Formal Security Analysis of Cryptographic Protocol Code

Karthikeyan Bhargavan
INRIA

Karthikeyan.Bhargavan@inria.fr

IIT Delhi, Fall 2010
Lecture 4: Security Proofs for Protocol Code

Formalizing Security: Interfaces and Opponents
F Syntax

\[ M, N ::= \]
\[ x \quad \text{variable} \]
\[ a \quad \text{name} \]
\[ f(M_1, \ldots, M_n) \quad \text{constructor application} \]
\[ e ::= \]
\[ M \quad \text{value} \]
\[ \ell \ M_1 \ldots M_n \quad \text{function application} \]
\[ \text{fork}(\text{fun}() \rightarrow e) \quad \text{fork a parallel thread} \]
\[ \text{match} \ M \ \text{with} \ (| M_i \rightarrow e_i)_{i \in 1..n} \quad \text{match: } M_i \text{ patterns, } n \geq 0 \]
\[ \text{let} \ x = e_1 \ \text{in} \ e_2 \quad \text{sequential evaluation} \]
\[ d ::= \]
\[ \text{type} \ s = \ (| f_i \ \text{of} \ s_{i_1} \ast \cdots \ast s_{i_m})_{i \in 1..n} \quad \text{datatype declaration} \]
\[ \text{let} \ x = e \quad \text{value declaration} \]
\[ \text{let} \ \ell \ x_1 \ldots x_n = e \quad \text{function declaration} \quad n > 0 \]
\[ S ::= d_1 \cdots d_n \quad \text{system: list of declarations} \]
Primitives, Constructors, & Functions

- Primitive types: unit, string, bool, tuples, chan
  - F is an untyped lambda calculus, but functions and constructors have arities.
  - We write protocol code that compiles/typechecks in F#, but the opponent is an untyped F program
Interfaces

- An interface is a set of mentions
- A mention declares
  - A value variable, or
  - A constructor of arity $n$, or
  - A function of arity $n$
Prim: An Interface for F Primitives

- Constructors
  - true : `ctor 0`
  - false : `ctor 0`
  - tuplei : `ctor i` (i ∈ 1..m)
  - Ss : `ctor 0` (s ∈ Strings)

- Functions
  - failwith: `fun 1` failwith “error condition”
  - log : `fun 1` log (Send(x))
  - name : `fun 1` name ()
  - send: `fun 2` send chan msg
  - recv: `fun 1` recv chan
Formation Rules for F Interfaces

\[
\begin{align*}
I \vdash x : \text{val} & \quad I \vdash f : \text{ctor} n & \quad I \vdash M_i & \quad \forall i \in 1..n \\
I \vdash x & \quad I \vdash f(M_1, \ldots, M_n) & \\
I \vdash e & \quad I \vdash e_1, I, x : \text{val} \vdash e_2 \\
I \vdash \text{fork}(\text{fun}() \to e) & \quad I \vdash \text{let} x = e_1 \text{ in } e_2 \\
I \vdash e & \quad I \vdash M \quad I, \text{fv}(M_i) : \text{val} \vdash M_i \quad \text{fn}(M_i) = \emptyset \\
& \quad I, \text{fv}(M_i) : \text{val} \vdash e_i \quad \forall i \in 1..n \\
& \quad I \vdash \text{match } M \text{ with } (| M_i \to e_i )_{i \in 1..n} \\
I \vdash \emptyset & \\
I \vdash \emptyset : \emptyset & \\
I \vdash \text{type } s = (\{ f_i : s_i \mid i \in 1..n \} : s_{i_1} \ast \cdots \ast s_{i_n} )_{i \in 1..n} \text{ S : I } \\
I \vdash \text{let } x = e \text{ S : x : val, I'} & \\
I \vdash \text{let } \ell x_1 \ldots x_n = e \text{ S : } \ell : \text{fun } n, I' \\
I \vdash \ell : \text{fun } n & I \vdash M_i & \forall i \in 1..n \\
I \vdash \ell M_1 \ldots M_n & \\
\end{align*}
\]
Example: System Interface

