形式化方法导引

第 6 章 案例分析

6.1 ProVerif: A Verifier of Security Protocols

黄文超

https://faculty.ustc.edu.cn/huangwenchao

$\longrightarrow$ 教学课程 $\longrightarrow$ 形式化方法导引

Outline

案例分析:

1. *ProVerif*: A *Verifier* of Security Protocols
2. *Software Analysis*: Abstract Interpretation and CEGAR
3. *Coq Proof Assistant*: A *Prover* based on Higher-order Logic

回顾: 程序验证方法的分类:

- **Verification (验证)**: 若程序设计正确，则给出正确性证明；若命题错误，则给出“错误”的判断，并给出反例
- **Proof (证明)**: 若程序设计正确，则给出正确性证明
- **Falsification**: 找出程序设计的错误，给出反例。一般方法包括测试、检测

ProVerif is an *automatic* cryptographic protocol *verifier*

Related Papers (*Introduction* to ProVerif):

- A survey of ProVerif
- The most cited one (1288 times)

Related Papers (*Application* of ProVerif):

- Verifying TLS 1.3 (S&P'17)
- Secure File Sharing (S&P'08)
- Just Fast Keying (TISSEC'07)

Keywords:

- Assumption of Adversary
  - *Symbolic model*, i.e., Dolev-Yao model
  - ps: another model - computational model
- *Modeling*
  - *Pi-calculus*
    - applied pi calculus
- *Specification*
  - Trace properties
  - Equivalence properties
- *Algorithm*
  - *Horn* clauses *Abstraction*
  - Resolution Horn Clauses by *Unification*

Why introducing these Keywords?

Keywords:

- Assumption of Adversary
  - *Symbolic model*, i.e., Dolev-Yao model
  - ps: another model - computational model
- *Modeling*
  - *Pi-calculus*
    - applied pi calculus
- *Specification*
  - Trace properties
  - Equivalence properties
- *Algorithm*
  - *Horn* clauses *Abstraction*
  - Resolution Horn Clauses by *Unification*

Why introducing these Keywords?

Assumption of Adversary

- *Symbolic model*, i.e., **Dolev-Yao model**
  - Cryptographic primitives are considered as perfect *blackboxes*
  - *Why?: Do not consider* the case of *cryptographic attacks*
    - e.g., brute-force attacks on encryption keys
- Another model - computational model
  - messages are bitstrings
  - cryptographic primitives are functions from bitstrings to bitstrings
  - the adversary is any probabilistic Turing machine
  - This is the model usually considered by *cryptographers*.

Assumption of Adversary

- *Symbolic model*, i.e., **Dolev-Yao model**
  - Cryptographic primitives are considered as perfect *blackboxes*
  - *Why?: Do not consider* the case of *cryptographic attacks*
    - e.g., brute-force attacks on encryption keys
- Another model - computational model
  - messages are bitstrings
  - cryptographic primitives are functions from bitstrings to bitstrings
  - the adversary is any probabilistic Turing machine
  - This is the model usually considered by *cryptographers*.

Assumption of Adversary

- *Symbolic model*, i.e., **Dolev-Yao model**
  - Cryptographic primitives are considered as perfect *blackboxes*
  - *Why?: Do not consider* the case of *cryptographic attacks*
    - e.g., brute-force attacks on encryption keys
- Another model - computational model
  - messages are bitstrings
  - cryptographic primitives are functions from bitstrings to bitstrings
  - the adversary is any probabilistic Turing machine
  - This is the model usually considered by *cryptographers*.

Modeling

- *Pi-calculus*
  - applied pi calculus

Why?: *Easier* for modeling

- Compared with *transition system* in model checking

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

Modeling

- *Pi-calculus*
  - applied pi calculus

Why?: Easier for modeling

- Compared with *transition system* in model checking

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

Specification

- Trace properties
- Equivalence properties

Why?: *Easier* for designing specifications

- Compared with *CTL*, *LTL* in model checking

Specification

- Trace properties
- Equivalence properties

Why?: Easier for designing specifications

- Compared with *CTL, LTL* in model checking

Algorithm

- *Horn* clauses *Abstraction*
- Resolution of Horn Clauses by *Unification*

Why?:

- Avoid state explosion
- Abstraction reduces the complexity

Algorithm

- *Horn* clauses *Abstraction*
- Resolution of Horn Clauses by *Unification*

Why?:

- Avoid state explosion
- Abstraction reduces the complexity

回顾: Keywords:

- Assumption of Adversary
  - *Symbolic model*, i.e., Dolev-Yao model
  - ps: another model - computational model
- *Modeling*
  - *Pi-calculus*
    - applied pi calculus
- *Specification*
  - Trace properties
  - Equivalence properties
- *Algorithm*
  - *Horn* clauses *Abstraction*
  - Resolution Horn Clauses by *Unification*

下一个问题: *How to study ProVerif in detail?*

回顾: Keywords:

- Assumption of Adversary
  - *Symbolic model*, i.e., Dolev-Yao model
  - ps: another model - computational model
- *Modeling*
  - *Pi-calculus*
    - applied pi calculus
- *Specification*
  - Trace properties
  - Equivalence properties
- *Algorithm*
  - *Horn* clauses *Abstraction*
  - Resolution Horn Clauses by *Unification*

下一个问题: *How to study ProVerif in detail?*

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

基本知识–密码的种类:

- (1) **对称密钥** (传统密码) (2) **非对称密钥** (公钥密码)

传统密码	公钥密码
加密和解密使用**相同的密钥**和**相同的算法**	加密和解密使用**不同的密钥**和**相同的算法**
收发双方**共享密钥**	双方使用的密钥**不同**
密钥必须保密	私钥必须保密，公钥不需要保密
若没有其它信息，则解密消息**不可能或不可行**	若没有其它信息，则解密消息**不可能或不可行**
已知算法和若干密文，**不足以确定密钥**	已知算法、若干密文、**其中一个密钥**，**不足以确定另一个密钥**

传统密码:

对称加密-解密过程:

- 对称密钥: k
- 明文消息: m
- 加密: c=senc(m, k)
- 解密: m=sdec(c, k)

公钥密码:

非对称加密-解密过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 明文消息: m
- 加密: c=aenc$(m, pk(sk))$
- 解密: m=adec(c, sk)

签名-验证过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 待签名消息: m
- 签名: c=sign(m, sk)
- 验证: verify $m \equiv$ check$(c, pk(sk))$

传统密码:

对称加密-解密过程:

- 对称密钥: k
- 明文消息: m
- 加密: c=senc(m, k)
- 解密: m=sdec(c, k)

公钥密码:

非对称加密-解密过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 明文消息: m
- 加密: c=aenc$(m, pk(sk))$
- 解密: m=adec(c, sk)

签名-验证过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 待签名消息: m
- 签名: c=sign(m, sk)
- 验证: verify $m \equiv$ check$(c, pk(sk))$

传统密码:

对称加密-解密过程:

- 对称密钥: k
- 明文消息: m
- 加密: c=senc(m, k)
- 解密: m=sdec(c, k)

公钥密码:

非对称加密-解密过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 明文消息: m
- 加密: c=aenc$(m, pk(sk))$
- 解密: m=adec(c, sk)

签名-验证过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 待签名消息: m
- 签名: c=sign(m, sk)
- 验证: verify $m \equiv$ check$(c, pk(sk))$

传统密码:

对称加密-解密过程:

- 对称密钥: k
- 明文消息: m
- 加密: c=senc(m, k)
- 解密: m=sdec(c, k)

公钥密码:

非对称加密-解密过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 明文消息: m
- 加密: c=aenc$(m, pk(sk))$
- 解密: m=adec(c, sk)

签名-验证过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 待签名消息: m
- 签名: c=sign(m, sk)
- 验证: verify $m \equiv$ check$(c, pk(sk))$

传统密码:

对称加密-解密过程:

- 对称密钥: k
- 明文消息: m
- 加密: c=senc(m, k)
- 解密: m=sdec(c, k)

公钥密码:

非对称加密-解密过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 明文消息: m
- 加密: c=aenc$(m, pk(sk))$
- 解密: m=adec(c, sk)

签名-验证过程:

