - Language-based Security. Matteo Maffei, Lecture, 2007.
  - Many thanks to Andy Gordon and Cédric Fournet (Microsoft Research), and Matteo Maffei

Outline

- Verifying Security Protocols
  - What are security protocols?
  - What are the properties we want to verify?
  - Why is it so difficult?
  - Formal methods
    - The Dolev-Yao model
    - Modeling protocols using process calculi
    - Specifying and analyzing security properties
- Verifying Implementations of Security Protocols



Verifying Security Protocols

Security Protocols

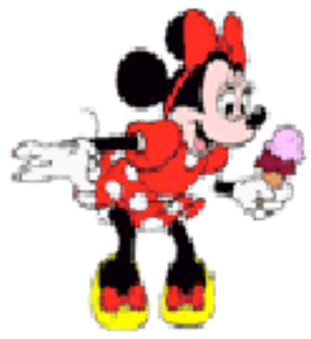
- Modern applications heavily rely on secure communication over an untrusted network
  - Malicious entities can read, block, modify, (re)transmit messages

Security Properties

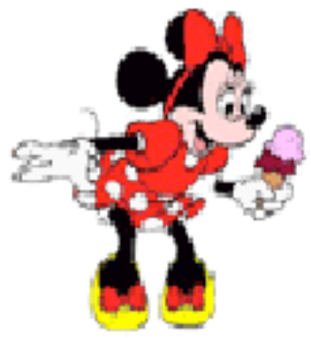
- The exact notion of security depends on application
- In this talk we focus on two simple properties:
  - Secrecy
    - Confidential information not disclosed to third-parties
  - Authentication
    - The recipient of a message should be able to verify its freshness and its originator

A Simple eBanking Protocol

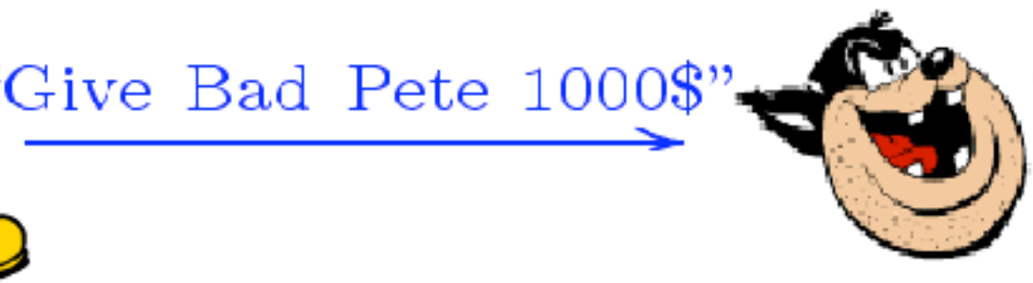
"Give Bad Pete 1000$"



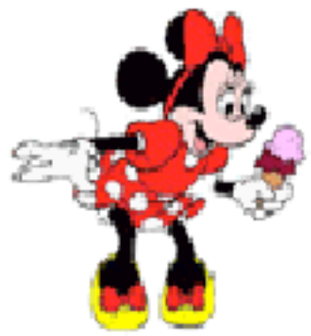
A Simple eBanking Protocol



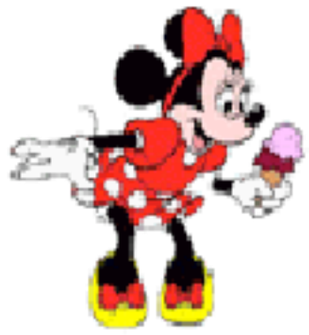
- Bad Pete intercepts the message and modifies it in order to get 2000$



A Simple eBanking Protocol

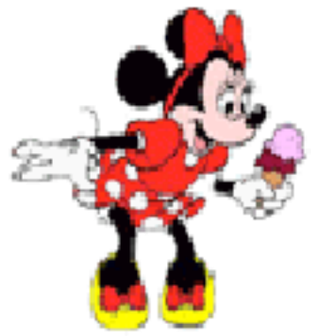


- Bad Pete intercepts the message and modifies it in order to get 2000$



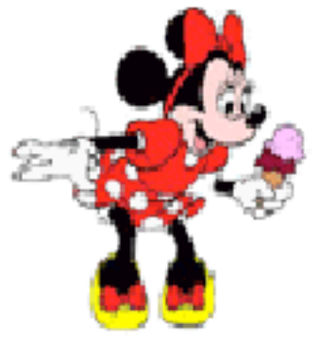
Cryptography

$\{\text{“Give Bad Pete 1000\$”}\}_k$



Cryptography

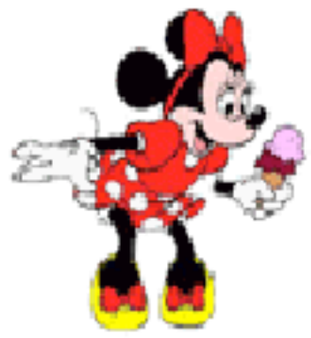
$\{\text{“Give Bad Pete 1000\$”}\}_k$



- Unfortunately, cryptography is not enough!
- Attacker can break security of the protocol
  - by simply intercepting, duplicating, sending back the messages in transit on the network, by interleaving simultaneous protocol sessions, etc
  - no need to break the encryption scheme
- In the following, we assume that cryptography is a fully reliable black box

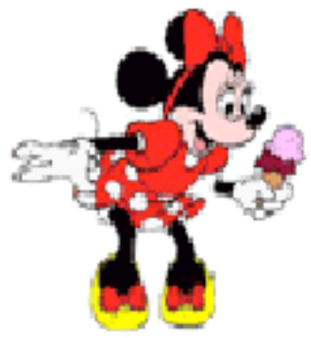
Reflection Attack

$\{\text{“Give Bad Pete 1000\$”}\}_k$



Reflection Attack

$\{\text{“Give Bad Pete 1000\$”}\}_k$



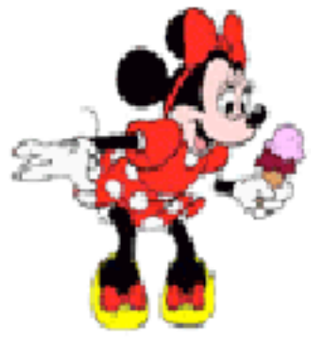
$\{\text{“Give Bad Pete 1000\$”}\}_k$

$\{\text{“Give Bad Pete 1000\$”}\}_k$

- The symmetric nature of the encryption key k does not allow Mickey to verify whether the encrypted message has been generated by Minnie or himself

Replay attack

$\{Mickey, \text{“Give Bad Pete 1000\$”}\}_k$



Replay attack

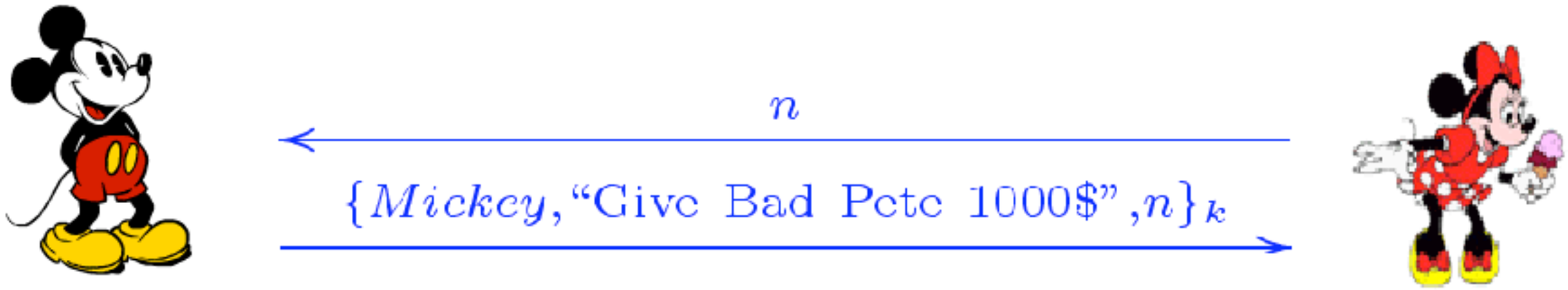
$\{Mickey, \text{“Give Bad Pete 1000\$”}\}_k$

$\{Mickey, \text{“Give Bad Pete 1000\$”}\}_k$

- Minnie has no way to verify the freshness of the message she receives

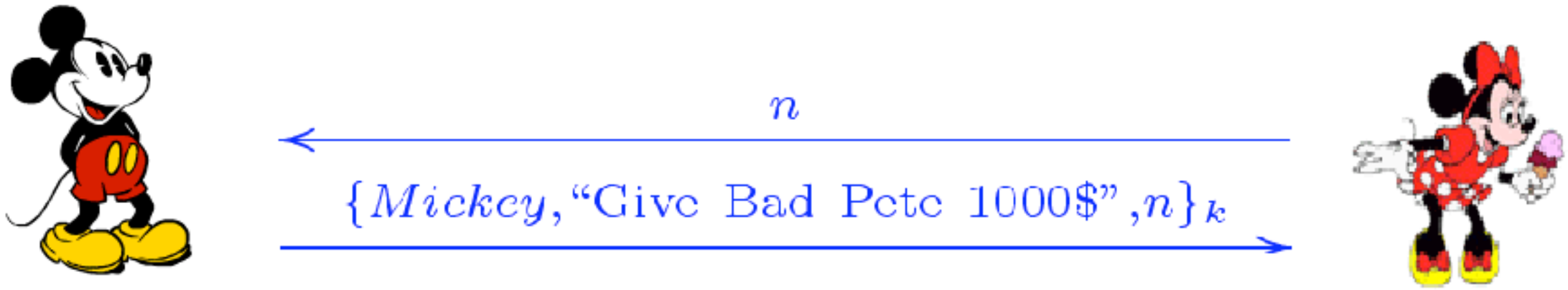
Fixed Protocol

- Possible solution is to use a nonce
  - Randomly generated number used only once