\[
S_{10} \triangleq d_{Ev} d_{Cipher} d_{enc} d_{dec} d_{net} d_{key} d_{init} d_{resp} d_{u1} d_{u2}
\]

\[
d_{Ev} \triangleq \text{type } Ev = \text{Send of string} \mid \text{Accept of string}
\]

\[
d_{Cipher} \triangleq \text{type Cipher = Enc of string } \ast \text{ name}
\]

\[
d_{enc} \triangleq \text{let } enc x y = \text{Enc}(x,y)
\]

\[
d_{dec} \triangleq \text{let } dec x y = \text{match } (x,y) \text{ with } \mid (\text{Enc}(p,z),z) \rightarrow p
\]

\[
d_{net} \triangleq \text{let } net = \text{name()}
\]

\[
d_{key} \triangleq \text{let } key = \text{name()}
\]

\[
d_{init} \triangleq \text{let } \text{init } x = \text{log } (\text{Send}(x)); \text{let } c = \text{enc } x \text{ key } \text{in } \text{send net } c
\]

\[
d_{resp} \triangleq \text{let } \text{resp } () = \text{let } m = \text{recv net in let } x = \text{dec } m \text{ key } \text{in } \text{log } (\text{Accept}(x))
\]

\[
d_{u1} \triangleq \text{let } u1 = \text{fork}(\text{fun()} \rightarrow \text{init } "msg1")
\]

\[
d_{u2} \triangleq \text{let } u2 = \text{fork}(\text{fun()} \rightarrow \text{resp } ()
\)
Example: System Interface

\[ S_{10} \triangleq d_{Ev} d_{Cipher} d_{enc} d_{dec} d_{net} d_{key} d_{init} d_{resp} d_{u1} d_{u2} \]

System (Program)

\[ d_{Ev} \triangleq \text{type Ev = Send of string | Accept of string} \]
\[ d_{Cipher} \triangleq \text{type Cipher = Enc of string * name} \]
\[ d_{enc} \triangleq \text{let enc x y = Enc(x,y)} \]
\[ d_{dec} \triangleq \text{let dec x y = match (x,y) with | (Enc(p,z),z) \rightarrow p} \]
\[ d_{net} \triangleq \text{let net = name()} \]
\[ d_{key} \triangleq \text{let key = name()} \]
\[ d_{init} \triangleq \text{let init x = log (Send(x)); let c = enc x key in send net c} \]
\[ d_{resp} \triangleq \text{let resp () = let m = recv net in let x = dec m key in log (Accept(x))} \]
\[ d_{u1} \triangleq \text{let u1 = fork(fun() \rightarrow init "msg1" )} \]
\[ d_{u2} \triangleq \text{let u2 = fork(fun() \rightarrow resp())} \]

System Interface

Send: \text{ctor 1}, Accept: \text{ctor 1}, Enc: \text{ctor 2},
enc: \text{fun 2}, dec: \text{fun 2},
net: \text{val}, key: \text{val},
init: \text{fun 1}, resp: \text{fun 1},
u1: \text{val}, u2: \text{val}
Public Interfaces and Opponents

Let $S :: I_{pub}$ if and only if $\text{Prim} \vdash S : I_{pub}, I_{priv}$ for some $I_{priv}$.

• For each system we declare a public interface
  – It must be a proper subset of the system interface

Let $O$ be an $I$-opponent if and only if $\text{Prim} \setminus \log, I \vdash O : I'$ for some $I'$.