- 非对称密钥: 公钥-pk, 私钥-sk
- 待签名消息: m
- 签名: c=sign(m, sk)
- 验证: verify $m \equiv$ check$(c, pk(sk))$

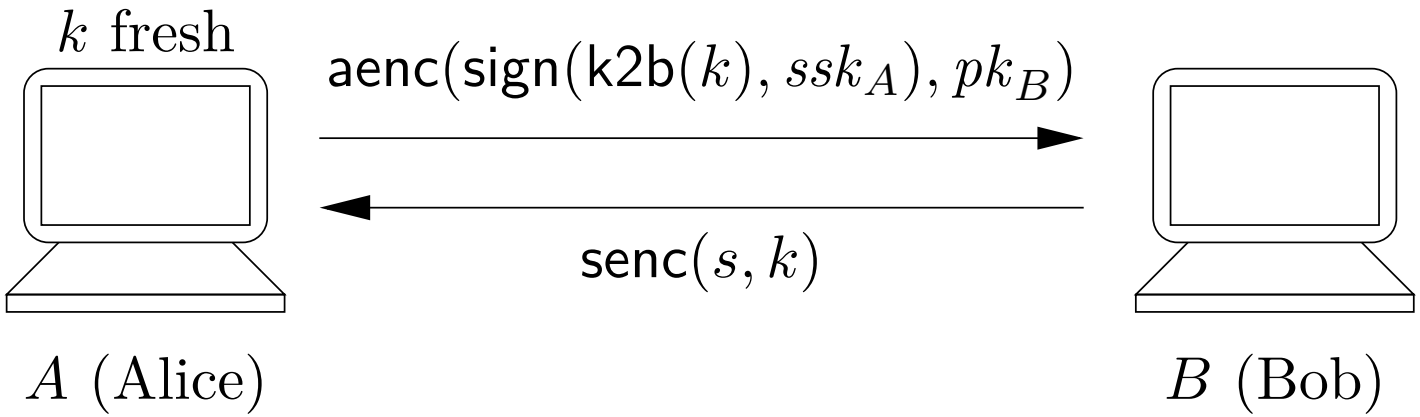
1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

(1) Model: A simple example of a security protocol

- Denning-Sacco key distribution protocol (Denning and Sacco, 1981)



1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

(2) Assumption: Introducing the Dolev-Yao model

- The network is represented by a set of *abstract* machines that can *exchange messages*.
- The *adversary* can *overhear*, *intercept*, and *synthesize* any message
- The *adversary* is only *limited* by the constraints of the *cryptographic* methods used

(2) Assumption: Introducing the Dolev-Yao model

- The network is represented by a set of *abstract* machines that can *exchange messages*.
- The *adversary* can *overhear*, *intercept*, and *synthesize* any message
- The *adversary* is only *limited* by the constraints of the *cryptographic* methods used

(2) Assumption: Introducing the Dolev-Yao model

- The network is represented by a set of *abstract* machines that can *exchange messages*.
- The *adversary* can *overhear*, *intercept*, and *synthesize* any message
- The *adversary* is only *limited* by the constraints of the *cryptographic* methods used

(2) Assumption: Introducing the Dolev-Yao model

- The network is represented by a set of *abstract* machines that can *exchange messages*.
- The *adversary* can *overhear*, *intercept*, and *synthesize* any message
- The *adversary* is only *limited* by the constraints of the *cryptographic* methods used

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

(3) Specification: Specifying the properties

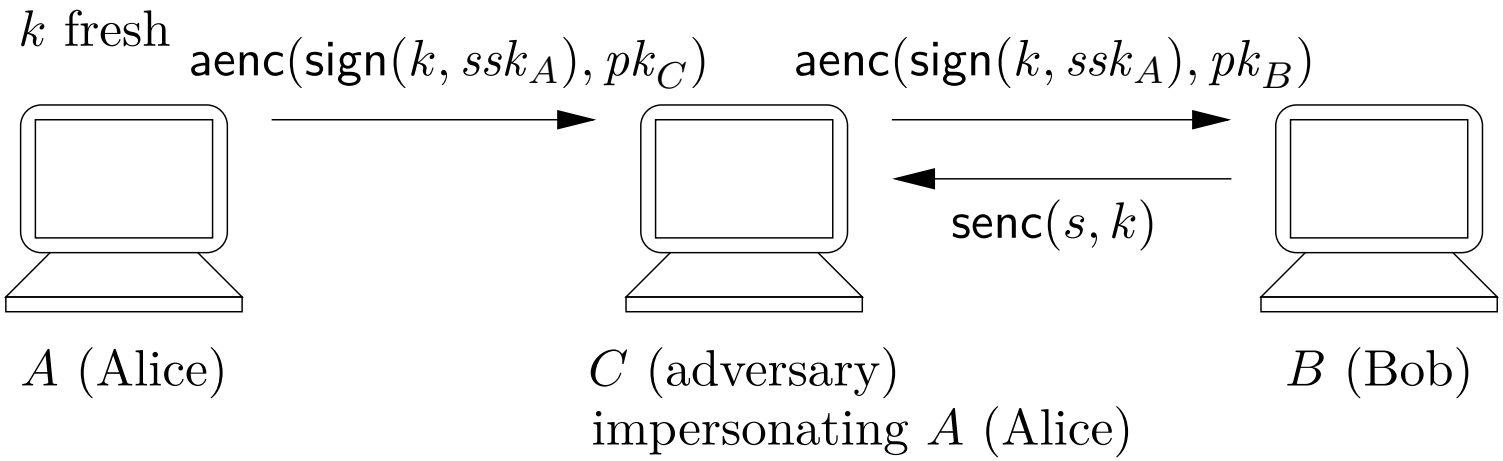
- *Only* A should be able to decrypt the message and *get the secret* s

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

(4) Manual Analysis: discovered attacks



1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application
$D ::=$	expressions
M	term
$h(D_1, \ldots, D_n)$	function application
fail	failure
$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application

- *Names*: represent *atomic data*, such as keys and nonces
- *Variables*: can be substituted by terms.
- *Constructors*: are used to build terms, e.g.,
  - $\mathrm{senc}(c, k)$ represents the encryption of c under the key k.
- *Destructors*:
  - *do not appear in terms*, but *manipulate terms* in expressions （见下页）

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application

- *Names*: represent *atomic data*, such as keys and nonces
- *Variables*: can be substituted by terms.
- *Constructors*: are used to build terms, e.g.,
  - $\mathrm{senc}(c, k)$ represents the encryption of c under the key k.
- *Destructors*:
  - *do not appear in terms*, but *manipulate terms* in expressions （见下页）

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application

- *Names*: represent *atomic data*, such as keys and nonces
- *Variables*: can be substituted by terms.
- *Constructors*: are used to build terms, e.g.,
  - $\mathsf{senc}(c, k)$ represents the encryption of c under the key k.
- *Destructors*:
  - *do not appear in terms*, but *manipulate terms* in expressions （见下页）

$M, N ::=$	terms
x, y, z	variable
a, b, c, k, s	name
$f(M_1, \ldots, M_n)$	constructor application

- *Names*: represent *atomic data*, such as keys and nonces
- *Variables*: can be substituted by terms.
- *Constructors*: are used to build terms, e.g.,
  - $\mathsf{senc}(c, k)$ represents the encryption of c under the key k.
- *Destructors*:
  - *do not appear in terms*, but *manipulate terms* in expressions （见下页）

$D ::=$	expressions
M	term
$h(D_1, \ldots, D_n)$	function application
fail	failure

Destructors:

- *do not appear in terms*, but *manipulate terms* in expressions
- They are *functions on terms* that *processes* can apply, *via* the expression evaluation *construct*, i.e.,
  - let $x : T = D$ in P else Q
- A destructor g is defined by a finite ordered list of rewrite rules $\mathrm{def}(g)$ of the form
  - $g(U_1, ..., U_n) \to U$ where $U_1, ..., U_n$, U are may-*fail terms*

Using *constructors* and *destructors*, we can represent data structures, such as

