Progress on Provable Implementations of Security Protocols

Andrew D. Gordon
Microsoft Research

Engineering Secure Software and Systems (ESSoS’10), Pisa, Feb 3-4, 2010
Problem of Verifying Protocol Code

- The problem of vulnerabilities in security protocols is remarkably resistant to the success of formal methods
- Perhaps, tools for verifying the actual protocol code will help – Csur (VMCAI’05), fs2pv (CS’06), F7 (CSF’08), Aspier (CSF’09), etc etc
- Started around 2005 by adapting special-purpose tools to code, but trend toward customizing general-purpose program verifiers
- We have made much progress, but the routine verification of legacy code remains challenging
- Still, tools developed so far are applicable to related problems – eg crypto APIs, language-based access control, and apps outside security

- My overview goes back to an invited talk at LICS 2006...
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.”
Informal Methods

Informal lists of prudent practices attempt to discern common patterns in the extensive record of flawed protocols, and to formulate positive advice for avoiding each pattern.

(eg Abadi and Needham 1994, Anderson and Needham 1995)

**The Explicitness Principle**

Robust security is about explicitness. A cryptographic protocol should make any necessary naming, typing and freshness information explicit in its messages; designers must also be explicit about their starting assumptions and goals, as well as any algorithm properties which could be used in an attack.

Anderson and Needham *Programming Satan’s Computer* 1995

For instance, Lowe’s famous fix of the Needham-Schroeder public-key protocol makes explicit that message 6, \{|NA,B,NB|\}KA, is sent by B, who is not mentioned in the original version of the message.
Dolev & Yao first formalize N&S model in early 80s

- Symmetric key decryption: $\{ \{M\}_K \}_K^{-1} = M$
- Public key decryption: $\{|\{ | M | \}_K | \}_K^{-1} = M$
- Their work now widely recognized, but at the time, no proof techniques, so little applied

In 1987, Burrows, Abadi and Needham (BAN) propose a systematic rule-based logic for reasoning about protocols

- If P believes that he shares a key K with Q, and sees the message M encrypted under K, then he will believe that Q once said M
- If P believes that the message M is fresh, and also believes that Q once said M, then he will believe that Q believes M
- Neither sound nor complete, but useful; hugely influential
1978: N&S propose authentication protocols for “large networks of computers”
1981: Denning and Sacco find attack found on N&S symmetric key protocol
1983: Dolev and Yao first formalize secrecy properties wrt N&S threat model, using formal algebra
1987: Burrows, Abadi, Needham invent authentication logic; neither sound nor complete, but useful
1994: Hickman (Netscape) invents SSL; holes in v2, but v3 fixes these, very widely deployed
1994: Ylonen invents SSH; holes in v1, but v2 good, very widely deployed
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 2005: Many FMs now developed; several deliver both accuracy and automation
2005: Cervesato et al find same insider attack as Lowe on proposed public-key Kerberos
2006: Job Done?

After intense effort on symbolic reasoning, there are now several techniques for automatically proving properties of protocols represented within a symbolic, algebraic model

- eg Athena, TAPS, ProVerif, FDR, AVISPA, etc

Moreover, many of the unwarranted Dolev Yao abstractions (eg that message length is unobservable) are being tidied up by relating symbolic techniques to the probabilistic computational models used by cryptographers

- See proceedings of the *Formal and Computational Cryptography* workshops, for example
The Trouble...

- ...is that while practitioners are typically happy for researchers to write formal models of their natural language specifications and to apply design principles and formal tools they are reluctant to do so themselves.

- So specs tend to be partial and ambiguous.

- Implementation code is the closest we get to a formal description of most protocols.

- Hence, we need to learn from other areas of verification, and build tools to analyze code
Many formalisms for crypto protocols (including those based on process algebra and process calculi) amount to small programming languages

Several tools have successfully demonstrated the idea:

- CAPSL: Muller and Millen (2001)
- Apparently, the resulting code cannot yet interoperate with other implementations

But this amounts to growing a formal model into a full programming language, building a compiler, educating developers and so on.
Many code analysis tools can detect security issues, such as buffer overruns, but tools to extract Dolev-Yao style models from code are comparatively new.