• Opponents are parameterized by public interfaces
• An $I$-opponent is an F program that can access
  the values, constructors and functions in
  $\text{Prim} \setminus \log + I$
  – An opponent must be well-formed (respect arities)
Example: System Interface

\[ S_{10} \triangleq d_{Ev} \ d_{Cipher} \ d_{enc} \ d_{dec} \ d_{net} \ d_{key} \ d_{init} \ d_{resp} \ d_{u1} \ d_{u2} \]

**System Interface**

Send: **ctor** 1, Accept: **ctor** 1, Enc: **ctor** 2,
enc: **fun** 2, dec: **fun** 2,
net: **val**, key: **val**,
init: **fun** 1, resp: **fun** 1,
u1: **val**, u2: **val**

**Public Interface**

enc: **fun** 2, dec: **fun** 2,
net: **val**,
init: **fun** 1, resp: **fun** 1,
Formalizing Security: Queries

A query $q$ is written $\text{ev}:E \Rightarrow \text{ev}:B_1 \lor \cdots \lor \text{ev}:B_n$
for values $E, B_1, \ldots, B_n$ containing no free names, with $\text{fv}(B_i) \subseteq \text{fv}(E)$ for each $i \in 1..n$.

- A query is a property of runtime configurations
  - $E$ is called the *end event*
  - $B_1, \ldots, B_n$ are called *begin events*
  - It encodes a safety property on reduction traces
  - E.g. $\text{ev}:\text{Accept}(x) \Rightarrow \text{ev}:\text{Send}(x)$
  - Whenever the event $\text{Accept}(x)$ is logged (for any $x$)
    the event $\text{Send}(x)$ must have been logged before
  - Free variables in end events (e.g. $x$) are universally quantified
Query Satisfaction

Let $\sigma$ stand for a substitution $\{M_1/x_1, \ldots, M_n/x_n\}$. Let $C \models ev:E \Rightarrow ev:B_1 \lor \cdots \lor ev:B_n$ if and only if whenever $C \equiv \text{event } E\sigma \mid C'$, we have $C' \equiv \text{event } B_i\sigma \mid C''$ for some $i \in 1..n$.

• A configuration satisfies a query iff
  – whenever the end event occurs in the configuration under some substitution $\sigma$
  – one of the begin events also occurs under $\sigma$

• E.g. if $C \equiv \text{event } \text{Accept}(x) \{"foo"/x\} \mid C'$,
  then $C' \equiv \text{event } \text{Send}(x) \{"foo"/x"\} \mid C''$
Formal Security Goal: Robust Safety

Let $C \rightarrow^* C'$ if and only if either $C \equiv C'$ or $C \rightarrow^* C'$. Let $S$ be safe for $q$ if and only if $C \models q$ whenever $S \rightarrow^* C$.

- A program is safe for a query $q$ if $q$ is satisfied by all its reachable configurations
  - Quantification over all reduction traces

Let $S$ be robustly safe for $q$ and $I$ if and only if $S :: I$ and $S O$ is safe for $q$ for all $I$-opponents $O$.

- A program is robustly safe for $q$ and public interface $I$ if it is safe against all $I$-opponents
  - Quantification over all opponents
Undecidability of Security Verification

• Robust safety quantifies over all reduction traces and all opponents
  – Is the number of traces finite?
  – No, but a number of traces look the same
e.g. (client k1 | server k1) vs (client k2 | server k2)
  – Is the number of distinct “shapes” of traces finite?
  – Only if the number of names is fixed
    • That is, if the number of sessions/keys/nonces is fixed

• In general, security verification is undecidable
  – Decidable under a variety of (unrealistic) restrictions
Example

• Let $S_{10}$ be our target program
• Let $I_{10}$ be its full system interface
• Let $I_{pub}$ be its public interface
• Let $q$ be the query $ev$:Accept($x$) => $ev$:Send($x$)

• Is $S_{10}$ safe for $q$?
• Is $S_{10}$ robustly safe for $q$ and $I_{10}$?
  – Is there an $I_{10}$-opponent $O$ s.t. $S_{10}O$ is unsafe for $q$?
• Is $S_{10}$ robustly safe for $q$ and $I_{pub}$?
Lecture 4: Security Proofs for Protocol Code