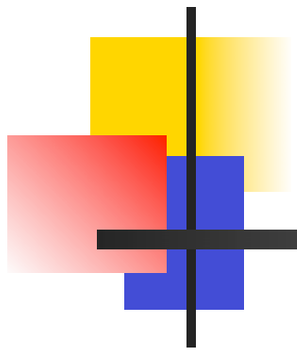


Fixed Protocol

- Possible solution is to use a nonce
  - Randomly generated number used only once



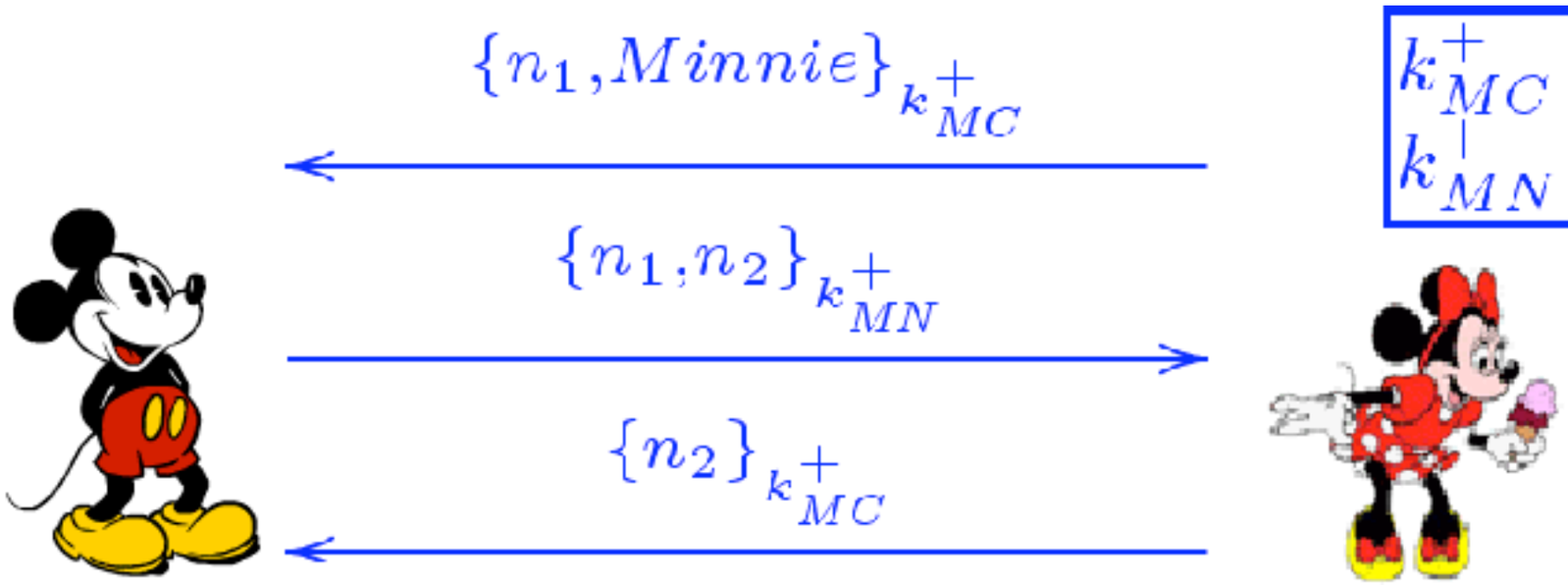
- This protocol is secure, it guarantees
  - the secrecy and authenticity of the message
- ISO two-pass unilateral authentication protocol



Protocols Are Hard to Get Right

- Historically, one keeps finding simple attacks against protocols
  - even carefully-written, widely-deployed protocols
  - even a long time after the design and deployment
- New protocols appear regularly, and the same mistakes are made again and again
  - Attacks on Web Services security
  - Recent MITM attack on public-key Kerberos (2005)
- What's so difficult about security protocols?
  - concurrency + distribution + cryptography
  - little control on the runtime environment
  - hard to test against active attackers

Needham-Schroeder Protocol



k^+_{MC}	Mickey's public key
k^+_{MN}	Minnie's public key

- This protocol was proposed in 1978
- The aim is to guarantee the secrecy and authenticity of the two nonces (later used for generating a symmetric session-key)

Man-in-the-middle Attack

- Found by Lowe in a 1995

$\leftarrow \{Minnie, n_1\}_{k^+_{MC}}$ ——

—— $\{n_1, n_2\}_{k^+_{MN}} \rightarrow$

$\leftarrow$ —— $\{n_2\}_{k^+_{MC}}$ ——

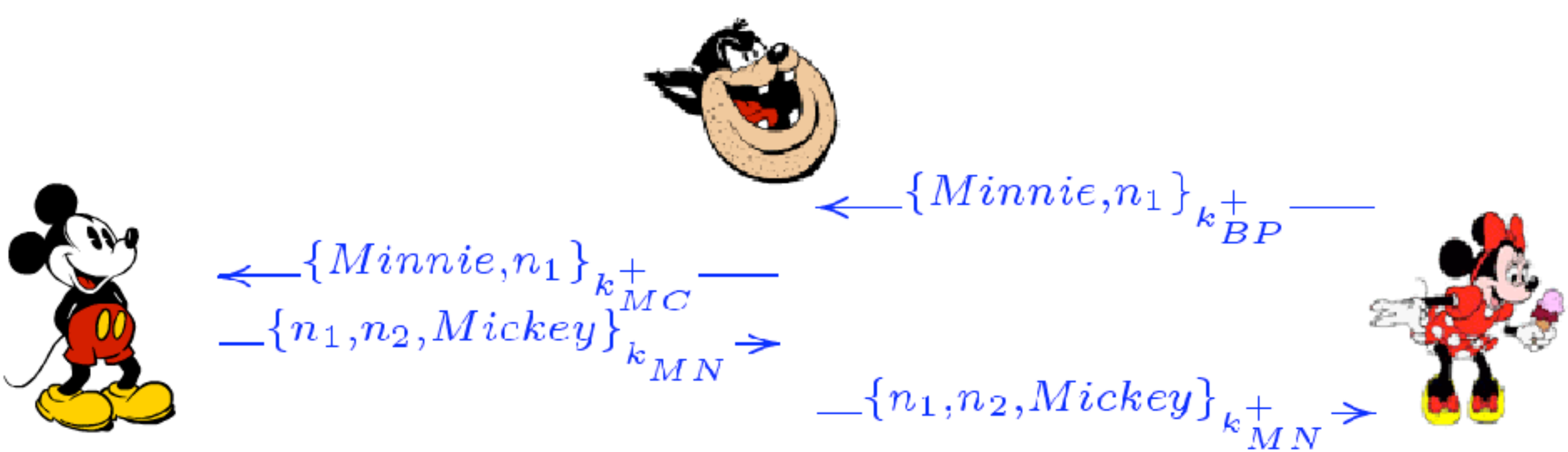
$\leftarrow \{Minnie, n_1\}_{k^+_{BP}}$ ——

—— $\{n_1, n_2\}_{k^+_{MN}} \rightarrow$

$\leftarrow$ —— $\{n_2\}_{k^+_{BP}}$ ——

Lowe's Fix

- Insert Mickey's identifier in the second message
- Rules out the man-in-the-middle attack



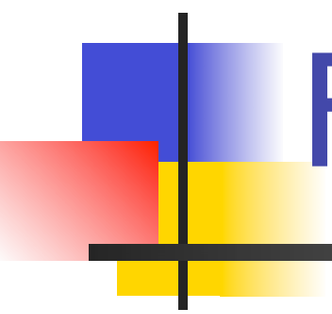
Informal Methods

- Principles codified in articles and textbooks since mid-90s:
  - Abadi and Needham, Prudent engineering practice for cryptographic protocols, 1994
  - Anderson and Needham, Programming Satan's Computer, 1995

Principle 1

Every message should say what it means: the interpretation of the message should depend only on its content. It should be possible to write down a straightforward English sentence describing the content — though if there is a suitable formalism available that is good too.

- For instance, Lowe's fix of the Needham-Schroeder protocol makes explicit that the second message is sent by Mickey
- These check lists are useful; yet hard for the inexperienced to understand and to apply
- No guarantee that they will make your protocol secure



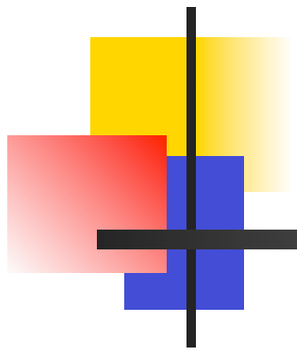
Formal Methods

Dolev-Yao Attacker Model

- Widely-used abstraction
  - Because of automatic tool support
- The attacker can:
  - Engage in any number of protocol sessions with any number of many honest principals (unbounded)
  - Read, block, modify, reply any message sent on the network (the attacker **is** the network)
  - Split composite messages, recompose arbitrarily
  - Encrypt messages in arbitrary ways
  - Decrypt encrypted messages with the appropriate key
  - No bound on the size of the messages, or number of fresh nonces and keys

Dolev-Yao Attacker Model

- Strong assumptions
  - Perfect cryptography, the attacker cannot:
    - Decrypt a message without knowing the encryption key
    - Guess or brute-force keys, nonces or even passwords
    - Obtain partial information (e.g. half the bits of a message)
    - 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
- In cryptography attacker is PPT that tries to break security with non-negligible probability (computational model)
- Justifying Dolev-Yao style symbolic models via computational models is sometimes possible



Protocols as Processes

- Security protocols in the Dolev-Yao model can be rephrased as processes in process calculi
  - E.g. spi calculus, applied pi-calculus etc.
- Their security properties can be rigorously formalized
- There are many applicable formalisms for analyzing these properties automatically
  - Type (and effect) systems, type inference
  - Abstract interpretation
    - E.g. abstract processes to Horn clauses (Prolog rules) then use resolution
  - Model checking (bounded or symbolic)