- tuples:
  - constructor: $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$, where term M_i is of type T_i, and $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$ returns a result of type bitstring.
  - destructor: $i\text{th}_{T_1,...,T_n}(\text{tuple}_{T_1,...,Tn}(x_1, ..., x_n)) \rightarrow x_i$
- cryptographic operations
  - constructor: senc(bitstring,key): bitstring
  - destructor: $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$
  - constructor: pk(skey):pkey, aenc(bitstring,pkey):bitstring
  - destructor: $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$
  - constructor: pk(skey):pkey, sign(bitstring,skey):bitstring
  - destructor: $\text{getmess}(\text{sign}(x, y)) \rightarrow x$
- Type converter
  - constructor: b2k(bitstring) : key
  - destructor: $\text{b2k}(\text{k2b}(x)) \rightarrow x$

Using *constructors* and *destructors*, we can represent data structures, such as

- tuples:
  - constructor: $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$, where term M_i is of type T_i, and $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$ returns a result of type bitstring.
  - destructor: $i\text{th}_{T_1,...,T_n}(\text{tuple}_{T_1,...,Tn}(x_1, ..., x_n)) \rightarrow x_i$
- cryptographic operations
  - constructor: senc(bitstring,key): bitstring
  - destructor: $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$
  - constructor: pk(skey):pkey, aenc(bitstring,pkey):bitstring
  - destructor: $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$
  - constructor: pk(skey):pkey, sign(bitstring,skey):bitstring
  - destructor: $\text{getmess}(\text{sign}(x, y)) \rightarrow x$
- Type converter
  - constructor: b2k(bitstring) : key
  - destructor: $\text{b2k}(\text{k2b}(x)) \rightarrow x$

Using *constructors* and *destructors*, we can represent data structures, such as

- tuples:
  - constructor: $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$, where term M_i is of type T_i, and $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$ returns a result of type bitstring.
  - destructor: $i\text{th}_{T_1,...,T_n}(\text{tuple}_{T_1,...,Tn}(x_1, ..., x_n)) \rightarrow x_i$
- cryptographic operations
  - constructor: senc(bitstring,key): bitstring
  - destructor: $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$
  - constructor: pk(skey):pkey, aenc(bitstring,pkey):bitstring
  - destructor: $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$
  - constructor: pk(skey):pkey, sign(bitstring,skey):bitstring
  - destructor: $\text{getmess}(\text{sign}(x, y)) \rightarrow x$
- Type converter
  - constructor: b2k(bitstring) : key
  - destructor: $\text{b2k}(\text{k2b}(x)) \rightarrow x$

Using *constructors* and *destructors*, we can represent data structures, such as

- tuples:
  - constructor: $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$, where term M_i is of type T_i, and $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$ returns a result of type bitstring.
  - destructor: $i\text{th}_{T_1,...,T_n}(\text{tuple}_{T_1,...,Tn}(x_1, ..., x_n)) \rightarrow x_i$
- cryptographic operations
  - constructor: senc(bitstring,key): bitstring
  - destructor: $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$
  - constructor: pk(skey):pkey, aenc(bitstring,pkey):bitstring
  - destructor: $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$
  - constructor: pk(skey):pkey, sign(bitstring,skey):bitstring
  - destructor: $\text{getmess}(\text{sign}(x, y)) \rightarrow x$
- Type converter
  - constructor: b2k(bitstring) : key
  - destructor: $\text{b2k}(\text{k2b}(x)) \rightarrow x$

Using *constructors* and *destructors*, we can represent data structures, such as

- tuples:
  - constructor: $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$, where term M_i is of type T_i, and $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$ returns a result of type bitstring.
  - destructor: $i\text{th}_{T_1,...,T_n}(\text{tuple}_{T_1,...,Tn}(x_1, ..., x_n)) \rightarrow x_i$
- cryptographic operations
  - constructor: senc(bitstring,key): bitstring
  - destructor: $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$
  - constructor: pk(skey):pkey, aenc(bitstring,pkey):bitstring
  - destructor: $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$
  - constructor: pk(skey):pkey, sign(bitstring,skey):bitstring
  - destructor: $\text{getmess}(\text{sign}(x, y)) \rightarrow x$
- Type converter
  - constructor: b2k(bitstring) : key
  - destructor: $\text{b2k}(\text{k2b}(x)) \rightarrow x$

Using *constructors* and *destructors*, we can represent data structures, such as

- tuples:
  - constructor: $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$, where term M_i is of type T_i, and $\text{tuple}_{T_1,...,T_n}(M_1, ..., M_n)$ returns a result of type bitstring.
  - destructor: $i\text{th}_{T_1,...,T_n}(\text{tuple}_{T_1,...,Tn}(x_1, ..., x_n)) \rightarrow x_i$
- cryptographic operations
  - constructor: senc(bitstring,key): bitstring
  - destructor: $\text{sdec}(\text{senc}(x, y), y) \rightarrow x$
  - constructor: pk(skey):pkey, aenc(bitstring,pkey):bitstring
  - destructor: $\text{adec}(\text{aenc}(x, \text{pk}(y)), y) \rightarrow x$
  - constructor: pk(skey):pkey, sign(bitstring,skey):bitstring
  - destructor: $\text{getmess}(\text{sign}(x, y)) \rightarrow x$
- Type converter
  - constructor: b2k(bitstring) : key
  - destructor: $\text{b2k}(\text{k2b}(x)) \rightarrow x$

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The nil process 0 does nothing.

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The output process **out**(N, M);P outputs the message M on the channel N and then executes P.

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The input process **in**$(N, x : T)$;P inputs a message on channel N, and executes P with x bound to the input message.

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The process $P \mid Q$ is the parallel composition of P and Q.

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The replication $!P$ represents an unbounded number of copies of P in parallel.

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The restriction **new** $a : T; P$ creates a new name a of type T, and then executes P.

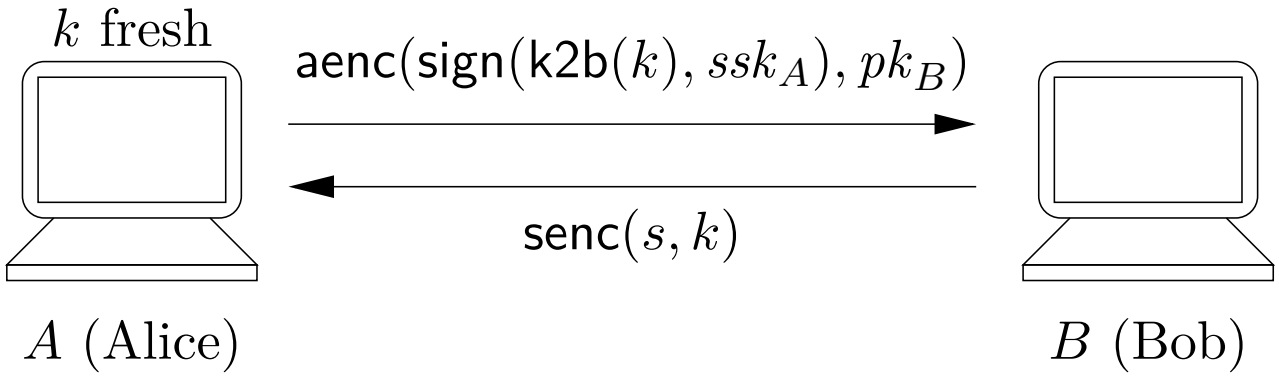
$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The process **let** $x : T = D$ **in** P **else** Q tries to evaluate D;

- if D evaluates to a term M, then x is bound to M and P is executed
- if the evaluation of D fails, then Q is executed

$P, Q ::=$	processes
$\mathbf{0}$	nil
$\mathsf{out}(N, M); P$	output
$\mathsf{in}(N, x : T); P$	input
$P \mid Q$	parallel composition
$!P$	replication
$\mathsf{new}\ a : T; P$	restriction
$\mathsf{let}\ x : T = D\ \mathsf{in}\ P\ \mathsf{else}\ Q$	expression evaluation
$\mathsf{if}\ M\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

The conditional **if** M **then** P **else** Q executes P if M is true (or is a variable bound to true); it executes Q if M is different from true.