A Proof of Security
Security Goal

• Is $S_{10}$ robustly safe for $q$ and $l_{pub}$?

$$S_{10} \triangleq d_{Ev} d_{Cipher} d_{enc} d_{dec} d_{net} d_{key} d_{init} d_{resp} d_{u1} d_{u2}$$

• where $q$ is: $ev:Accept(x) \Rightarrow ev:Send(x)$

• and $l_{pub}$ is:

$$enc: \text{fun} \ 2, \ dec: \text{fun} \ 2, \ net: \text{val}, \ init: \text{fun} \ 1, \ resp: \text{fun} \ 1$$
Towards a Runtime Invariant

- We can show that $S_{10}$ always goes to $\langle d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{init}} \mid d_{\text{resp}} \mid d_{u1} \mid d_{u2} \rangle$
  
  As a shorthand, let $C_0 = \langle d_{\text{enc}} \mid d_{\text{dec}} \mid d_{\text{init}} \mid d_{\text{resp}} \rangle$.

- Let $O$ be an $I_{\text{pub}}$-opponent
- Let $C_1$ be $C_0 \mid d_{u1} \mid d_{u2} \mid O$
- Then $S_{10} O \rightarrow^* C_1$
- The program $d_{u1} \mid d_{u2} \mid O$ is also an $I_{\text{pub}}$-opponent
  - Does not use key, uses only functions and values in $I_{\text{pub}}$
- Hence, $C_1$ is of the form $C_0 \mid \text{O'}$, where $C_0$ is inactive

\[ d_{u1} \triangleq \text{let } u1 = \text{fork} \langle \text{fun}() \rightarrow \text{init} \text{ "msg1"} \rangle \]
\[ d_{u2} \triangleq \text{let } u2 = \text{fork} \langle \text{fun}() \rightarrow \text{resp} () \rangle \]
Towards a Runtime Invariant

• How can the opponent $O'$ produce event $\text{Accept}(x)$?
  – by calling $\text{resp}()$, and
  – by producing $\text{Enc}(x,k)$

• Can the opponent obtain $k$?
  – No, otherwise the protocol is broken

• How can the opponent produce $\text{Enc}(x,k)$?
  – by calling $\text{init} x$, and
  – hence, triggering $\text{Send}(x)$ before obtaining $\text{Enc}(x,k)$

• So, our runtime invariant must capture
  – The states of different calls to $\text{resp}$,
  – The occurrences of $k$
  – The states of different calls to $\text{init}$
A Runtime Invariant

For all configurations \( C \) such that \( C_1 \rightarrow^* C \)

\( C \) must be of the form

\[
C_0 \mid \textbf{event} \text{ Accept}(s1) \mid \textbf{event} \text{ Send}(s1) \\
\mid \ldots \\
\mid \textbf{event} \text{ Accept}(sk) \mid \textbf{event} \text{ Send}(sk) \\
\mid \textbf{event} \text{ Send}(t1) \mid \ldots \mid \textbf{event} \text{ Send}(tm) \\
\mid i_1 \mid \ldots \mid i_l \mid r_1 \mid \ldots \mid r_l \mid O' \text{ where} \\
\]

– All occurrences of \( k \) in \( O' \) are of the form \( \text{Enc}(ti, k) \)
– Each \( i_j \) is of the form \( \text{let } x = e_j \text{ in } O_j \) where
  \( e_j \) is an intermediate expression of init and
  all occurrences of \( k \) in \( O_j \) are of the form \( \text{Enc}(ti, k) \)
– Each \( r_j \) is of the form \( \text{let } x = e_j \text{ in } O_j \) where
  \( e_j \) is an intermediate expression of resp
Recall: Reductions of init

Let $C_1 = C_0 \mid \text{let } u_2 = \text{resp }()$.