Applied Pi-calculus (ProVerif)

x, y, z	variable
a, b	name
f	constructor
g	destructor function
$M, N ::=$	value
x	variable
a	name
$f(M_1, \ldots, M_n)$	constructor application

- constructors: enc, pair
- value: enc(pair(pair(id, m), n), k)
- destructors: dec, left, right

Processes

$P,Q,R ::=$	process
$\quad \textbf{in}(M,x);P$	$\quad$ input of x from M
$\quad \textbf{out}(M,N);P$	$\quad$ output of N on M
$\quad \textbf{new}\ a;P$	$\quad$ make new name a
$\quad !P$	$\quad$ replication of P
$\quad P \mid Q$	$\quad$ parallel composition
$\quad \mathbf{0}$	$\quad$ inactivity
$\quad \textbf{event}\ M$	$\quad$ event M
$\quad \textbf{let}\ x_1,\ldots,x_n\ \textbf{suchthat}\ M = N\ \textbf{in}\ P\ \textbf{else}\ Q$	$\quad$ match
$\quad \textbf{let}\ x = g(M_1,\ldots,M_n)\ \textbf{in}\ P\ \textbf{else}\ Q$	$\quad$ destructor application

Declarations and Scripts

$\Delta ::=$	declaration
free a	name a
data f/n	data constructor
[**private**]**fun** f/n	[private]constructor
reduc $g(M_1,\ldots,M_n) = M$	destructor
[**private**]**reduc** $g(M_1,\ldots,M_n) = M$	[private]destructor
$\Delta s ::= \Delta_1.\cdots\Delta_n.$	set of declarations ($n \geq 0$)
$\Sigma ::= \Delta s$ **process** P	script

- Example

```
fun enc / 2. fun pair / 2.
reduc dec(enc(x,y),y) = x.
reduc left(pair(x,y)) = x.
reduc right(pair(x,y)) = y.
```

Structural Equivalence

$$P \equiv P$$
$$Q \equiv P \Rightarrow P \equiv Q$$
$$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$$
$$P \mid \mathbf{0} \equiv P$$
$$P \mid Q \equiv Q \mid P$$
$$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$!P \equiv P \mid !P$$
$$a \notin fn(P) \Rightarrow \textbf{new}\ a; (P \mid Q) \equiv P \mid \textbf{new}\ a; Q$$
$$\textbf{new}\ a; \textbf{new}\ b; P \equiv \textbf{new}\ b; \textbf{new}\ a; P$$
$$\textbf{new}\ a; \mathbf{0} \equiv \mathbf{0}$$
$$P \equiv P' \Rightarrow \textbf{new}\ a; P \equiv \textbf{new}\ a; P'$$
$$P \equiv P' \Rightarrow P \mid R \equiv P' \mid R$$

Internal Reduction

$$P \equiv Q, Q \to Q', Q' \equiv P' \Rightarrow P \to P'$$

$$P \to P' \Rightarrow P \mid Q \to P' \mid Q$$

$$P \to P' \Rightarrow \textbf{new}\ a; P \to \textbf{new}\ a; P'$$

$$\textbf{in}(M, x); P \mid \textbf{out}(M, N); Q \to P\{N/x\} \mid Q$$

$$\textbf{let}\ x_1, \ldots, x_n\ \textbf{suchthat}\ M = N\ \textbf{in}\ P\ \textbf{else}\ Q \to \begin{cases} P\sigma & \text{if } M = N\sigma \text{ and } dom(\sigma) = \{x_1, \ldots, x_n\} \\ Q & \text{otherwise} \end{cases}$$

$$\textbf{let}\ x = g(M'_1, \ldots, M'_n)\ \textbf{in}\ P\ \textbf{else}\ Q \to \begin{cases} P\{M\sigma/x\} & \text{if } M'_i = M_i\sigma \text{ for all } i \in 1..n \\ Q & \text{otherwise} \end{cases}$$

where $(g(M_1, \ldots, M_n) = M)$ declared in Δs_a

Modeling our protocol

$$n$$

$$\{Mickey, \text{“Give Bad Pete 1000\$”}, n\}_k$$

free ch.
fun enc / 2. **data** pair / 2. **data** mickey / 0. **data** minnie / 0.
private fun begin / 1. **private fun** end / 1.
reduc dec(enc(x,y),y) = x.
reduc left(pair(x,y)) = x.
reduc right(pair(x,y)) = y.

let Mickey = **new** m; in(ch,xn); **event** begin(mickey, minnie, xn);
  out(ch, enc(pair(pair(mickey, m), xn), k)).

let Minnie = **new** n; **out**(ch,n); **in**(ch, x); **let** y = dec(x, k) **in**
  let xm **suchthat** y = pair(pair(mickey, xm), n) **in event** end(mickey, minnie, n).

process new k; Mickey | Minnie

Correspondence Properties

Query Satisfaction and Safety:

$P \models \mathbf{ev}{:}E \Rightarrow \mathbf{ev}{:}B_1 \vee \cdots \vee \mathbf{ev}{:}B_n$ if and only if whenever $P \equiv \mathbf{new}\ as; (\mathbf{event}\ E\sigma \mid P')$, we have $P' \equiv \mathbf{event}\ B_i\sigma \mid P''$ for some $i \in 1..n$ and some P''.

A process P is *safe for* q if and only if, for all reductions $P \rightarrow_{\equiv}^{*} P'$, we have $P' \models q$.

Opponent Processes and Robust Safety:

A Δs-*opponent* is a process O with no events such that Δs **process** O is well formed and O contains no constructor or destructor declared **private** in Δs.

A script Δs **process** P is *robustly safe for* q if and only if
for all Δs-opponents O, $P \mid O$ is safe for q.

verification tool: ProVerif

- ProVerif, an automated cryptographic protocol verifier developed by Bruno Blanchet (ENS)
- Input: protocol scripts written in applied pi calculus
  - Concurrent processes + parametric cryptography
- What it can prove:
  - Secrecy, authenticity (correspondence properties)
  - Equivalences (e.g. protection of weak secrets)
- How it works (flavor):
  - Internal representation based on Horn clauses
  - Resolution-based algorithm, with clever selection rules
  - Attack reconstruction

B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 82–96. IEEE Computer Society Press, 2001.

B. Blanchet, M. Abadi, and C. Fournet. Automated verification of selected equivalences for security protocols. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS'05)*, 2005.

Queries in ProVerif

- query attacker: k.

  Starting query not attacker:k[]
  RESULT not attacker:k[] is true.

- query ev: end(x,y,z) ==> ev: begin(x,y,z).

  Starting query ev:end(x_15,y_16,z_17) ==>
  ev:begin(x_15,y_16,z_17)

  goal reachable: begin:begin(mickey(),minnie(),n[]) ->
  end:end(mickey(),minnie(),n[])

  RESULT ev:end(x_15,y_16,z_17) ==>
  ev:begin(x_15,y_16,z_17) is true.

2007: mission accomplished

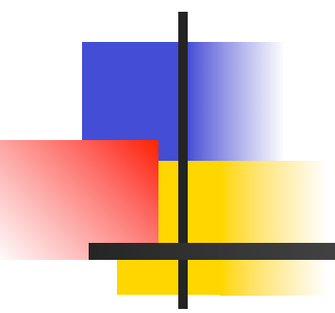
- Authentication and secrecy properties for crypto protocols have been formalized and thoroughly studied
- After intense effort on symbolic reasoning, several techniques and tools are available for automatically proving these properties
  - e.g. Athena, TAPS, ProVerif, FDR, AVISPA, etc
- We can now automatically verify most security properties for detailed models of crypto protocols
  - e.g. IPSEC, Kerberos, Web Services, Infocard

2007: mission accomplished?

- Best practice: apply formal methods and tools throughout the protocol design & review process
- Not so easy
  - Specifying a protocol is a lot of work
  - Most practitioners don't understand formal models
- Protocols go wrong because...
  - they are logically flawed, or
  - they are used wrongly, or
  - they are wrongly implemented

2007: mission accomplished?

- Best practice: apply formal methods and tools throughout the protocol design & review process
- Not so easy
  - Specifying a protocol is a lot of work
  - Most practitioners don't understand formal models
- Protocols go wrong because...
  - they are logically flawed, or
  - they are used wrongly, or
  - they are wrongly implemented
- Q: How to relate formal models to executable code?



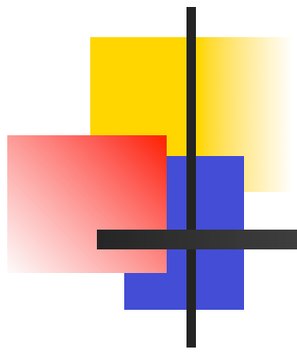
Verifying Implementations of Security Protocols

The Trouble with Models

- Protocol designers are typically reluctant to write formal models - specs are natural language
- Specs are always refined by implementation experience, so absolute correctness is not a goal
  - Timely agreement is more important
  - So specs continue to be partial and ambiguous
    - In spite of successful research on model verification, new specs exhibit the same old mistakes, again and again (e.g. the attack on Kerberos 5 from 2005)
- The Ugly Reality: implementation code is the closest we get to a formal description of most protocols

The Trouble with Models (2)

- Formal models are short and abstract
  - 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?

