ProVerif ハンズオン

FWS 準備委員会 *

Contents

1 ハンズオンの概要 **1**

2 準備 **2**
2.1 オンラインデモの使い方 . . . 2
2.2 事前知識 . . . 3
2.2.1 暗号プロトコルとその安全性 . . . 3
2.2.2 Dolev-Yao モデル . . . 3

3 ProVerif を用いた暗号プロトコルの安全性検証 **4**
3.1 最初の例 . . . 4
3.2 公開鍵暗号を使ってみよう . . . 6
3.3 電子署名を使ってみよう＆認証要件を検証してみよう . . . 9
3.4 クエリの種類 . . . 11

4 発展演習 **11**

5 おわりに **12**

1 ハンズオンの概要

ProVerif は暗号プロトコルの形式検証ツールです．暗号プロトコルと検証したい安全性要件を入力すると，そのプロトコルが（検証可能な範囲内で）要件を満たしているか，満たしていないかを出力してくれます．TLS 1.3 や 5G 認証プロトコルなどの様々なプロトコルの検証に用いられており，その安全性の向上に貢献しています [1]．

このハンズオンでは，ProVerif の簡単な使い方を習得することをゴールとします．具体的には，ProVerif における

- 暗号プロトコルの記述方法
- 安全性要件の記述方法
- 検証結果の読み方

の基礎を理解してもらうことをゴールとします．時間の都合上，細かい文法や記述方法については扱いません．興味を持っていただけた方は ProVerif の公式マニュアル [BSCS21] などを参照してください．

なお，本テキストで扱うコードは FWS の Web ページ

- `https://www.iwsec.org/fws/2023/`

から入手可能です．

*中林美郷（NTT），花谷嘉一（東芝），吉田真紀（NICT），米山一樹（茨城大学）

[1] ProVerif の公式ページ [Bla21] に関連論文の一覧があります．検証例についてはそちらをご覧ください．

2 準備

2.1 オンラインデモの使い方

このハンズオンでは ProVerif のオンラインデモ [MB21] を使用します．

- http://proverif20.paris.inria.fr/index.php

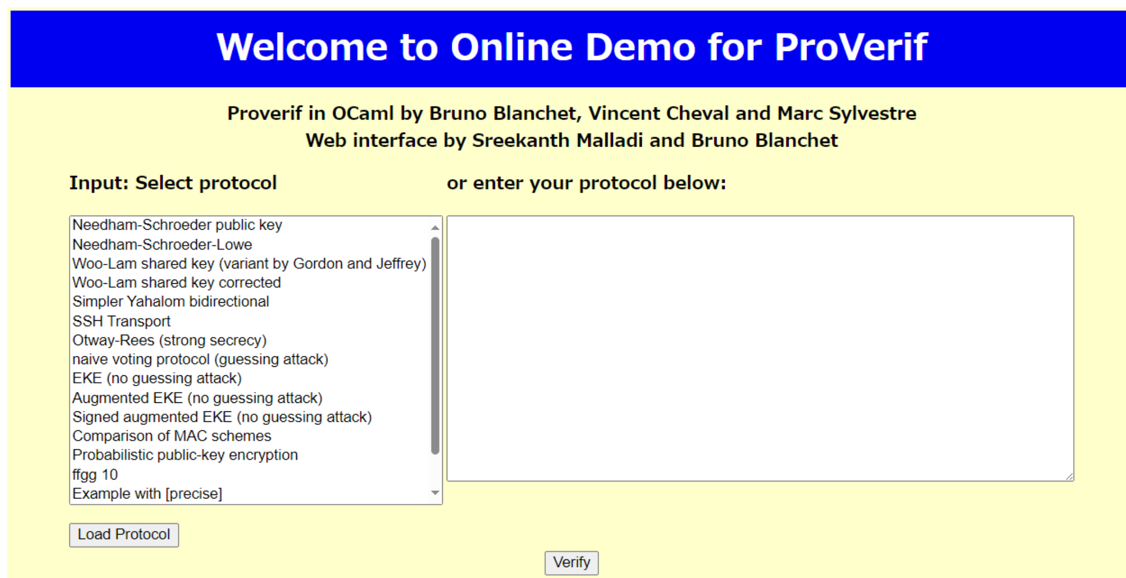

図 1: ProVerif オンラインデモの画面．

右側のフォームにコードを入力し，Verify ボタンを押すと検証が始まり，検証結果が出力されます．検証結果の最下部に Verification summary という部分があるので，そこを見ると各安全性要件（クエリ）に対する検証結果が分かります（図 2）．true が攻撃発見なし，false が攻撃発見を表します．

ProVerif HTML output

The HTML output is kept for at least 30 minutes. It is kept much longer if there is enough free space. You can obviously save the web pages if you want to keep them.