$C_0 \mid \text{let } u_1 = \text{init }"\text{msg1}" \mid \text{let } u_2 = \text{resp }()$

$\rightarrow C_1 \mid \text{let } u_1 = (\log (\text{Send}("\text{msg1}"))); \text{let } c=\text{enc }"\text{msg1}"k \text{ in send }n\ c$

$\rightarrow C_1 \mid \text{let } u_3 = \log (\text{Send}("\text{msg1}")); \text{let } u_1=(\text{let } c=\text{enc }"\text{msg1}"k \text{ in send }n\ c$

$\rightarrow C_1 \mid \text{event }\text{Send}("\text{msg1}"); \text{let } u_1=(\text{let } c=\text{enc }"\text{msg1}"k \text{ in send }n\ c$

$\rightarrow C_1 \mid \text{event }\text{Send}("\text{msg1}"); \text{let } c=\text{enc }"\text{msg1}"k \text{ let } u_1=\text{send }n\ c$

$\rightarrow C_1 \mid \text{event }\text{Send}("\text{msg1}"); \text{let } c=\text{Enc}("\text{msg1}";k) \text{ let } u_1=\text{send }n\ c$

$\rightarrow C_1 \mid \text{event }\text{Send}("\text{msg1}"); \text{let } u_1=\text{send }n\ (\text{Enc}("\text{msg1}";k))$

$d_{\text{init}} \triangleq \text{let } \text{init } x = \log (\text{Send}(x)); \text{let } c = \text{enc } x \text{ key in send }\text{net } c$

$d_{\text{resp}} \triangleq \text{let } \text{resp }() = \text{let } m = \text{recv }\text{net in let } x = \text{dec } m \text{ key in log } (\text{Accept}(x))$
Recall: Reductions of init

As a further abbreviation, let $C_2 = C_0 \mid \text{event Send("msg1")} \mid$

$$\text{let } u_1 = \text{send n (Enc("msg1", k))}.$$  

$C_1 \mid \text{event Send("msg1")} \mid \text{let } u_1 = \text{send n (Enc("msg1", k))}$

$$= C_2 \mid \text{let } u_2 = \text{resp ()}$$

$$\rightarrow C_2 \mid \text{let } u_2 = (\text{let } m = \text{recv n in let } x = \text{dec m k in log (Accept(x))})$$

$$\rightarrow C_2 \mid \text{let } m = \text{recv n let } u_2 = (\text{let } x = \text{dec m k in log (Accept(x))})$$
Recall: Reductions of recv

Let $C_3 = C_0 \mid \text{event } \text{Send}("\text{msg1}").$

$C_2 \mid \text{let } m = \text{recv } n \text{ let } u_2 = (\text{let } x = \text{dec } m \text{ k in } \text{log } (\text{Accept}(x)))$

$= C_3 \mid \text{let } u_2 = (\text{let } x = \text{dec } (\text{Enc}("\text{msg1}",k)) \text{ k in } \text{log } (\text{Accept}(x)))$

$\rightarrow C_3 \mid \text{let } x = \text{dec } (\text{Enc}("\text{msg1}",k)) \text{ k let } u_2 = \text{log } (\text{Accept}(x)))$

$\rightarrow C_3 \mid \text{let } x = \text{match } (\text{Enc}("\text{msg1}",k),k) \text{ with } (\text{Enc}(p,z),z) \rightarrow p$

$\quad \text{let } u_2 = \text{log } (\text{Accept}(x)))$

$\rightarrow C_3 \mid \text{let } x = "\text{msg1}" \text{let } u_2 = \text{log } (\text{Accept}(x)))$

$\rightarrow C_3 \mid \text{let } u_2 = \text{log } (\text{Accept}("\text{msg1}")))$

$\rightarrow C_3 \mid \text{event } \text{Accept}("\text{msg1}"))$
Proof of Invariant

• By induction on the length of $C_1 \rightarrow^* C$

*Base Case* (length 0): $C_1$ obeys the invariant

*Induction Hypothesis:* After $n$ steps,

$$C = C_0 \mid \text{event } \text{Accept}(s1) \mid \text{event } \text{Send}(s1) \mid \ldots$$

Let $C \rightarrow C'$

We proceed by case analysis on this reduction.