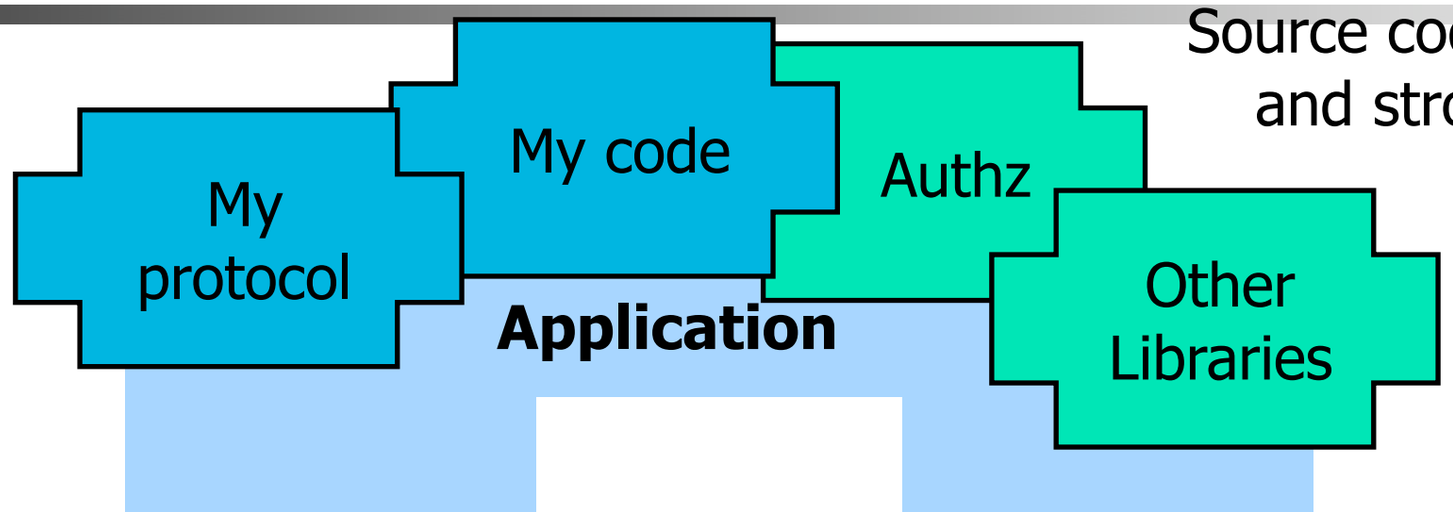


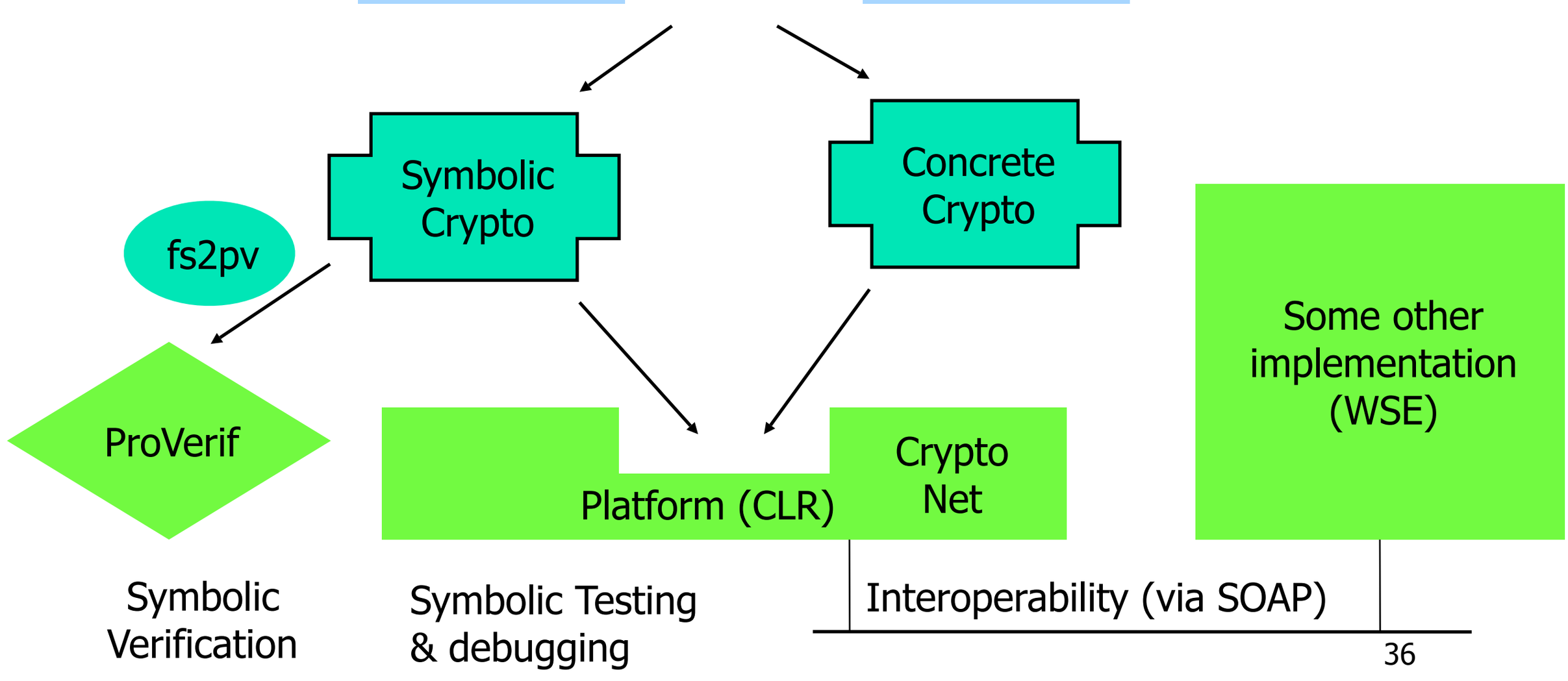
From Code to Model

- Automatically extract models from interoperable protocol implementations
  - Analyze model automatically using existing tool
  - Reference implementations, not (yet) production code
- Largely avoids potential mismatches between model and implementation
- 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
  - Type checkers, compilers, debuggers, libraries, other verification tools

One Source, Three Tasks

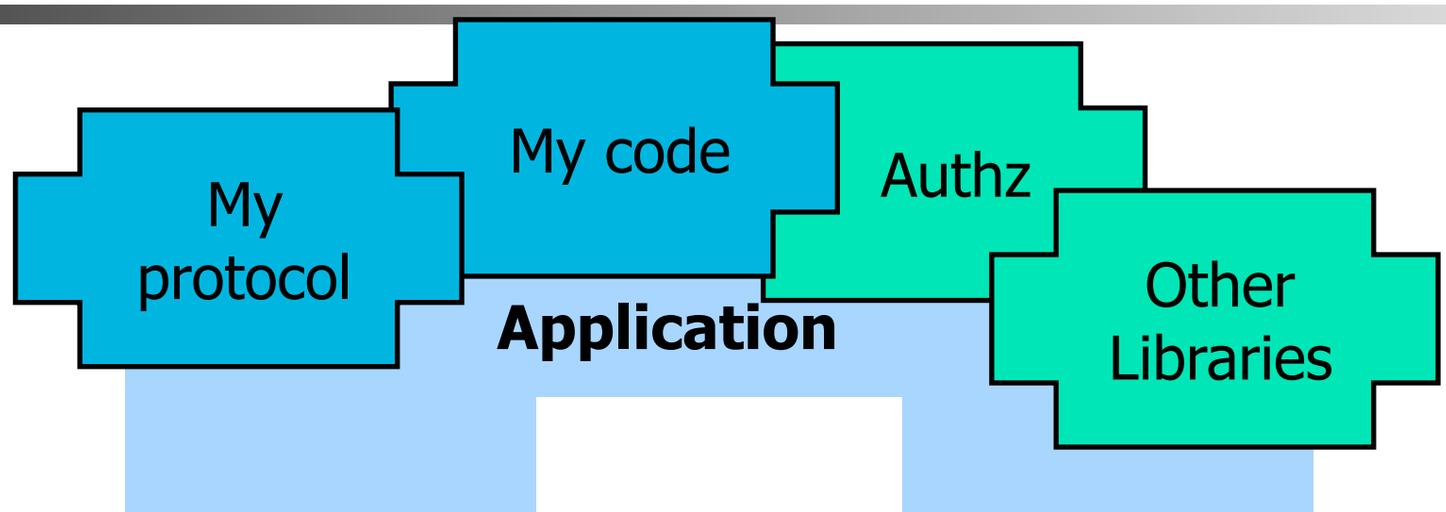
Source code with modules and strong interfaces





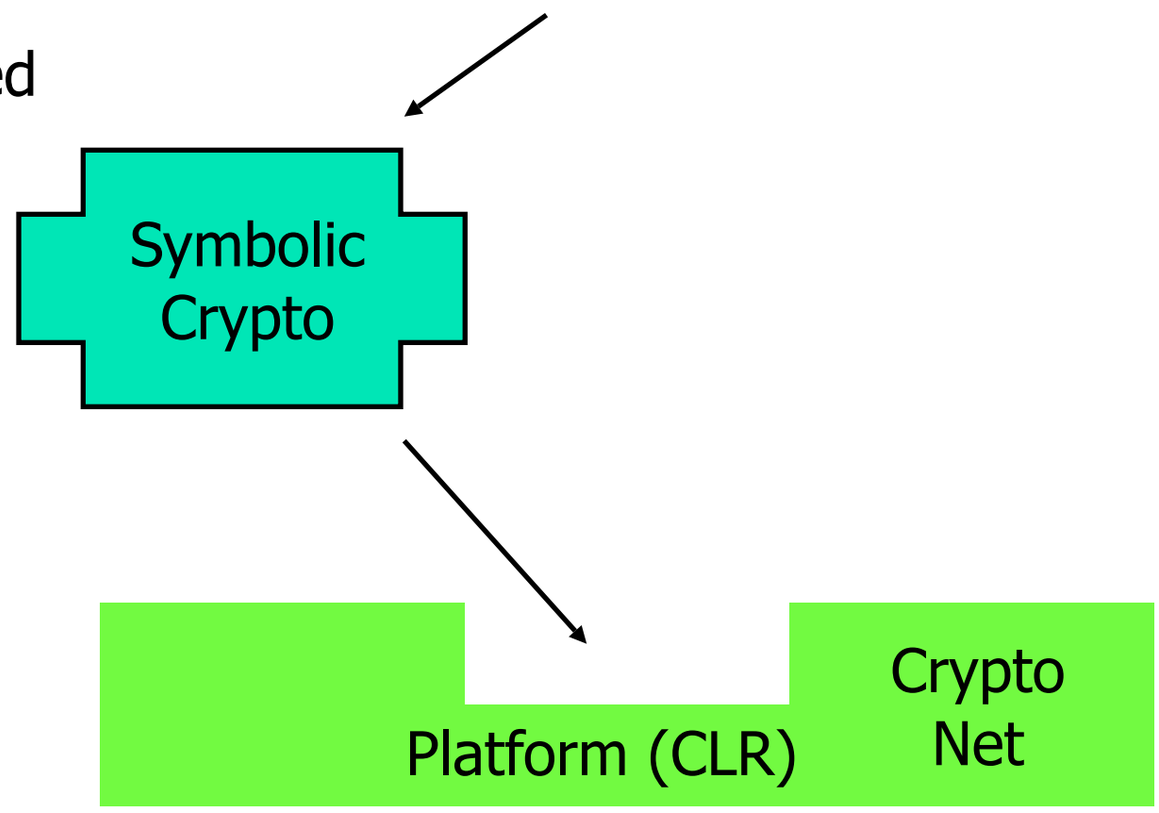
1. Symbolic testing and debugging

Coded in C#, F#...