Bhargavan, Fournet, and Gordon (CCS’04) extracted verifiable pi-calculus models from XML policies configuring some WS-Security protocols.

- This was the first time a Dolev-Yao model was extracted from the implementation files of a working system.

Goubault-Larrecq and Parrennes (VMCAI’05) were the first to build a tool to extract a formal model from the actual implementation code (in C) of a cryptographic protocol.

- Based on a pointer analysis they extract a Horn clause model suitable for analysis for other tools eg SPASS.
- The main example reported at VMCAI was one of the roles of the NSL protocol.
At MSR Cambridge and the INRIA-MSR Joint Centre we developed the Cryptographic Verification Kit, a collection of tools for analyzing security protocols implemented in F#, an Ocaml-like language for .NET

The verification tools Fs2pv and F7 are joint work with Karthik Bhargavan, INRIA, and Cédric Fournet, Microsoft Research

VERIFYING PROTOCOLS IN F#

http://research.microsoft.com/cvk
Verifying Protocol Code (not just specs)

One Source
Many Tasks

Protocol Code

Applications

Security Goals

Crypto, Net
Concrete Libraries

Crypto, Net
Symbolic Libraries

Other Implementations

Compile

Run

Compile

Run

Compile

Verifying Protocol Code

No Attack

Diverges

Proof

No Proof
F7: Latest Member of the CVK

• Currently, fs2pv most developed tool, but hitting a wall
  – Translates libraries and protocol code from F#/OCaml to ProVerif, Blanchet’s state-of-the-art verifier for crypto protocols
  – ProVerif does whole-program analysis of code versus symbolic attacker
  – Long, unpredictable run times on F# reference implementations of Cardspace (ASIACCS’08), TLS (CCS’08)

• F7 is a typechecker for F# code, enhanced with refinement types
  – Types qualified with logical formulas, eg, x:int {x>0}
  – Refinement types are general purpose, not security specific

• F7 supports a new scalable analysis for the fs2pv codebase, based on compositional and modular type-checking
Our new method:

**Invariants for Cryptographic Structures**

1. We model cryptographic structures as elements of a symbolic algebra, e.g. $MAC(k,M)$.

2. We use a “Public” predicate and events keep track of protocols.
   - $Pub(x)$ holds when the value $x$ is known to the adversary.
   - $Request(a, b, x)$ holds when $a$ intends to send message $x$ to $b$.

3. We define logical invariants on cryptographic structures.
   - $Bytes(x)$ holds when the value $x$ appears in the protocol run.
   - $KeyAB(k_{ab}, a, b)$ holds when key $k_{ab}$ is shared between $a$ and $b$.
   - After verifying the MAC (if no principals are compromised),
     $KeyAB(k_{ab}, a, b) \land Bytes(hash \ k_{ab} \ x) \implies Request(a, b, x)$.

4. We verify that the protocol code maintains these invariants (by typing)
   - $KeyAB(k_{ab}, a, b) \land Request(a, b, x)$ is a precondition for computing $hash \ k_{ab} \ x$.
AUTHENTICATED RPC

We obtain no guarantee of request/response correlation:

Client sends request1, request2 awaits replies
Service computes and sends response1, response2
Opponent swaps response1, response2
Client successfully checks MACs, and acts on the swapped responses
Informal Description

We design and implement authenticated RPCs over a TCP connection.
We have two roles, client and server, and a population of principals, $a \ b \ c \ldots$

Our security goals:

- if $b$ accepts a request $s$ from $a$,
  then $a$ has indeed sent this request to $b$;

- if $a$ accepts a response $t$ from $b$,
  then $b$ has indeed sent $t$ in response to $a$’s request.