1. If the reduction is in one of the $i_j$
   - then it either stays an $i_j$ or produces a $\text{Send}(x)$ and an $\text{Enc}(x,k)$

2. If the reduction is in one of the $r_j$
   - then it either stays an $r_j$ or takes an $\text{Enc}(x,k)$ and produces $\text{Accept}(x)$

3. If the reduction is in $O$
   - then it either stays an $O$ or produces an $i_j$ or an $r_j$
Proof of Robust Safety

• Any configuration that satisfies the invariant satisfies our target query
• Every reachable configuration of our program with any opponent satisfies the invariant
• Hence, $S_{10}$ is robustly safe
Lecture 4:
Security Proofs for Protocol Code

Proof Techniques
Proof Techniques for Security

• We have used an invariant-based technique to verify the security of our program

• Particularly suited to manual proof, but not easily automatable
  – Figuring out the invariant is the difficult bit

• Developing automated proof techniques for security protocols is an active area of research
  – At least 3 international conferences
  – Intersection of Programming Languages, Security, Cryptography, and Formal Methods
Early Attempts

- Dolev & Yao first formalize N&S problem in early 80s
  - Shared key decryption: $\{\{M\}K\}K^{-1} = M$
  - Public key decryption: $\{|\{|M|\}KA\}KA^{-1} = M$
  - Their work now widely recognised, but at the time, few 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
  - Incomplete, but useful; hugely influential
A Potted History: 1978-2005

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.

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; incomplete, 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 FM

circa 2000: Several FM for “D&Y problem”: tradeoff between accuracy and approximation
2000: Abadi and Rogaway initiate connections between formal and computational models of crypto

circa 2005: Many FM now developed; several (eg ProVerif) deliver both accuracy and automation
2005: Cervesato et al find same insider attack as Lowe on proposed public-key Kerberos
2005: Goubault-Larrecq and Parrennes pioneer direct verification of implementation code in C
The Inductive Method [Paulson]

- Formalize the protocol from the viewpoint of the attacker, using Higher-order Logic
- Attacker’s knowledge represented by a set $K$
  - If $k \in K$ and $x \in K$, then $\text{Enc}(k,x) \in K$
  - If $k \in K$ and $\text{Enc}(k,x) \in K$, then $x \in K$
  - If $\text{Enc}(k,x) \in K$, then $\text{Accept}(x)$
  - $\forall x. \text{Send}(x)$
  - $\forall x. \text{Enc}(k,x) \in K$
  - $N \in K$, $M \in K$
- Use a theorem-prover (Coq or Isabelle/HOL) to prove robust safety by rule induction
Resolution with First-order Unification [Blanchet]

- Formalize the protocol from the viewpoint of the attacker, using First-order Horn Clauses
- Attacker’s knowledge represented by predicate att and a set of rewrite rules
  - att:k and att:x => att:Enc(k,x)
  - att:Enc(k,x) => ev:Accept(x)
  - ev:Send(x)
- Use resolution to compute the completion of these rewrite rules
- Upon completion, ask query of the set of generated rules (like a Prolog program)
Model Checking [Roscoe, Basin]

- Formalize the protocol as a communicating finite state machine (CSP/CCS/CFSM)
- Attacker is also a finite state process with specific capabilities
- Limit number of sessions/names/nonces
- Find all reduction traces and verify query on each reachable state
- Violation of query => attack
  - No violation found does not mean proof
Verifying Programs not Models

• Most previous methods focus on models
• What about code?
  – Csur, Aspier: C model checking
  – FS2PV: F# to ProVerif
  – F7: Security Typechecking

• Active research groups
  – CertiCrypt (France)
  – Aspier (CMU)
  – F7 (France)