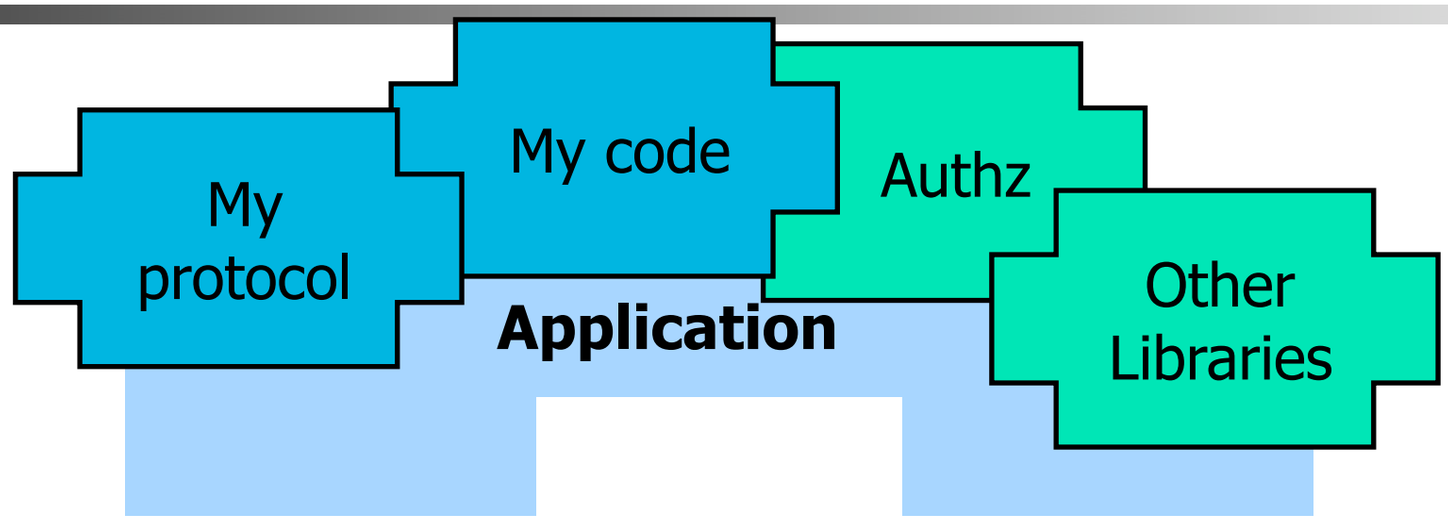
We use idealized "black-box" cryptographic primitives

Safety relies on typing



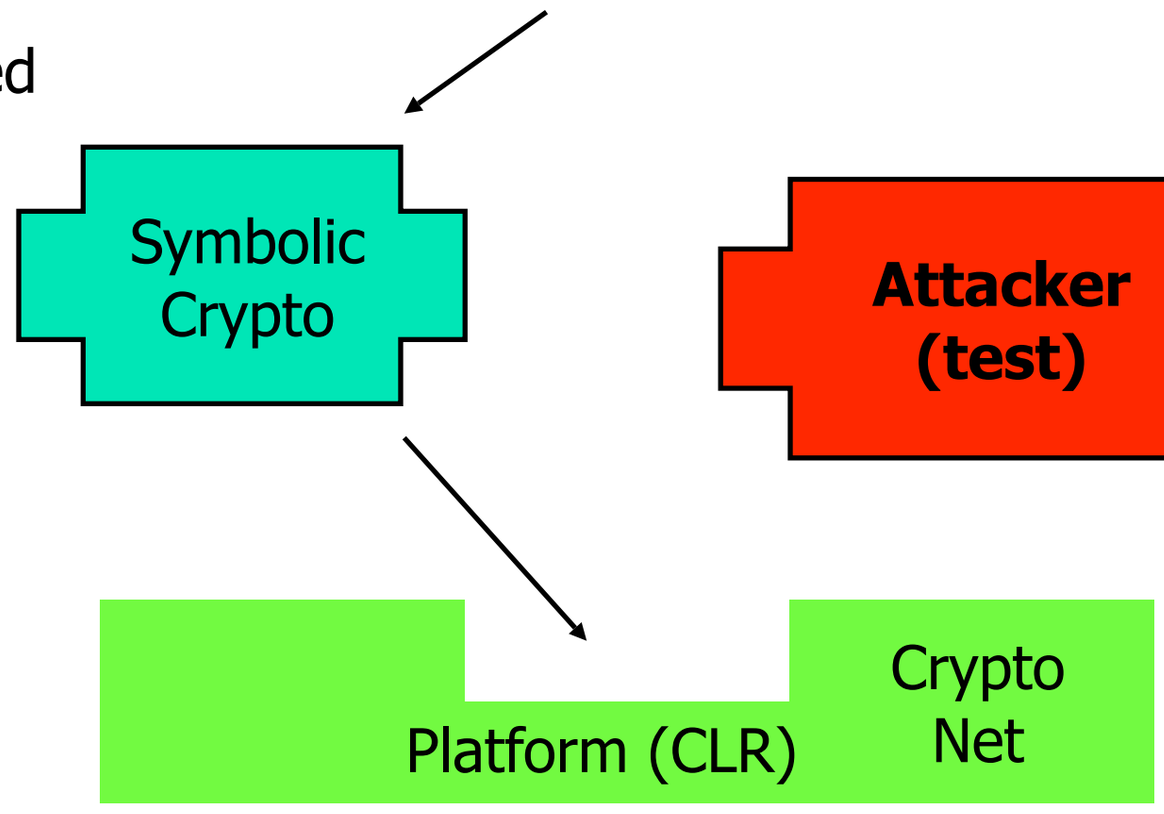
1. Symbolic testing and debugging

Coded in C#, F#...