ProVerif text output:

```
Query not attacker_bitstring(secretANa[]) is true.
Query not attacker_bitstring(secretANb[]) is true.
Query not attacker_bitstring(secretBNa[]) is false.
Query not attacker_bitstring(secretBNb[]) is false.
Query inj-event(endBparam(x,y)) ==> inj-event(beginBparam(x,y)) is false.
Query inj-event(endBfull(x1,x2,x3,x4,x5,x6)) ==> inj-event(beginBfull(x1,x2,x3,x4,x5,x6)) is false.
Query inj-event(endAparam(x,y)) ==> inj-event(beginAparam(x,y)) is true.
Query inj-event(endAfull(x1,x2,x3,x4,x5,x6)) ==> inj-event(beginAfull(x1,x2,x3,x4,x5,x6)) is true.
--------------------------------------------------------------
```

There is no security mechanism to protect the confidentiality of your data, so you should not enter confidential data in this form. If you want to verify a confidential protocol, please download and install your own copy of ProVerif.

図 2: ProVerif の出力の例．

ProVerif HTML output では，より詳細な出力を見ることができます．false の場合は ProVerif HTML output の Trace graph から発見された攻撃のトレース（図 3）を見ることができます．

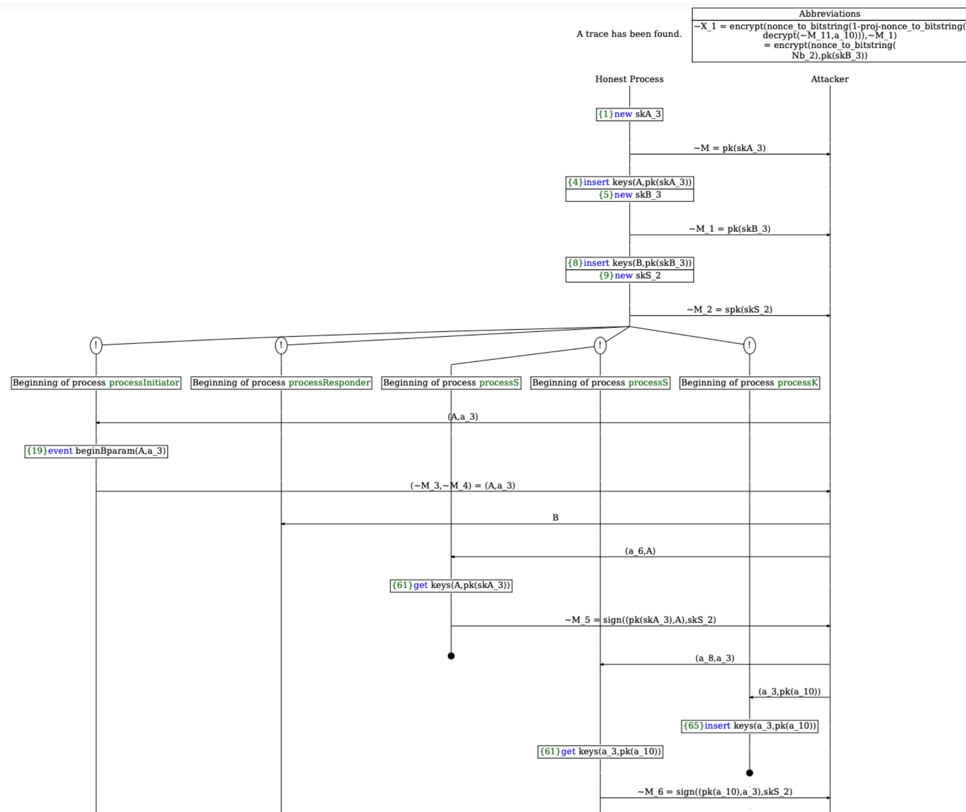

図 3: ProVerif が出力する攻撃トレースの出力の例（一部）.

2.2　事前知識

2.2.1　暗号プロトコルとその安全性

暗号プロトコルとは，通信の機密性や完全性を保証するための通信プロトコルで，公開鍵暗号や電子署名，ハッシュ関数などの暗号プリミティブを用いて構成されます．

鍵交換プロトコル，認証プロトコル，電子投票プロトコルなどがその一例です．このハンズオンでは主に鍵交換プロトコルと認証プロトコルにフォーカスします．

鍵交換プロトコルは通信相手と安全に共通の鍵（セッション鍵）を共有するプロトコルです．通信路の盗聴や改ざんができる攻撃者（アクティブな攻撃者）に対するセッション鍵の機密性などが主要な安全性要件です．認証プロトコルは電子署名や共通の秘密を用いて通信相手が正しいことを確認するプロトコルです．認証が受理された場合その相手は必ず正しい通信相手であること（ここでは認証要件と呼ぶことにします）などが主要な安全性要件です．通信相手の認証を行い，かつ鍵交換を行うプロトコルを認証付き鍵交換プロトコルと呼びます．より詳しい情報は [BMS19] などを参照してください．