$P_0 =$ new ssk_A : skey; new sk_B : skey; let $spk_A =$ pk(ssk_A) in
let $pk_B =$ pk(sk_B) in out(c, spk_A); out(c, pk_B);
($P_A(ssk_A, pk_B) \mid P_B(sk_B, spk_A)$)

$P_A(ssk_A, pk_B) =$! new k : key;
out(c, aenc(sign(k2b(k), ssk_A), pk_B));
in(c, x : bitstring); let $z =$ sdec(x, k) in $\mathbf{0}$

$P_B(sk_B, spk_A) =$! in(c, y : bitstring); let $y' =$ adec(y, sk_B) in
let $x_k =$ b2k(check(y', spk_A)) in out(c, senc(s, x_k))

问题 1: This model of the protocol is *weak*, because

- A and B talk only to each other
- they do not interact with other, possibly *dishonest participants*

We can strengthen model by *replacing* the process P_A with the following process:

$$P_A(ssk_A, pk_B) = !\,\mathsf{in}(c, x_{pk_B} : \mathsf{pkey}); \mathsf{new}\ k : \mathsf{key};$$
$$\mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), ssk_A), x_{pk_B}));$$
$$\mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0}$$

问题 2: The above model still assumes for *simplicity* that A and B each play *only one role* of the protocol.

- One could easily write an even more general model in which they play both roles (改进过程: 略)

问题 1: This model of the protocol is *weak*, because

- A and B talk only to each other
- they do not interact with other, possibly *dishonest participants*

We can strengthen model by *replacing* the process P_A with the following process:

$$P_A(ssk_A, pk_B) = !\,\mathsf{in}(c, x_{pk_B} : \mathsf{pkey}); \mathsf{new}\ k : \mathsf{key};$$
$$\mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), ssk_A), x_{pk_B}));$$
$$\mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0}$$

问题 2: The above model still assumes for *simplicity* that A and B each play *only one role* of the protocol.

- One could easily write an even more general model in which they play both roles (**改进过程**: 略)

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

Specifying the properties:

query not $attacker(\mathsf{s})$.

What if we model the property in CTL?

$$\mathcal{M}, s_0 \models AG\ \neg \mathrm{attacker}(s)$$

1. ProVerif

Outline

Now coding:(1st version)

free c: channel.

type key.
type pkey.
type skey.
type spkey.
type sskey.

fun k2b(key):bitstring $[data, typeConverter]$.
reduc forall k:key; b2k(k2b(k)) = k.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.
reduc forall x: bitstring, y: skey; adec(aenc(x, pk(y)), y) = x.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.
reduc forall m: bitstring, k: sskey; checksign(sign(m, k), spk(k)) = m.

fun senc(bitstring, key): bitstring.
reduc forall x: bitstring, y: key; sdec(senc(x, y), y) = x.

//Specification
free s: bitstring [*private*].
query *attacker*(s).

//PA will be revised in 2nd version

$$\begin{aligned}
&\textbf{let } PA(sskA\text{: sskey}, pkB\text{:pkey}) = \\
&\qquad \textbf{new } k\text{:key}; \\
&\qquad \textbf{out}(\mathsf{c}, \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), sskA), pkB)); \\
&\qquad \textbf{in}(\mathsf{c}, x\text{:bitstring}); \\
&\qquad \textbf{let } z = \mathsf{sdec}(x, k) \textbf{ in } 0.
\end{aligned}$$

$$\begin{aligned}
&\textbf{let } PB(skB\text{:skey}, spkA\text{:spkey}) = \\
&\qquad \textbf{in}(\mathsf{c}, y\text{:bitstring}); \\
&\qquad \textbf{let } y1 = \mathsf{adec}(y, skB) \textbf{ in} \\
&\qquad \textbf{let } xk = \mathsf{b2k}(\mathsf{checksign}(y1, spkA)) \textbf{ in} \\
&\qquad \textbf{out}(\mathsf{c}, \mathsf{senc}(\mathsf{s}, xk)).
\end{aligned}$$

process

new $sskA$: sskey;
new skB: skey;
let $spkA = \mathsf{spk}(sskA)$ **in**
let $pkB = \mathsf{pk}(skB)$ **in**
$\mathbf{out}(\mathsf{c}, spkA)$;
$\mathbf{out}(\mathsf{c}, pkB)$;
$(!PA(sskA, pkB) \mid !PB(skB, spkA))$

// PA in 2nd version
let $PA(sskA$: sskey, pkB:pkey$) =$
in(c, $xpkB$:pkey);
new k:key;
out(c, aenc(sign(k2b(k), $sskA$), $xpkB$));
in(c, x:bitstring);
let $z =$ sdec(x, k) **in** 0.

// Recall PA in 1st version
let $PA(sskA$: sskey, pkB:pkey$) =$
new k:key;
out(c, aenc(sign(k2b(k), $sskA$), pkB));
in(c, x:bitstring);
let $z =$ sdec(x, k) **in** 0.

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

//Results in 1st version:

Query not attacker(s[]) is true.

//Results in 2st version:

Query not attacker(s[]) is false.

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. **理论**: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

问 (1): How to ensure the code is well-typed?
答: Type System

- Type environment Γ
  - The type system uses a type environment Γ that maps variables and names to their type.
  - This type environment initially contains the types of the free names of the closed process under consideration.
- The type system defines three judgments:
  - $\Gamma \vdash M : T$ – the term M is well-typed of type T in the type environment Γ
  - $\Gamma \vdash D : T$ – the expression D is well-typed of type T in the type environment Γ
  - $\Gamma \vdash P$ – the process P is well-typed in the type environment .

$$\frac{(x:T)\in\Gamma}{\Gamma\vdash x:T} \qquad \frac{(a:T)\in\Gamma}{\Gamma\vdash a:T}$$

$$\frac{f(T_1,\ldots,T_n):T \quad \Gamma\vdash M_1:T_1 \quad \ldots \quad \Gamma\vdash M_n:T_n}{\Gamma\vdash f(M_1,\ldots,M_n):T}$$

$$\frac{h(T_1,\ldots,T_n):T \quad \Gamma\vdash D_1:T_1 \quad \ldots \quad \Gamma\vdash D_n:T_n}{\Gamma\vdash h(D_1,\ldots,D_n):T}$$

$$\Gamma\vdash \mathsf{fail}:T$$

$$\frac{\Gamma\vdash N:\mathsf{channel} \quad \Gamma\vdash M:T \quad \Gamma\vdash P}{\Gamma\vdash \mathsf{out}(N,M);P}$$

$$\frac{\Gamma\vdash N:\mathsf{channel} \quad \Gamma,x:T\vdash P}{\Gamma\vdash \mathsf{in}(N,x:T);P}$$

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

问 (2): How to translate the code into *Horn clauses*?
答:

1. Define basic predicates
2. Model Dolev-Yao *adversary* in Horn clauses
3. Translate the *processes* into Horn clauses

问 (2): How to translate the code into *Horn clauses*?
答:

1. Define basic predicates
2. Model Dolev-Yao *adversary* in Horn clauses
3. Translate the *processes* into Horn clauses

问 (2): How to translate the code into *Horn clauses*?
答:

1. Define basic predicates
2. Model Dolev-Yao *adversary* in Horn clauses
3. Translate the *processes* into Horn clauses

问 (2.1): Define basic predicates

- $\mathbf{attacker}(mp)$: the adversary *may* have mp
- $\mathbf{message}(p, p')$ means that the message p' may appear on channel p
- ...