We use idealized "black-box" cryptographic primitives

Safety relies on typing

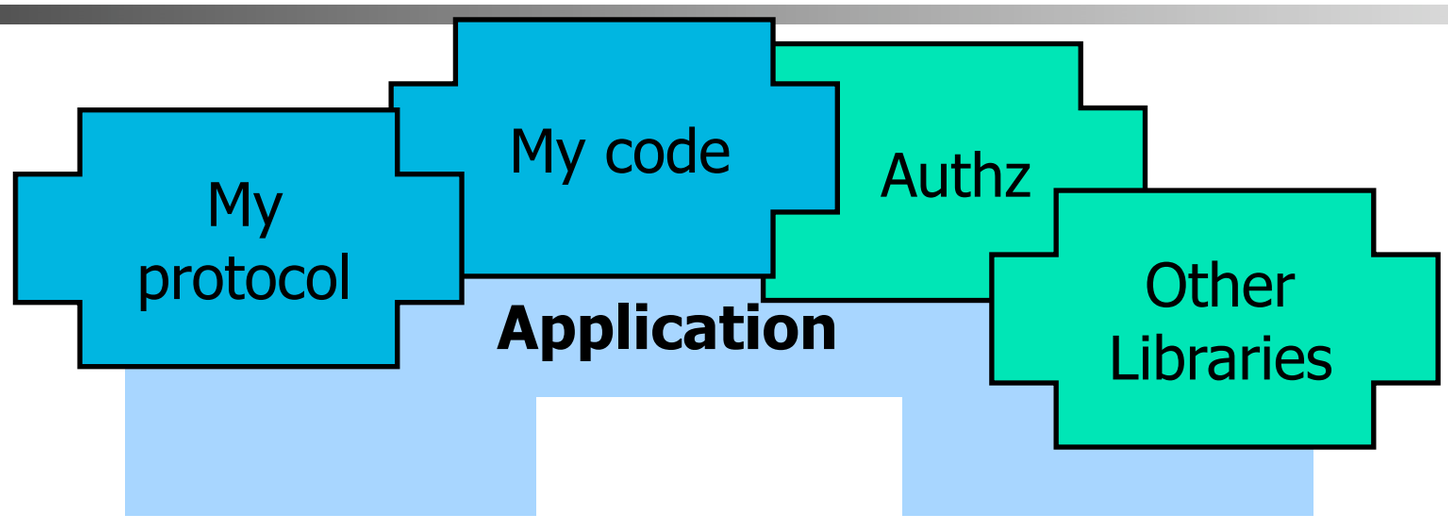


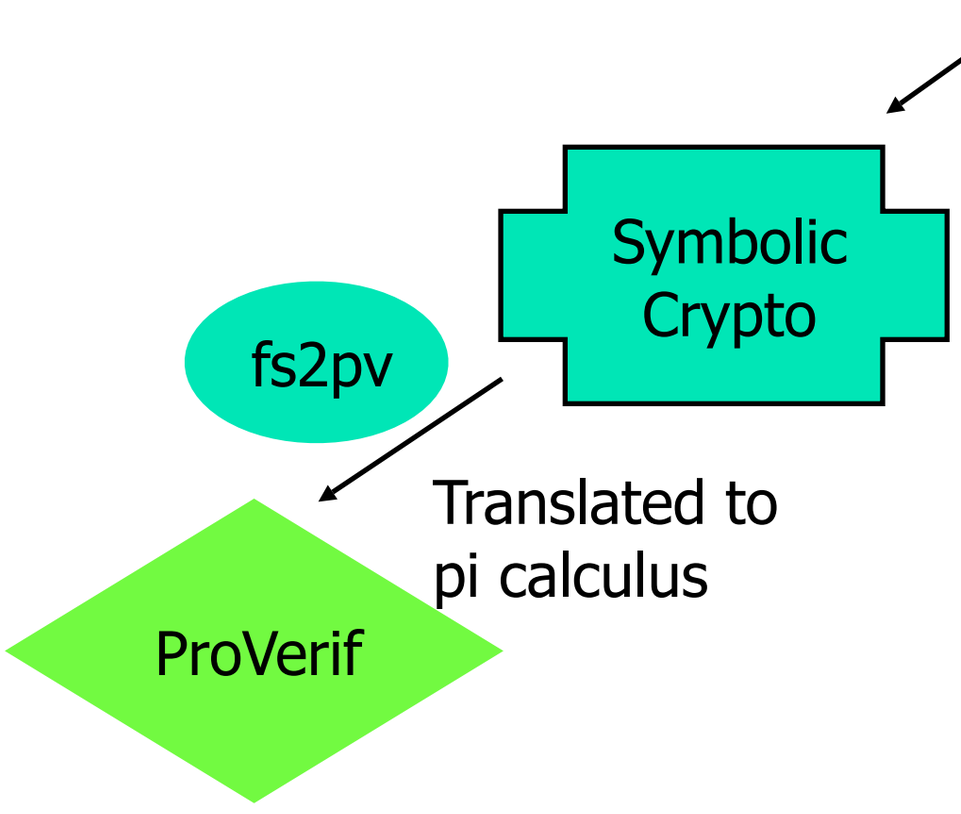
We model attackers as arbitrary code with access to selected libraries

We can code any given potential attack as a program

2. Formal Verification

We only support a subset of F#





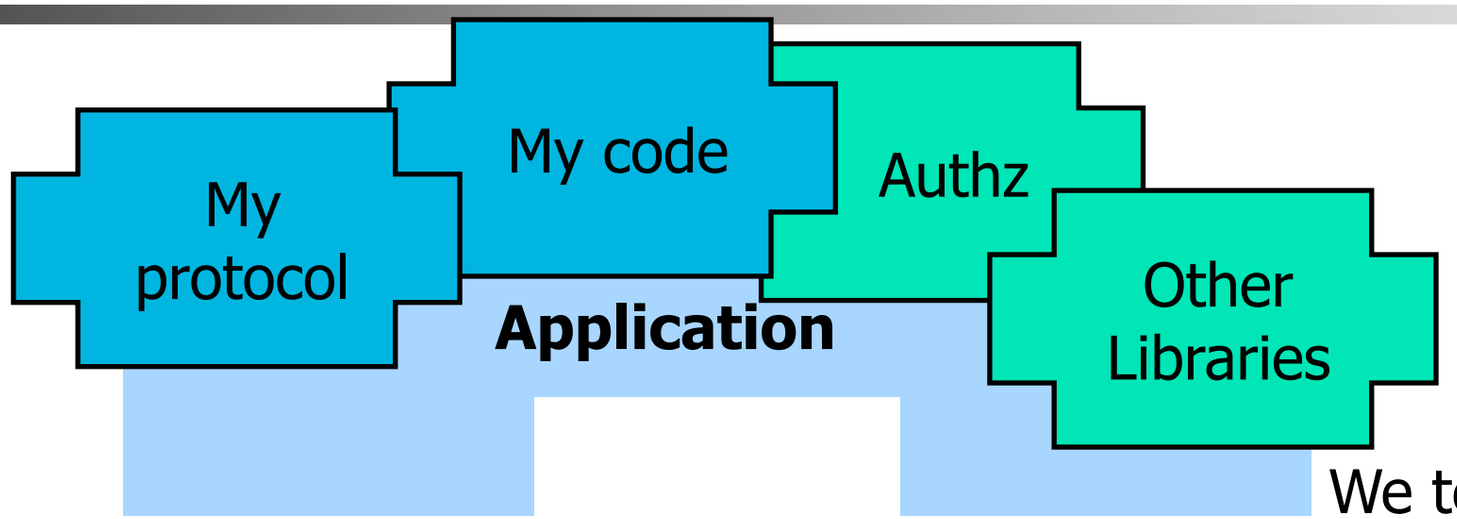
Pass: Ok for all attackers, or
No + potential attack trace

We model attackers as arbitrary code with access to selected libraries

Formal verification considers ALL such attackers

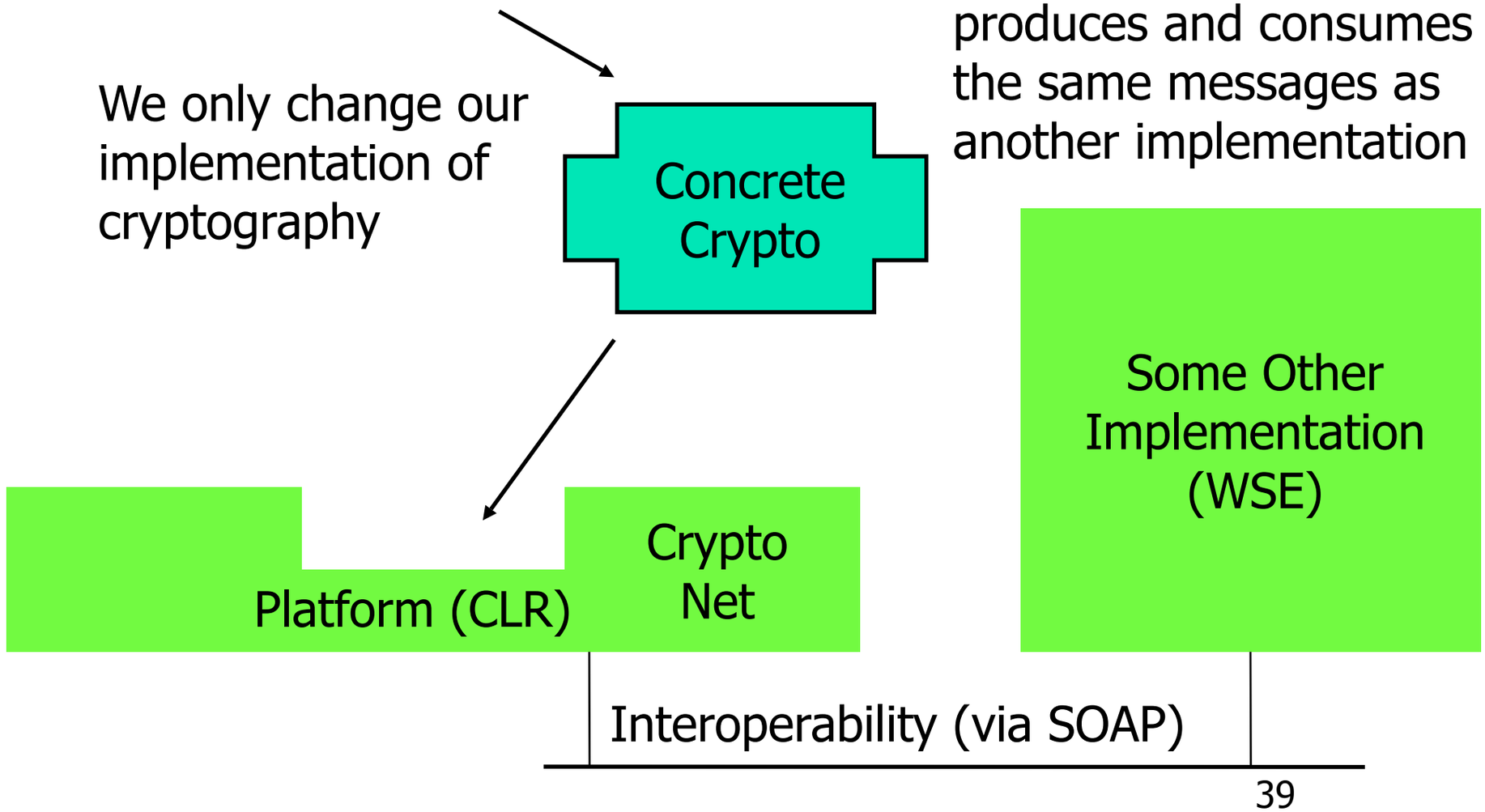
3. Concrete testing & interop

Coded in C#, F#...