2.2.2　Dolev-Yao モデル

ProVerif による検証 [2]では，Dolev-Yao モデル [DY83] という攻撃者モデルを考えています．Dolev-Yao モデルでは暗号プリミティブは理想的なものとして扱われます．すなわち，共通鍵暗号方式では暗号文に対応する復号鍵をもっているときに限り平文を手に入れることができ，電子署名方式では署名鍵を持っているときに限り正しく検証される署名を作ることができると仮定されます．メッセージや鍵などの情報を文字列ではなく記号的に扱うため，シンボリックモデルとも呼ばれています．その記号的な操作を用いて，中間者攻撃やリプレイ攻撃，メッセージの順番を入れ替える攻撃などを捉えることができます．

[2]（小噺その１）ちなみに，ProVerif はいつ頃から使われていたかご存じでしょうか．2003 年頃から使われていたため，なんと今年で約 20 歳です．結構昔からありますね．

3 ProVerifを用いた暗号プロトコルの安全性検証

ProVerif の入力コードは，主に次の 4 つの部分から構成されます．

- **宣言**．通信路や変数の型，プロトコル内で使用する暗号プリミティブを定義する．
- **クエリ**．検証したい安全性要件を記述する．
- **マクロ**．各プロトコル参加者の動作をサブプロセスとして記述する．
- **メインプロセス**．プロトコル実行を定義する．

3.1 最初の例

例として，次の二者間鍵交換プロトコル P を考えてみましょう． パーティA がナンス N_A を生成して B

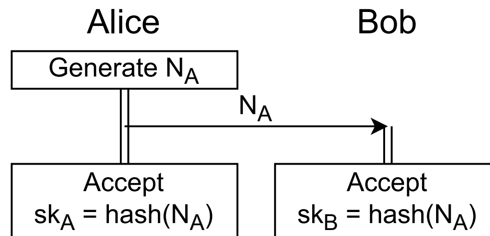

図 4: 二者間鍵交換プロトコル P.

に送り，そのハッシュ値 $\mathbf{hash}(N_A)$ をセッション鍵として受理するプロトコルです．プロトコルに詳しい方はちょっと違和感を感じるかもしれませんが，どうぞお付き合いください．このプロトコルのアクティブな攻撃者に対するセッション鍵の機密性を検証するコードは次のようになります．

Code 1: 二者間鍵交換プロトコル P

```
1 (* Setting of channels and types *)
2 free c:channel.
3
4 (* Hash function *)
5 fun hash(bitstring):bitstring.
6
7 (* Symmetric key encryption *)
8 fun senc(bitstring, bitstring):bitstring.
9 reduc forall message:bitstring, key:bitstring; sdec(senc(message, key), key) = message.
10
11 (* Queries *)
12 free message_for_verification_A:bitstring[private].
13 query attacker(message_for_verification_A).
14 free message_for_verification_B:bitstring[private].
15 query attacker(message_for_verification_B).
16
17 (* Behavior of Alice *)
18 let Alice() =
19     new Na:bitstring;
20     out(c, Na);
21     let sk_Alice = hash(Na) in
22     out(c, senc(message_for_verification_A, sk_Alice)).
23
24 (* Behavior of Bob *)
25 let Bob() =
26     in(c, Na_recv:bitstring);
27     let sk_Bob = hash(Na_recv) in
28     out(c, senc(message_for_verification_B, sk_Bob)).
29
30 (* Start process *)
31 process
32 ((!Alice()) | !Bob())
```

1 行目から 9 行目が宣言部，11 行目から 15 行目がクエリ部，17 行目から 28 行目がマクロ部，30 行目から 32 行目がメインプロセス部です．それぞれについて簡単に説明しますが，まずは重要なのはクエリ部とマクロ部です．それ以外の部分はおまじないとして捉えてもらっても構いません．

宣言部（1 行目から 9 行目）

2 行目は通信路の定義です．c という盗聴・改ざん可能な通信路を定義しています．

4 行目から 9 行目は暗号プリミティブの定義です．5 行目でハッシュ関数を，8 行目・9 行目で共通鍵暗号を定義しています [3]．ProVerif では，暗号プリミティブはこのように暗号化関数などの暗号部品を指定するコンストラクタ（5 行目や 8 行目）と復号条件などを指定するデストラクタ（9 行目）のペアによって定義されます．ハッシュ関数は復号条件を指定する必要がないため，コンストラクタのみで定義されます．ここで定義される関数は攻撃者も自由に使うことができます．