问 (2.2): Model Dolev-Yao *adversary* in Horn clauses
答: 回顾: The *adversary* can *overhear*, *intercept*, and *synthesize* any message

- *Overhear*: $\mathbf{message}(x, y) \land \mathbf{attacker}(x) \Rightarrow \mathbf{attacker}(y)$
  - the adversary can listen on all channels it has
- *Intercept*: $\mathbf{attacker}(x) \land \mathbf{attacker}(y) \Rightarrow \mathbf{message}(x, y)$
  - it can send all messages it has on all channels it has
- *Synthesize*:
  - $\mathbf{attacker}(x_1) \land \cdots \land \mathbf{attacker}(x_n) \Rightarrow \mathbf{attacker}(f(x_1, \ldots, x_n))$
    - constructor f
  - $\mathbf{attacker}(U_1) \land \cdots \land \mathbf{attacker}(U_n) \Rightarrow \mathbf{attacker}(U)$
    - destructor $g(U_1, \ldots, U_n) = U$

注: 这里 $\Rightarrow$ 即为一阶逻辑里的 $\rightarrow$

问 (2.2): Model Dolev-Yao *adversary* in Horn clauses
答: 回顾: The *adversary* can *overhear*, *intercept*, and *synthesize* any message

- *Overhear*: $\mathbf{message}(x, y) \land \mathbf{attacker}(x) \Rightarrow \mathbf{attacker}(y)$
  - the adversary can listen on all channels it has
- *Intercept*: $\mathbf{attacker}(x) \land \mathbf{attacker}(y) \Rightarrow \mathbf{message}(x, y)$
  - it can send all messages it has on all channels it has
- *Synthesize*:
  - $\mathbf{attacker}(x_1) \land \cdots \land \mathbf{attacker}(x_n) \Rightarrow \mathbf{attacker}(f(x_1, \ldots, x_n))$
    - constructor f
  - $\mathbf{attacker}(U_1) \land \cdots \land \mathbf{attacker}(U_n) \Rightarrow \mathbf{attacker}(U)$
    - destructor $g(U_1, \ldots, U_n) = U$

注: 这里 $\Rightarrow$ 即为一阶逻辑里的 $\rightarrow$

问 (2.2): Model Dolev-Yao *adversary* in Horn clauses
答: 回顾: The *adversary* can *overhear*, *intercept*, and *synthesize* any message

- *Overhear*: $\mathbf{message}(x, y) \land \mathbf{attacker}(x) \Rightarrow \mathbf{attacker}(y)$
  - the adversary can listen on all channels it has
- *Intercept*: $\mathbf{attacker}(x) \land \mathbf{attacker}(y) \Rightarrow \mathbf{message}(x, y)$
  - it can send all messages it has on all channels it has
- *Synthesize*:
  - $\mathbf{attacker}(x_1) \land \cdots \land \mathbf{attacker}(x_n) \Rightarrow \mathbf{attacker}(f(x_1, \ldots, x_n))$
    - constructor f
  - $\mathbf{attacker}(U_1) \land \cdots \land \mathbf{attacker}(U_n) \Rightarrow \mathbf{attacker}(U)$
    - destructor $g(U_1, \ldots, U_n) = U$

注: 这里 $\Rightarrow$ 即为一阶逻辑里的 $\rightarrow$

问 (2.2): Model Dolev-Yao *adversary* in Horn clauses
答: 回顾: The *adversary* can *overhear*, *intercept*, and *synthesize* any message

- *Overhear*: $\mathbf{message}(x, y) \land \mathbf{attacker}(x) \Rightarrow \mathbf{attacker}(y)$
  - the adversary can listen on all channels it has
- *Intercept*: $\mathbf{attacker}(x) \land \mathbf{attacker}(y) \Rightarrow \mathbf{message}(x, y)$
  - it can send all messages it has on all channels it has
- *Synthesize*:
  - $\mathbf{attacker}(x_1) \land \cdots \land \mathbf{attacker}(x_n) \Rightarrow \mathbf{attacker}(f(x_1, \ldots, x_n))$
    - constructor f
  - $\mathbf{attacker}(U_1) \land \cdots \land \mathbf{attacker}(U_n) \Rightarrow \mathbf{attacker}(U)$
    - destructor $g(U_1, \ldots, U_n) = U$

注: 这里 $\Rightarrow$ 即为一阶逻辑里的 $\rightarrow$

问 (2.2): Model Dolev-Yao *adversary* in Horn clauses
答: 回顾: The *adversary* can *overhear*, *intercept*, and *synthesize* any message

- *Overhear*: $\mathbf{message}(x, y) \land \mathbf{attacker}(x) \Rightarrow \mathbf{attacker}(y)$
  - the adversary can listen on all channels it has
- *Intercept*: $\mathbf{attacker}(x) \land \mathbf{attacker}(y) \Rightarrow \mathbf{message}(x, y)$
  - it can send all messages it has on all channels it has
- *Synthesize*:
  - $\mathbf{attacker}(x_1) \land \cdots \land \mathbf{attacker}(x_n) \Rightarrow \mathbf{attacker}(f(x_1, \ldots, x_n))$
    - constructor f
  - $\mathbf{attacker}(U_1) \land \cdots \land \mathbf{attacker}(U_n) \Rightarrow \mathbf{attacker}(U)$
    - destructor $g(U_1, \ldots, U_n) = U$

注: 这里 $\Rightarrow$ 即为一阶逻辑里的 $\rightarrow$

Examples for *Synthesize*:

- Constructors

$$\text{attacker}(m) \land \text{attacker}(k) \Rightarrow \text{attacker}(\text{senc}(m, k)) \quad \text{(senc)}$$

$$\text{attacker}(sk) \Rightarrow \text{attacker}(\text{pk}(sk)) \quad \text{(pk)}$$

$$\text{attacker}(m) \land \text{attacker}(pk) \Rightarrow \text{attacker}(\text{aenc}(m, pk)) \quad \text{(aenc)}$$

$$\text{attacker}(m) \land \text{attacker}(sk) \Rightarrow \text{attacker}(\text{sign}(m, sk)) \quad \text{(sign)}$$

- Destructors

$$\text{attacker}(\text{senc}(m, k)) \land \text{attacker}(k) \Rightarrow \text{attacker}(m) \quad \text{(sdec)}$$

$$\text{attacker}(\text{aenc}(m, \text{pk}(sk))) \land \text{attacker}(sk) \Rightarrow \text{attacker}(m) \quad \text{(adec)}$$

$$\text{attacker}(\text{sign}(m, sk)) \land \text{attacker}(\text{pk}(sk)) \Rightarrow \text{attacker}(m) \quad \text{(check)}$$

问 (2.3): Translate the *processes* into Horn clauses

答: The translation $[\![P]\!]\rho sH$ of a process P is a set of *clauses*

- ρ is an *environment* that associates a *pattern* with each *name* and *variable*,
- s is a *sequence* of *patterns*, representing the current values of *session identifiers* and *inputs*
- H is a *sequence* of *facts*, representing the *hypothesis* of the clauses

$$[\![0]\!]\rho sH = \emptyset$$

$$[\![P \mid Q]\!]\rho sH = [\![P]\!]\rho sH \cup [\![Q]\!]\rho sH$$

$$[\![!P]\!]\rho sH = [\![P]\!]\rho(s, i)H \text{ where } i \text{ is a fresh variable}$$

$$[\![\text{new } a; P]\!]\rho sH = [\![P]\!](\rho[a \mapsto a[s]])sH$$

$$[\![\text{in}(M, x); P]\!]\rho sH = [\![P]\!](\rho[x \mapsto x'])(s, x')(H \wedge \text{message}(\rho(M), x'))$$
$$\text{where } x' \text{ is a fresh variable}$$

$$[\![\text{out}(M, N); P]\!]\rho sH = [\![P]\!]\rho sH \cup \{H \Rightarrow \text{message}(\rho(M), \rho(N))\}$$

问 (2.3): Translate the *processes* into Horn clauses

答: The translation $[\![P]\!]\rho sH$ of a process P is a set of *clauses*

- ρ is an *environment* that associates a *pattern* with each *name* and *variable*,
- s is a *sequence* of *patterns*, representing the current values of *session identifiers* and *inputs*
- H is a *sequence* of *facts*, representing the *hypothesis* of the clauses

$$[\![\mathbf{0}]\!]\rho sH = \emptyset$$

$$[\![P \mid Q]\!]\rho sH = [\![P]\!]\rho sH \cup [\![Q]\!]\rho sH$$

$$[\![!P]\!]\rho sH = [\![P]\!]\rho(s, i)H \text{ where } i \text{ is a fresh variable}$$

$$[\![\mathsf{new}\ a; P]\!]\rho sH = [\![P]\!](\rho[a \mapsto a[s]])sH$$

$$[\![\mathsf{in}(M, x); P]\!]\rho sH = [\![P]\!](\rho[x \mapsto x'])(s, x')(H \wedge \mathsf{message}(\rho(M), x'))$$

where x' is a fresh variable

$$[\![\mathsf{out}(M, N); P]\!]\rho sH = [\![P]\!]\rho sH \cup \{H \Rightarrow \mathsf{message}(\rho(M), \rho(N))\}$$