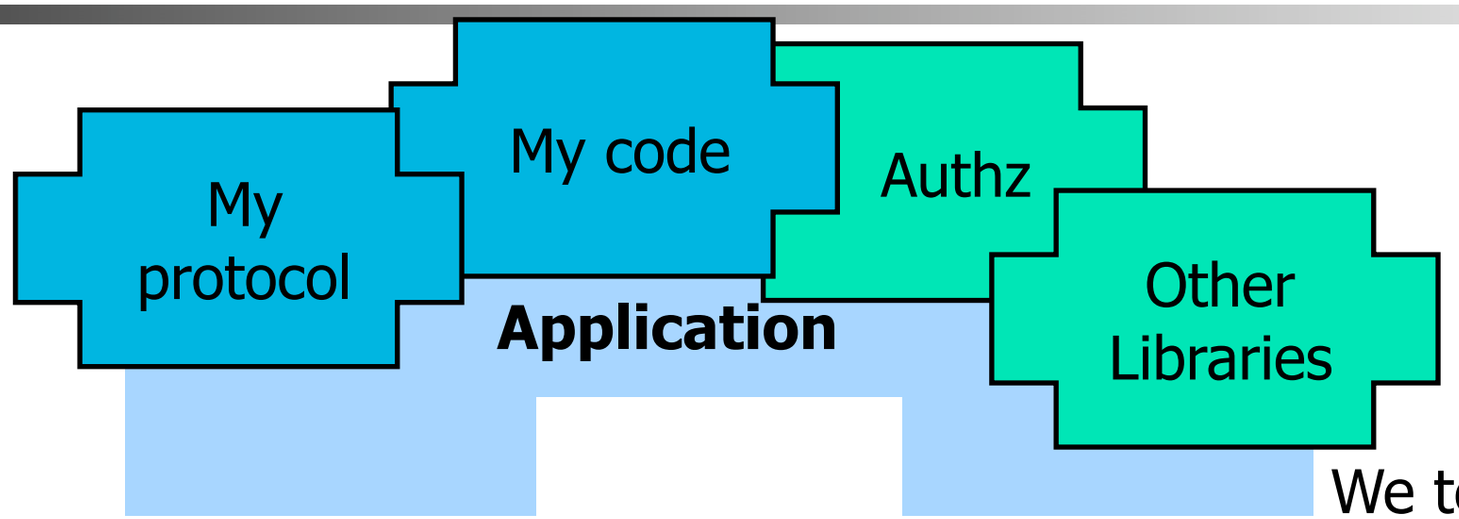
We test that our code produces and consumes the same messages as another implementation

We only change our implementation of cryptography



3. Concrete testing & interop

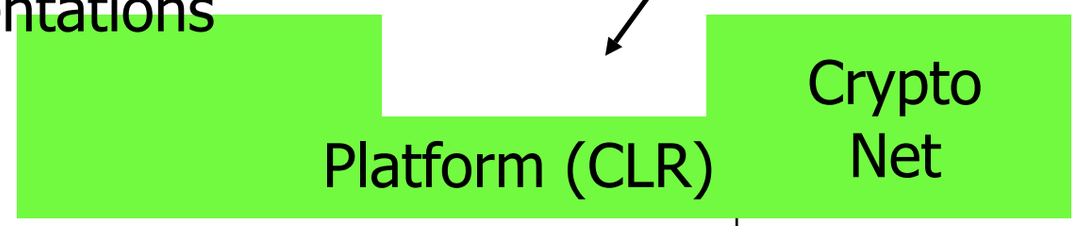
Coded in C#, F#...

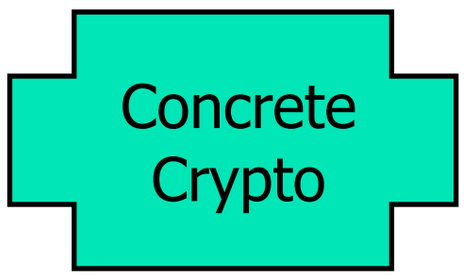


We test that our code produces and consumes the same messages as another implementation

We only change our implementation of cryptography

We can also run attacks to test other implementations





Interoperability (via SOAP)

Source language: F#

- F#, a variant of ML for the .NET runtime

  http://research.microsoft.com/fsharp

  Experimental language for research and prototyping

- Formally, a clean strongly-typed semantics
  - We support first-order functional programming
  - We rely on abstract interfaces
  - We use algebraic data types and pattern-matching for symbolic cryptography, for XML applications

Password-Based Authentication

$$A \rightarrow B: \quad \begin{array}{l} \mathrm{HMACSHA1}(\mathit{nonce}, \mathit{pwd}_A \,|\, \mathit{text}), \\ \mathrm{RSAEncrypt}(pk_B, \mathit{nonce}), \\ \mathit{text} \end{array}$$

- A simple, one-message authentication protocol
- Two roles
  - client (A) sends some text, along with a MAC
  - server (B) checks authenticity of the text
  - the MAC is keyed using a nonce and a shared password
  - the password is protected from guessing attacks by encrypting the nonce with the server's public key

Making and Checking Messages

$$A \rightarrow B: \quad \begin{array}{l} \mathrm{HMACSHA1}(\mathit{nonce}, \mathit{pwd}_A \,|\, \mathit{text}), \\ \mathrm{RSAEncrypt}(pk_B, \mathit{nonce}), \\ \mathit{text} \end{array}$$

```
let mac n pwd text = hmacsha1 n (concat (utf8 pwd) (utf8 text))

let make text pke pwd =
   let nonce = mkNonce() in
   (mac nonce pwd text, rsa_encrypt pke nonce, text)

let verify (m,en,text) skd pwd =
   let nonce = rsa_decrypt skd en in
   if not (m = mac nonce pwd text) then failwith "bad MAC"
```

Coding Client and Server roles

```
let address = S "http://server.com/pwdmac"
let pwdA = Prins.getPassword(S "A")
let pkB = Prins.getPublicKey(S "B")

type Ev = Send of str | Accept of str
```

```
// public interface
pkB: rsa_key
client: str → unit
server: unit → unit
```

```
let client text =
   log(Send(text));
   Net.send address (marshall (make text pkB pwdA))

let skB = Prins.getPrivateKey("B")
let server () =
   let m,en,text = unmarshall (Net.accept address) in
   verify (m,en,text) skB pwdA; log(Accept(text))
```

One Source, Three Tasks

- Using concrete libraries, our client and server can interact on top of a TCP socket (and could interoperate with other implementations)

  Sending FADCIzZhW3XmgUABgRJj1KjnWyDvEoAAe...

- Using symbolic libraries, we can see through cryptography

  Sending HMACSHA1{nonce3}['pwd1' | 'Hi']
  | RSAEncrypt{PK(rsa_secret2)}[nonce3] | 'Hi'

- Using symbolic libraries, fs2pv generates a ProVerif model for verification, direct from source code

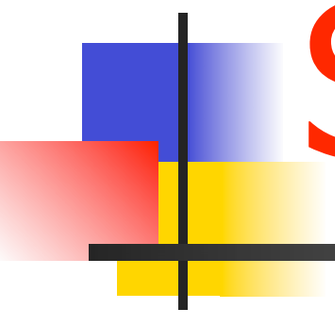
  RESULT Accept(x) ==> Send(x) is true.

Two Implementations of Crypto

```
module Crypto // concrete code in F#
open System.Security.Cryptography
type bytes = byte[]
type rsa_key = RSA of RSAParameters
...
let rng = new RNGCryptoServiceProvider ()
let mkNonce () =
  let x = Bytearray.make 16 in
  rng.GetBytes x; x
...
let hmacsha1 k x =
  new HMACSHA1(k).ComputeHash x
...
let rsa = new RSACryptoServiceProvider()
let rsa_keygen () = ...
let rsa_pub (RSA r) = ...
let rsa_encrypt (RSA r) (v:bytes) = ...
let rsa_decrypt (RSA r) (v:bytes) =
  rsa.ImportParameters(r);
  rsa.Decrypt(v,false)
```

```
module Crypto // symbolic code in F
type bytes =
  | Name of Pi.name
  | HmacSha1 of bytes * bytes
  | RsaKey of rsa_key
  | RsaEncrypt of rsa_key * bytes
  ...
and rsa_key = PK of bytes | SK of bytes
...
let freshbytes label = Name (Pi.name label)
let mkNonce () = freshbytes "nonce"
...
let hmacsha1 k x = HmacSha1(k,x)
...
let rsa_keygen () = SK (freshbytes "rsa")
let rsa_pub (SK(s)) = PK(s)
let rsa_encrypt s t = RsaEncrypt(s,t)
let rsa_decrypt (SK(s)) e = match e with
  | RsaEncrypt(pke,t) when pke = PK(s) → t
  | _ → failwith "rsa_decrypt failed"
```

Formalizing a Subset of F#



A First-Order Functional Language

x, y, z	variable
a, b	name
f	constructor (uncurried)
ℓ	function (curried)
$\mathrm{true}, \mathrm{false}, \mathrm{tuple}n, \mathrm{S}s$	primitive constructors
$\mathrm{name}, \mathrm{send}, \mathrm{recv}, \mathrm{log}, \mathrm{failwith}$	primitive functions
$M, N ::=$	value
$\quad x$	variable
$\quad a$	name
$\quad f(M_1, \ldots, M_n)$	constructor application
$e ::=$	expression
$\quad M$	value
$\quad \ell\ M_1\ \ldots\ M_n$	function application
$\quad \mathrm{fork}(\mathbf{fun}() \rightarrow e)$	fork a parallel thread
$\quad \mathbf{match}\ M\ \mathbf{with}(\mid M_i \rightarrow e_i)^{i \in 1..n}$	pattern match
$\quad \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$	sequential evaluation
$d ::=$	declaration
$\quad \mathbf{type}\ s = (\mid f_i\ \mathbf{of}\ s_{i1} * \ldots * s_{im_i})^{i \in 1..n}$	datatype declaration
$\quad \mathbf{let}\ x\ = e$	value declaration
$\quad \mathbf{let}\ \ell\ x_1 \ldots x_n = e \qquad n > 0$	function declaration
$S ::= d_1 \cdots d_n$	system: list of declarations

A Security Goal

Authentication queries are of the form **ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n.

$C \models$ **query ev**:$E \Rightarrow$ **ev**:$B_1 \vee \cdots \vee$ **ev**:B_n if and only if
whenever $C \equiv$ **event** $E\sigma \mid C'$,
there is C'' and $i \in 1..n$ such that $C' \equiv$ **event** $B_i\sigma \mid C''$.

The system S is safe for q if and only if, whenever $S \rightarrow^*_{\equiv} C$, we have $C \models q$.