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

There are multiple concurrent RPCs between any number of principals. The adversary controls the network. Keys and principals never compromised.
Logical Specification

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

Event predicates record the main steps of the protocol:
- \( \text{Request}(a,b,s) \) before \( a \) sends message 1;
- \( \text{Response}(a,b,s,t) \) before \( b \) sends message 2;
- \( \text{KeyAB}(k,a,b) \) before issuing a key \( k \) associated with \( a \) and \( b \).

Authentication goals are stated in terms of events:
- \( \text{RecvRequest}(a,b,s) \) after \( b \) accepts message 1;
- \( \text{RecvResponse}(a,b,s,t) \) after \( a \) accepts message 2;

where the predicates \( \text{RecvRequest} \) and \( \text{RecvResponse} \) are defined by

\[
\forall a,b,s. \ \text{RecvRequest}(a,b,s) \iff \text{Request}(a,b,s)
\]

\[
\forall a,b,s,t. \ \text{RecvResponse}(a,b,s,t) \iff (\text{Request}(a,b,s) \land \text{Response}(a,b,s,t))
\]
F# Implementation

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

Our F# implementation of the protocol:

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

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

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

The messages exchanged over TCP are:

Connecting to localhost:8080
Sending \{BgAyICsgMj9mhJa7iDACW3Rrk...\} (28 bytes)
Listening at ::1:8080
Received Request 2 + 2?
Sending \{AQA0NccjcuL/WOaYSOGGtOtPm...\} (23 bytes)
Received Response 4
Opponent as F# Program

Our security goals should hold in the face of an opponent
- able to generate arbitrary protocol instances
- able to send, receive, and do crypto on network messages.

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

**Opponent Interface (excerpts):**

```fsharp
define val send: conn → bytespub → unit
val recv: conn → bytespub
val hmacsha1: keypub → bytespub → bytespub
val hmacsha1Verify: keypub → bytespub → bytespub → unit
val setup: strpub → strpub → (strpub → unit) * (unit → unit)
```

The implementation

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

Let $I_L$ be the opponent interface for our library.
Let $I_R$ be the opponent interface for our protocol (the \textit{setup} function).

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

An expression is \textit{semantically safe} when every executed assertion logically follows from previous assumptions.

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

\textbf{Theorem 1 (Authentication for the RPC Protocol)}

\textit{For any opponent $O$, the composition $X[O]$ is semantically safe.}
F7: Refinement Types for F#

• We use extended interfaces (.fs7)
  – We typecheck implementations
  – Interfaces include types refined with first-order formulas
  – Only libraries security-specific

• We support a large subset of F#
  – Algebraic types, records, patterns, refs
  – Concurrency

\[
\begin{align*}
n &: \text{int}\{n > 0\} \\
& \quad \text{is the type of positive integers} \\
n &: \text{bytes}\{KeyAB(k, a, b)\} \\
& \quad \text{is the type of byte arrays used as keys by } a \text{ and } b \\
x &: \text{str}\{Request(a, b, x)\} \\
& \quad \text{is the type of strings sent as requests from } a \text{ to } b
\end{align*}
\]
Security by Typing

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

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

\( MKey(k) \) means \( k \) is a key for forming MACs
\( MACSays(k,b) \) is the logical payload of a MAC of bytes \( b \) with key \( k \)
\( Pub(b) \) means the opponent may know \( b \)
\( IsMAC(h,k,b) \) means \( h \) is indeed a hash of \( b \) with \( k \)

Some of the Refinement Types: MACs (Protocol)

```haskell
private val hmac_keygen: unit → k: key{MKey(k)}
val hmacsha1:
  k: key →
  b: bytes{ (MKey(k) ∧ MACSays(k,b)) } →
  h: bytes{ IsMAC(h,k,b) }
val hmacsha1Verify:
  k: key{MKey(k)} → b: bytes → h: bytes → unit{IsMAC(h,k,b)}
```
Security by Typing

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

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

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

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

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

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

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

private val hmac_keygen: unit → k:key{MKey(k)}
val hmacsha1:
    k:key →
    b:bytes{ (MKey(k) ∧ MACSays(k, b)) ∨ (Pub(k) ∧ Pub(b)) } →
    h:bytes{ IsMAC(h, k, b) ∧ (Pub(b) ⇒ Pub(h)) } 
val hmacsha1Verify:
    k:key{MKey(k) ∨ Pub(k)} → b:bytes → h:bytes → unit{IsMAC(h, k, b)}