Example:

$$P_0 = \text{new } ssk_A : \text{skey}; \text{new } sk_B : \text{skey}; \text{let } spk_A = \text{pk}(ssk_A) \text{ in}$$
$$\text{let } pk_B = \text{pk}(sk_B) \text{ in } \text{out}(c, spk_A); \text{out}(c, pk_B);$$
$$(P_A(ssk_A, pk_B) \mid P_B(sk_B, spk_A))$$

From P_A, we can obtain a Horn clause:

$$\text{attacker}(\text{pk}(ssk_A[\,])) \tag{3.1}$$
$$\text{attacker}(\text{pk}(sk_B[\,])) \tag{3.2}$$

Example:

$$P_A(ssk_A, pk_B) = !\,\mathsf{in}(c, x_{pk_B} : \mathsf{pkey}); \mathsf{new}\ k : \mathsf{key};$$
$$\mathsf{out}(c, \mathsf{aenc}(\mathsf{sign}(\mathsf{k2b}(k), ssk_A), x_{pk_B}));$$
$$\mathsf{in}(c, x : \mathsf{bitstring}); \mathsf{let}\ z = \mathsf{sdec}(x, k)\ \mathsf{in}\ \mathbf{0}$$

From P_A, we can obtain a Horn clause:

$$\mathsf{attacker}(x_{pk_B}) \Rightarrow \mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(k[i, x_{pk_B}], ssk_A[\,]), x_{pk_B})) \tag{3.3}$$

Example:

$$P_B(sk_B, spk_A) = !\,\mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in}$$
$$\mathsf{let}\ x_k = \mathsf{b2k}(\mathsf{check}(y', spk_A))\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))$$

From P_B, we can obtain a Horn clause:

$$\mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(x_m, ssk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \mathsf{attacker}(\mathsf{senc}(s[\,], x_m)) \tag{3.4}$$

Totally, $\mathbf{attacker}(s[])$ is derivable from the above clauses.
Let's see the generated clauses:
命令: $ ./proverif -test DenningSacco-simple2.pv
部分结果: 见后页

Example:

$$P_B(sk_B, spk_A) = !\,\mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in}$$
$$\mathsf{let}\ x_k = \mathsf{b2k}(\mathsf{check}(y', spk_A))\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))$$

From P_B, we can obtain a Horn clause:

$$\mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(x_m, ssk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \mathsf{attacker}(\mathsf{senc}(s[\,], x_m)) \tag{3.4}$$

Totally, $\mathbf{attacker}(s[])$ is derivable from the above clauses.

Let's see the generated clauses:

命令: $ ./proverif -test DenningSacco-simple2.pv

部分结果: 见后页

Example:

$$P_B(sk_B, spk_A) = !\,\mathsf{in}(c, y : \mathsf{bitstring}); \mathsf{let}\ y' = \mathsf{adec}(y, sk_B)\ \mathsf{in}$$
$$\mathsf{let}\ x_k = \mathsf{b2k}(\mathsf{check}(y', spk_A))\ \mathsf{in}\ \mathsf{out}(c, \mathsf{senc}(s, x_k))$$

From P_B, we can obtain a Horn clause:

$$\mathsf{attacker}(\mathsf{aenc}(\mathsf{sign}(x_m, ssk_A[\,]), \mathsf{pk}(sk_B[\,]))) \Rightarrow \mathsf{attacker}(\mathsf{senc}(s[\,], x_m)) \tag{3.4}$$

Totally, $\mathbf{attacker}(s[])$ is derivable from the above clauses.
Let's see the generated clauses:
命令: $ ./proverif -test DenningSacco-simple2.pv
部分结果: 见后页

```
-- Query not attacker(s[]) in process 1.
Translating the process into Horn clauses...
Initial clauses:
Clause 0: attacker(true)
(The attacker applies function true.)

Clause 1: attacker(v) -> attacker(spk(v))
(The attacker applies function spk.)

Clause 2: attacker(v) && attacker(v_1) -> attack
er(sign(v,v_1))
(The attacker applies function sign.)

Clause 3: attacker(v) && attacker(v_1) -> attack
er(senc(v,v_1))
(The attacker applies function senc.)
```

```
Clause 4: attacker(senc(x_1,y_1)) && attacker(y_
1) -> attacker(x_1)
(The attacker applies function sdec.)

Clause 5: attacker(v) -> attacker(pk(v))
(The attacker applies function pk.)

Clause 6: attacker(false)
(The attacker applies function false.)

Clause 7: attacker(sign(m,k_1)) && attacker(spk(
k_1)) -> attacker(m)
(The attacker applies function checksign.)
```

Totally, $\mathbf{attacker}(s[])$ is derivable from the above clauses.

- This derivation corresponds to the following well-known attack (Abadi and Needham, 1996) against this protocol:

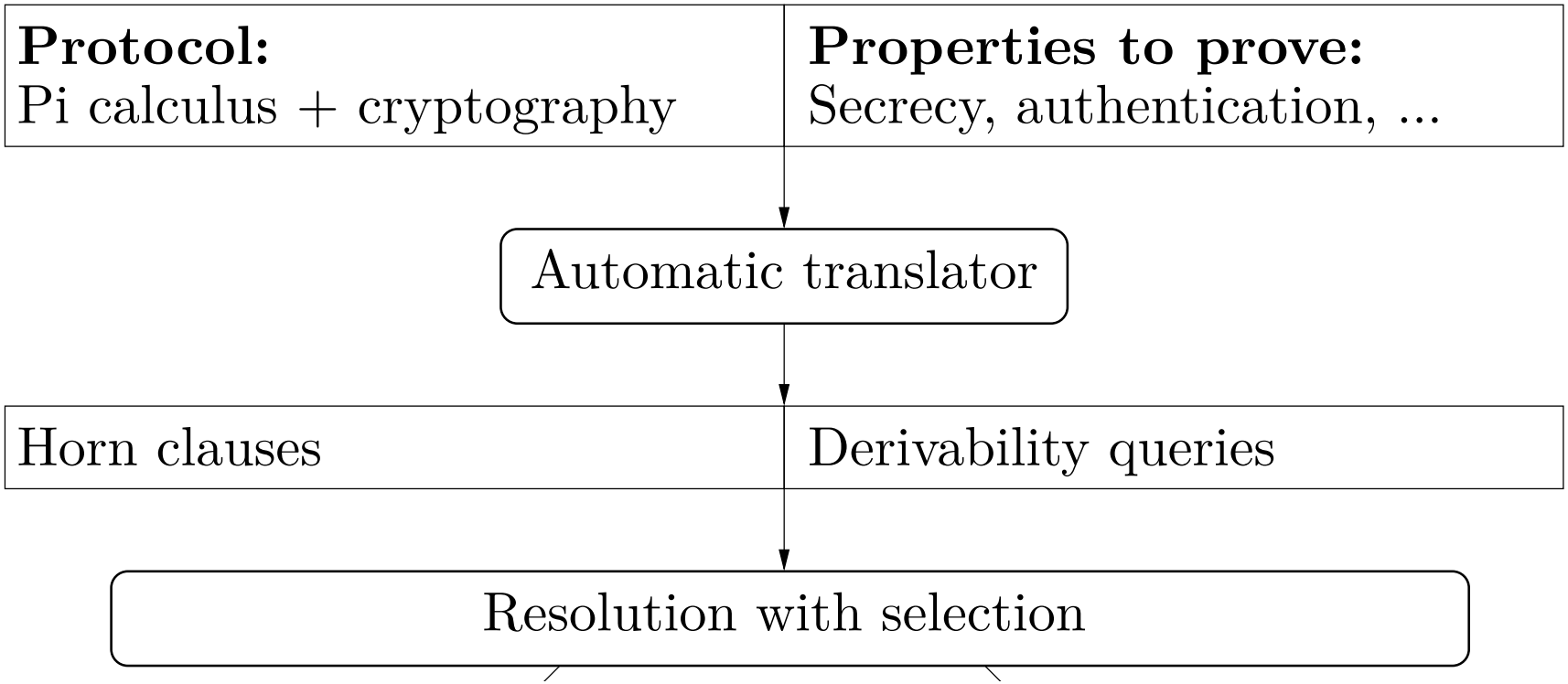
  Message 1. $A \to C$: $\mathsf{aenc}(\mathsf{sign}(k, ssk_A), pk_C)$
  Message 1'. $C(A) \to B$: $\mathsf{aenc}(\mathsf{sign}(k, ssk_A), pk_B)$
  Message 2. $B \to C(A)$: $\mathsf{senc}(s, k)$