We write $I \vdash S : I'$ to mean that system S
assumes an implementation of interface I,
and exports the interface I'.

We write $S :: I_{pub}$ to mean that $\mathsf{Prim} \vdash S : I_{pub}, I_{priv}$ for some I_{priv}.

An opponent O for $S :: I_{pub}$ is any system with $\mathsf{Prim}\backslash\mathsf{log}, I_{pub} \vdash O$.

$S :: I_{pub}$ is robustly safe for q when $S :: I_{pub}$ and $S\ O$ is safe for q for all opponents O.

Safety Against an Attacker

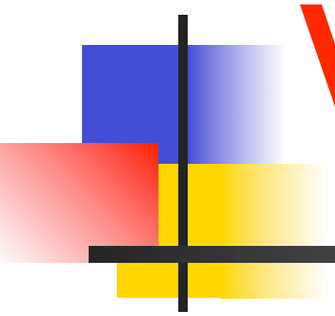
For instance, let system S be our example application code plus symbolic libraries.

Let I_{pub} be the interface

Net.send: **fun** 2, Net.accept: **fun** 1,
Crypto.S: **fun** 1, Crypto.iS: **fun** 1,
Crypto.base64: **fun** 1, Crypto.ibase64: **fun** 1,
Crypto.utf8: **fun** 1, Crypto.iutf8: **fun** 1,
Crypto.concat: **fun** 1, Crypto.iconcat: **fun** 1,
Crypto.concat3: **fun** 1, Crypto.iconcat3: **fun** 1,
Crypto.mkNonce: **fun** 1, Crypto.mkPassword: **fun** 1,
Crypto.rsa_keygen: **fun** 1, Crypto.rsa_pub: **fun** 1,
Crypto.rsa_encrypt: **fun** 2, Crypto.rsa_decrypt: **fun** 2,
Crypto.hmacsha1: **fun** 2,
pkB: **val**, client: **fun** 1, server: **fun** 1

We can verify that $S :: I_{pub}$ is robustly safe for **ev**:Accept(x) $\Rightarrow$ **ev**:Send(x)

Mapping F to a Verifiable Model



How to compile a function?

- Our compiler specifically targets symbolic verification
- We select a translation for each function
  - Complete inlining (anticipating resolution)
  - ProVerif reductions (eliminated by ProVerif)
  - ProVerif predicate declarations (logic programming)
  - ProVerif processes (most general, also most expensive)
    - We follow Milner's classic "functions as processes"
    - Each call takes two channel-based communication steps
    - We use private or public channels depending on the interface

How to compile a function?

Consider the F# function

```
let mac nonce pwd text =
  Crypto.hmacsha1 nonce (concat (utf8 pwd) (utf8 text))
```

We can translate it as a process

```
!in(mac, (nonce,pwd,text,k));
 out(k,Hmacsha1(nonce,Concat(Utf8(pwd),Utf8(text))))
```

We actually translate mac into a ProVerif reduction rule:

```
reduc mac(nonce,pwd,text) =
  HmacSha1(nonce,Concat(Utf8(pwd),Utf8(text)))
```

Interlude: Functions as Processes

Expressions of the λ-calculus:

$e ::=$	expression
x	variable
$\lambda x.e$	function (x has scope e)
$e_1\ e_2$	application
$(\lambda x.e)e' \rightarrow e\{e'/x\}$	"β-reduction"
$\mathbf{I} \triangleq \lambda x.x$	identity function
$\mathbf{K} \triangleq \lambda x.\lambda y.x$	constant function

$$\mathbf{K}\ \mathbf{I} \rightarrow \lambda y.\mathbf{I}$$
$$(\mathbf{K}\ \mathbf{I})\ \mathbf{K} \rightarrow (\lambda y.\mathbf{I})\ \mathbf{K} \rightarrow \mathbf{I}$$

Interlude: Functions as Processes

A Call-By-Value Translation from λ to π:

$$[\![x]\!]k \triangleq \textbf{out}\ k(x)$$

$$[\![\lambda x.e]\!]k \triangleq \textbf{new}\ f; (\textbf{out}\ k(f)\ |\ !\textbf{in}\ f(\langle x, k'\rangle); [\![e]\!]k')$$

$$[\![e_1\ e_2]\!]k \triangleq \textbf{new}\ k_1; ([\![e_1]\!]k_1\ |\ \textbf{in}\ k_1(f); \textbf{new}\ k_2; ([\![e_2]\!]k_2\ |\ \textbf{in}\ k_2(x); \textbf{out}\ f(\langle x, k\rangle)))$$

- Example

$$\begin{aligned} [\![\mathbf{I}]\!]k &= \textbf{new}\ f_{\mathbf{I}}; (\textbf{out}\ k(f_{\mathbf{I}})\ |\ !\textbf{in}\ f_{\mathbf{I}}(\langle x, k'\rangle); \textbf{out}\ k'(x)) \\ [\![\mathbf{K}]\!]k &= \textbf{new}\ f_{\mathbf{K}}; (\textbf{out}\ k(f_{\mathbf{K}})\ |\ !\textbf{in}\ f_{\mathbf{K}}(\langle x, k'\rangle); [\![\lambda y.x]\!]k') \\ [\![\lambda y.x]\!]k' &= \textbf{new}\ f; (\textbf{out}\ k'(f)\ |\ !\textbf{in}\ f(\langle y, k''\rangle); \textbf{out}\ k''(x)) \end{aligned}$$

- Other than this translation of functions everything else trivial

Soundness of the translation

Theorem 1 (Reflection of Robust Safety)

If $S_0 :: I_{pub}$ *and* $[\![S_0 :: I_{pub}]\!]$ *is robustly safe for* q *(in the pi calculus) then* S_0 *is robustly safe for* q *and* I_{pub} *(in F#)*

– S_0 is the series of modules that define our system;
– I_{pub} is the list of values and functions of S_0 available to the attacker;
– q is our target security query; and
– $[\![S_0 :: I_{pub}]\!]$ is the ProVerif script compiled from S_0 and I_{pub}.

To verify that S_0 is robustly safe for q and I_{pub},
1. we run ProVerif on $[\![S_0 :: I_{pub}]\!]$ with query q;
2. if ProVerif completes successfully, we apply Theorem 1.

The proof relies on an operational correspondence between reductions on F configurations and reductions in the pi calculus.

Experimental results

- We coded and verified a series of protocols
  - An implementation of a nested RPC protocol (Otway-Rees)
  - A library for Web Services Security
  - A range of web services protocols, checking interoperability
- We experimented with a range of security properties
  - Secrecy
  - Authentication
  - Correlation properties
- We coded libraries enabling various realistic attacker models
  - The attacker creates new principals, triggers their role, control the generation of cryptographic materials, and can ask for password- and key-compromise

Some Verification Results

Protocol	**Implementation**			
	LOCs	messages	bytes	symbols
password-based MAC	38	1	208	16
password-based MAC variant	75	1	238	21
Otway-Rees	148	4	74; 140; 134; 68	24; 40; 20; 11
WS password signing	85	1	3835	394
WS X.509 signing	85	1	4650	389
WS password-based MAC	85	1	6206	486
WS request-response	149	2	6206; 3187	486; 542

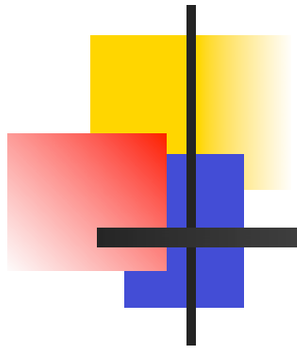
Protocol	**Security Goals**				**Verification**	
	queries	secrecy	authenticity	insiders	clauses	time
password-based MAC	4	weak pwd	msg	no	69	0.8s
password-based MAC variant	5	pwd	msg, sender	yes	213	2.2s
Otway-Rees	16	key	msg, sender	yes	155	1m50s
WS password signing	5	no	msg, sender	yes	456	5.3 s
WS X.509 signing	5	no	msg, sender	yes	460	2.6 s
WS password-based MAC	3	weak pwd	msg, sender	no	436	10.9s
WS request-response	15	no	session	yes	503	44m45s

Limits of our model

- As usual, formal security guarantees hold only within the boundaries of the model
  - We keep model and implementation in sync
  - We automatically deal with very precise models
  - We can precisely “program” the attacker model
- We verify our own implementations, not legacy code
- We trust the F# compiler and the .NET runtime
  - Certification is possible, but a separate problem
- We trust our symbolic model of cryptography
  - Partial computational soundness results may apply
  - Further verification tools may use a concrete model

Summary (part 2)

- We verify reference implementations of security protocols
- Our implementations run with both concrete and symbolic cryptographic libraries.
  - Concrete implementation for production and interop testing
  - Symbolic implementation for debugging and verification
- We develop our approach for protocols written in F#, running on the CLR, verified by ProVerif.
  - We show its correctness for a range of security properties, against realistic classes of adversaries
  - We validate our approach on WS protocols



Open Problems

- Verifying production code in a real language
  - How to verify C code for SSH or SSL or Kerberos?
- Some difficulties: discovery of loop invariants, alias analysis, discovery of heap invariants, etc.

Related Work

- From models to code (usually Java)
  - Strand spaces: Perrig, Song, Phan(2001), Lukell et al (2003)
  - CAPSL: Muller and Millen (2001)
  - Spi calculus: Lashari (2002), Pozza, Sisto, Durante (2004)
    - Newest version of spi2java supports interoperability
      - Implementation of SSH generated: Sisto, Pironti (2007)
- Giambagi and Dam (2003) show conformance between models and their implementation in terms of information flow
- Goubault-Larrecqand and Parennes (2005) analyze the secrecy of C code by first performing an alias analysis to generate Horn clauses, which they feed to FOL theorem prover (SPASS)
  - Applied to implementation of Needham-Schroeder-Lowe protocol