Opponent’s abstract type bytespub $\triangleq x : \text{bytes}\{\text{Pub}(x)\}$
Some of the Formulas Assumed during Typechecking:

\((\text{KeyAB MACSays})\)
\[\forall a,b,k,m. \text{KeyAB}(k,a,b) \Rightarrow (\text{MACSays}(k,m) \iff (\exists s. \text{Request}(a,b,s) \land m = \text{request}(s)) \lor (\exists s,t. \text{Response}(a,b,s,t) \land m = \text{response}(s,t))))\]

\((\text{IsMAC MACSays})\)
\[\forall h,k,b. \text{IsMAC}(h,k,b) \land MKey(k) \Rightarrow \text{MACSays}(k,b)\]

\((\text{KeyAB MACSays})\) is a \textit{definition} for the library predicate \textit{MACSays}. It states the intended usage of keys in this protocol.

\((\text{IsMAC MACSays})\) is a \textit{theorem}: existence of a MAC with key \(k\) implies the logical payload \textit{MACSays}(k,b).

Using these assumptions, F7 typechecks our protocol code. This automatically completes our protocol verification.
What Else Is In The F7 Paper?

• (In proceedings of ACM POPL 2010)
• Theory of refined modules to justify independent type-checking of F# libraries
  – Inductive definitions expressed as Horn clauses in F7 interface files
• Comprehensive libraries for standard crypto
  – Key management, authenticated encryption, hybrid encryption, derived keys, endorsing signatures
• Example protocols: Otway-Rees key exchange, secure conversations
• Case study: Windows CardSpace
Verifying CardSpace

- We reviewed the protocol design
- We built a **modular reference implementation**
  - For the three CardSpace roles: client, relying party, identity provider
  - For the protocol stack: WS-Security standards & XML formats
  - For the underlying cryptographic primitives
- We first analyzed this code using PS2PV and ProVerif
- We now verify the same code by typing using **F7**
  - No change needed!
  - Fast, modular verification of F# code
  - We get stronger security properties, for a more precise model (reflecting all details of the XML format)
Experimental Evaluation (F7 versus fs2pv)

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

VERIFYING PROTOCOLS IN C
Security Protocol Implementation Verification

**Problem**: Establish security properties of legacy network protocol implementations

**Result**: ASPIER: Automated Security Protocol Implementation vERifier

- Checks security properties of protocol implementations in C
- Method: Builds on software model checking with a symbolic protocol and attacker model
- Tool: Method implemented in ASPIER tool
- Application: Tool used to verify authentication and secrecy properties of OpenSSL implementation of SSL handshake protocol
Overall ASPIER Framework
Case Study: OpenSSL Verification

Verified correctness of OpenSSL implementing SSL 3.0 handshake

- Checked both authentication and secrecy properties
- Client and server implementation each consists of about 1200 LOC

Confirmed the presence of the version-rollback attack in OpenSSL when clients and servers implement both SSL 2.0 and SSL 3.0

- Version number is not adequately protected in SSL 2.0
- Attacker can force client and server to use SSL 2.0 when both actually want to use SSL 3.0
Some Conclusions

- We have made much progress on verifying security protocol code, but the routine verification of legacy code remains challenging.

- ASPIER is impressive progress on C, but run-times are high, even for small numbers of clients and servers.

- F7 achieves flexible and scalable verification of security code:
  - can beat sophisticated whole-program analysis on realistic examples.
  - The only domain-specific part of our theory is the predicate library (Pub, IsMAC, etc), so it should port to general-purpose verifiers for C.

- The verification community has made substantial progress:
  - “Verified Software Initiative” aka “The War on Error”
  - In 2015, perhaps, crypto protocols will be just another routine application of general-purpose verifying compilers.

http://research.microsoft.com/cvk
QUESTIONS?