1. ProVerif

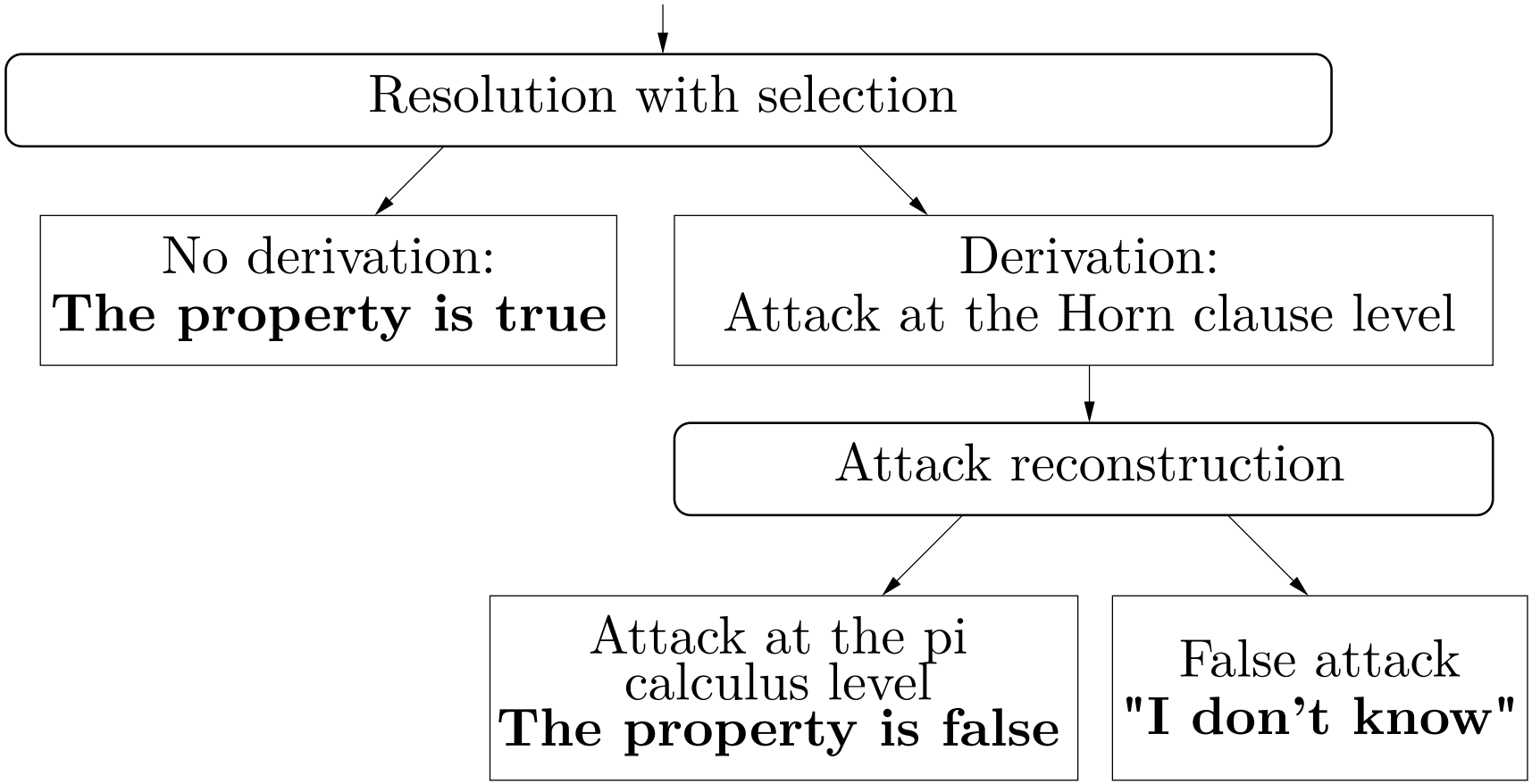
Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

Where are we now?



问 (3): How to *solve*?



Algorithm: *Resolution* Algorithm

(3.1) What is resolution?

- Given two clauses R and R'
  - $R = H \Rightarrow C, \quad R' = F \wedge H' \Rightarrow C'$
- Infer $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$
  - C and F are *unifiable*, σ is the most general *unifier* of C and F

(3.2) How to guide resolution? *selection function* $sel(R)$

- returns
  - a hypothesis of R
  - the empty (meaning that the conclusion of R is selected)
- the resolution step above is performed *only when* $sel(R) = \emptyset$ and $sel(R') = \{F\}$

Then,(3.3) How to design $sel(R)$,(3.4) Which $sel(R)$ to choose? (见下页)

Algorithm: *Resolution* Algorithm

(3.1) What is resolution?

- Given two clauses R and R'
  - $R = H \Rightarrow C, \quad R' = F \wedge H' \Rightarrow C'$
- Infer $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$
  - C and F are *unifiable*, σ is the most general *unifier* of C and F

(3.2) How to guide resolution? *selection function* $sel(R)$

- returns
  - a hypothesis of R
  - the empty (meaning that the conclusion of R is selected)
- the resolution step above is performed *only when* $sel(R) = \emptyset$ and $sel(R') = \{F\}$

Then,(3.3) How to design $sel(R)$,(3.4) Which $sel(R)$ to choose? (见下页)

Algorithm: *Resolution* Algorithm

(3.1) What is resolution?

- Given two clauses R and R'
  - $R = H \Rightarrow C, \quad R' = F \wedge H' \Rightarrow C'$
- Infer $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$
  - C and F are *unifiable*, σ is the most general *unifier* of C and F

(3.2) How to guide resolution? *selection function* $sel(R)$

- returns
  - a hypothesis of R
  - the empty (meaning that the conclusion of R is selected)
- the resolution step above is performed *only when* $sel(R) = \emptyset$ and $sel(R') = \{F\}$

Then,(3.3) How to design $sel(R)$,(3.4) Which $sel(R)$ to choose? (见下页)

Algorithm: *Resolution* Algorithm

(3.1) What is resolution?

- Given two clauses R and R'
  - $R = H \Rightarrow C, \quad R' = F \wedge H' \Rightarrow C'$
- Infer $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$
  - C and F are *unifiable*, σ is the most general *unifier* of C and F

(3.2) How to guide resolution? *selection function* $sel(R)$

- returns
  - a hypothesis of R
  - the empty (meaning that the conclusion of R is selected)
- the resolution step above is performed *only when* $sel(R) = \emptyset$ and $sel(R') = \{F\}$

Then,(3.3) How to design $sel(R)$,(3.4) Which $sel(R)$ to choose? (见下页)

Algorithm: *Resolution* Algorithm

(3.1) What is resolution?

- Given two clauses R and R'
  - $R = H \Rightarrow C, \quad R' = F \wedge H' \Rightarrow C'$
- Infer $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$
  - C and F are *unifiable*, σ is the most general *unifier* of C and F

(3.2) How to guide resolution? *selection function* $sel(R)$

- returns
  - a hypothesis of R
  - the empty (meaning that the conclusion of R is selected)
- the resolution step above is performed *only when* $sel(R) = \emptyset$ and $sel(R') = \{F\}$

Then,(3.3) How to design $sel(R)$,(3.4) Which $sel(R)$ to choose? (见下页)

Algorithm: *Resolution* Algorithm

(3.1) What is resolution?

- Given two clauses R and R'
  - $R = H \Rightarrow C, \quad R' = F \wedge H' \Rightarrow C'$
- Infer $R \circ_F R' = \sigma H \wedge \sigma H' \Rightarrow \sigma C'$
  - C and F are *unifiable*, σ is the most general *unifier* of C and F

(3.2) How to guide resolution? *selection function* $sel(R)$

- returns
  - a hypothesis of R
  - the empty (meaning that the conclusion of R is selected)
- the resolution step above is performed *only when* $sel(R) = \emptyset$ and $sel(R') = \{F\}$

Then,(3.3) How to design $sel(R)$,(3.4) Which $sel(R)$ to choose? (见下页)

(3.3) How to design $sel(R)$?

这里要面对的问题: reduce possibility of *non-termination*

- a case: the *fact* $\mathbf{attacker}(v)$ where v is a *variable* or a may-fail *variable* unifies with ***any*** *fact* $\mathbf{attacker}(p)$
  - so if $\mathbf{attacker}(v)$ is selected, the algorithm will *almost never terminate*.
- other cases: **略**

So, a natural selection function is then:

$$sel_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if all elements of } H \text{ are disequalities or of the} \\ & \text{form attacker}(v),\ v \text{ variable or may-fail variable} \\ \{F\} & \text{where } F \text{ is not a disequality,} \\ & F \neq \text{attacker}(v) \text{ and } F \in H, \text{ otherwise} \end{cases}$$

(3.3) How to design $sel(R)$?

这里要面对的问题: reduce possibility of *non-termination*

- a case: the *fact* $\mathbf{attacker}(v)$ where v is a *variable* or a may-fail *variable* unifies with ***any*** *fact* $\mathbf{attacker}(p)$
  - so if $\mathbf{attacker}(v)$ is selected, the algorithm will *almost never terminate*.
- other cases: **略**

So, a natural selection function is then:

$$sel_0(H \Rightarrow C) = \begin{cases} \emptyset & \text{if all elements of } H \text{ are disequalities or of the} \\ & \text{form } \mathsf{attacker}(v),\ v \text{ variable or may-fail variable} \\ \{F\} & \text{where } F \text{ is not a disequality,} \\ & F \neq \mathsf{attacker}(v) \text{ and } F \in H, \text{ otherwise} \end{cases}$$

(3.4) Which $sel(R)$ to choose?
答: (3.4.1) use several optimizations in the resolution algorithm:

- Elimination of subsumed clauses
  - $H_1 \Rightarrow C_1$ *subsumes* $H_2 \Rightarrow C_2$, iff, there exists a substitution σ such that $\sigma H_1 \subseteq H_2$ (multiset inclusion) and $\sigma C_1 = C_2$.
- Elimination of duplicate hypotheses
- Elimination of tautologies
- ...

(3.4) Which $sel(R)$ to choose?
答: (3.4.1) use several optimizations in the resolution algorithm:

- Elimination of subsumed clauses
  - $H_1 \Rightarrow C_1$ *subsumes* $H_2 \Rightarrow C_2$, iff, there exists a substitution σ such that $\sigma H_1 \subseteq H_2$ (multiset inclusion) and $\sigma C_1 = C_2$.
- Elimination of duplicate hypotheses
- Elimination of tautologies
- ...

(3.4) Which $sel(R)$ to choose?
答: (3.4.1) use several optimizations in the resolution algorithm:

- Elimination of subsumed clauses
  - $H_1 \Rightarrow C_1$ *subsumes* $H_2 \Rightarrow C_2$, iff, there exists a substitution σ such that $\sigma H_1 \subseteq H_2$ (multiset inclusion) and $\sigma C_1 = C_2$.
- Elimination of duplicate hypotheses
- Elimination of tautologies
- ...

(3.4) Which $sel(R)$ to choose?

答: (3.4.1) use several optimizations in the resolution algorithm:

- Elimination of subsumed clauses
  - $H_1 \Rightarrow C_1$ *subsumes* $H_2 \Rightarrow C_2$, iff, there exists a substitution σ such that $\sigma H_1 \subseteq H_2$ (multiset inclusion) and $\sigma C_1 = C_2$.
- Elimination of duplicate hypotheses
- Elimination of tautologies
- ...

(3.4) Which $sel(R)$ to choose?
答: (3.4.2) use heuristics, for example:

- using destructors when possible yields better performance than equations
- adjusting the arguments of patterns that represent names
- When ProVerif does not terminate, *tuning* the selection function of the resolution algorithm may help.
  - e.g., one can tell ProVerif to *avoid selecting a fact* that matches F, by the declaration $\mathbf{nounif}\, F$.

(3.4) Which $sel(R)$ to choose?
答: (3.4.2) use heuristics, for example:

- using destructors when possible yields better performance than equations
- adjusting the arguments of patterns that represent names
- When ProVerif does not terminate, *tuning* the selection function of the resolution algorithm may help.
  - e.g., one can tell ProVerif to *avoid selecting a fact* that matches F, by the declaration $\mathbf{nounif}\, F$.

(3.4) Which $sel(R)$ to choose?
答: (3.4.2) use heuristics, for example:

- using destructors when possible yields better performance than equations
- adjusting the arguments of patterns that represent names
- When ProVerif does not terminate, *tuning* the selection function of the resolution algorithm may help.
  - e.g., one can tell ProVerif to *avoid selecting a fact* that matches F, by the declaration $\mathbf{nounif}\, F$.

(3.4) Which $sel(R)$ to choose?
答: (3.4.2) use heuristics, for example:

- using destructors when possible yields better performance than equations
- adjusting the arguments of patterns that represent names
- When ProVerif does not terminate, *tuning* the selection function of the resolution algorithm may help.
  - e.g., one can tell ProVerif to *avoid selecting a fact* that matches F, by the declaration $\mathbf{nounif}\, F$.

1. ProVerif

Outline

1. 问题介绍: Study security protocols *manually*
   - *Model:* A simple example a security protocol
   - *Assumption:* Introducing the Dolev-Yao model
   - *Specification:* Specifying the properties
   - *Algorithm:* Manual Analysis
2. 应用: *Modeling* and *Verification* using ProVerif
   - *Modeling* in language of Pi-calculus
   - *Specification:* Specifying the properties
   - *Modeling* in language of Pi-calculus
   - Run ProVerif
3. 理论: *Algorithms*
   - How to ensure the code is *well-typed*?
   - How to *translate* the code into *Horn clauses*?
   - How to *solve*?
   - Performance Analysis

问 (4): Performance Analysis

Pros

- High efficiency due to abstractions into Horn clauses

Cons

- still may not terminate
- incompleteness
  - since Horn clauses introduce approximations
  - attack reconstruction fails corresponds to an "I do not know" answer

小问: What approximation?

小答: Ignores non-monotonous *state transition*, e.g.,

- repetitions (or not) of actions are ignored
  - in case some value first needs to be kept secret and is revealed later in the protocol
- bad support for private channels
  - $\mathbf{out}(M, N)|P$: P can be executed *only after* sending N on channel M.

问 (4): Performance Analysis

Pros

- High efficiency due to abstractions into Horn clauses

Cons

- still may not terminate
- incompleteness
  - since Horn clauses introduce approximations
  - attack reconstruction fails corresponds to an "I do not know" answer

小问: What approximation?

小答: Ignores non-monotonous *state transition*, e.g.,

- repetitions (or not) of actions are ignored
  - in case some value first needs to be kept secret and is revealed later in the protocol
- bad support for private channels
  - $\mathbf{out}(M, N)|P$: P can be executed *only after* sending N on channel M.

作业

实验大作业 (可选): 做 ProVerif 相关的大实验。题目开放，下面为参考选题，但不限于下面选题

- 选题 (1): 使用 ProVerif 验证更复杂安全协议，可模仿一下 CCF A，B 类论文，来进行建模，并成功验证
- 选题 (2): 尝试设计核心验证算法，改进 ProVerif 的验证效率
- 选题 (3): 阅读其它验证器的论文和代码，自己模仿设计新的验证器，或修改核心模块 (不要求完整实现，可以只包含核心求解算法)
- 注：评分标准根据实现难度和工作量来评定，不要求完整实现所有内容，上述实现需尽可能提供完整文档

本章节大作业参考论文

大作业可参考论文 (但不限于下列论文):

- 应用
  - Just fast keying in the pi calculus
  - Hash Gone Bad: Automated discovery of protocol attacks that exploit hash function weaknesses
  - Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate
  - Automated Formal Analysis of a Protocol for Secure File Sharing on Untrusted Storage
  - Election Verifiability with ProVerif
  - A Formal Analysis of 5G Authentication
- 工具实现
  - The TAMARIN Prover for the Symbolic Analysis of Security Protocols
  - An Efficient Cryptographic Protocol Verifier Based on Prolog Rules