クエリ部（11 行目から 15 行目）

12 行目・13 行目は A のセッション鍵 sk_A の機密性を検証するクエリです．ProVerif の仕様の都合上，仮の検証用メッセージを定義し（12 行目），検証したい秘密情報を共通鍵としてそのメッセージを暗号化した暗号文を通信路に流し（22 行目），攻撃者が検証用メッセージを計算できるかどうかを見る（13 行目）というちょっと回りくどい方法で機密性の検証を表現しています．14 行目・15 行目も同様に，B のセッション鍵 sk_B の機密性を検証するクエリです．

マクロ部（17 行目から 28 行目）

17 行目から 22 行目が A の動作を表しています．ナンス N_A を生成し（19 行目），それを通信路に流します（20 行目）．そしてセッション鍵を計算します（21 行目）．22 行目は上で述べた通りです．24 行目から 28 行目は B の動作を表しており，ナンス N_A を受け取り（26 行目），セッション鍵を計算します（27 行目）．

メインプロセス部 (30 行目から 32 行目)

30 行目から 32 行目で，マクロ部で定義したサブプロセスを動かします．このように書くことで，無限個のセッションを並列に動作させることができます．

> **演習 3.1.** コード 1をオンラインデモに入力し，出力結果を確認してみましょう．

次のような結果が出力されたと思います．

- `Query not attacker(message_for_verification_A[]) is false.`
- `Query not attacker(message_for_verification_B[]) is false.`

両方とも `false` が出力されているため，A のセッション鍵の機密性，B のセッション鍵の機密性それぞれに対する攻撃が発見されています．ついでに `ProVerif HTML output` から攻撃トレースも見てみましょう．図 5と同じものが確認できると思います． トレースを見てみると，攻撃者は通信路を盗聴して N_A を盗み，セッション鍵 $\mathbf{hash}(N_A)$ を計算しています．

[3]（小噺その 2）この定義では，共通鍵暗号の暗号文を `senc(message, key)` としています．この定義を見て，なんだか気持ち悪いと感じる方も多いのではないでしょうか．暗号界隈では引数を「鍵，平文」の順番で書くことが多いらしいのですが，形式検証界隈では「平文，鍵」の順で書かれることが多いです．

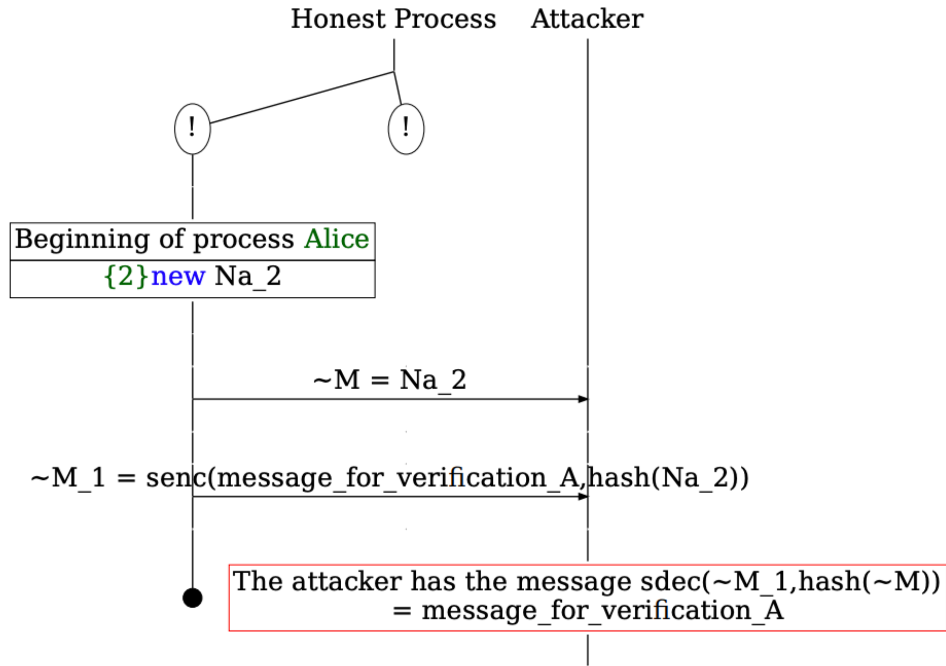

図 5: コード 1 の A のセッション鍵の機密性についての攻撃トレース.

3.2　公開鍵暗号を使ってみよう

上の例を機密性を満たすように修正していきましょう．通信路に平文でセッション鍵の要素となる情報（N_A）を流していたことが攻撃につながっていたため，公開鍵暗号を用いて通信路に情報を流すようにしたいと思います．次のプロトコル P' を考えてみます． ここで，$\mathrm{pk}_B, \mathrm{sk}_B$ は B の公開鍵と秘密鍵です．

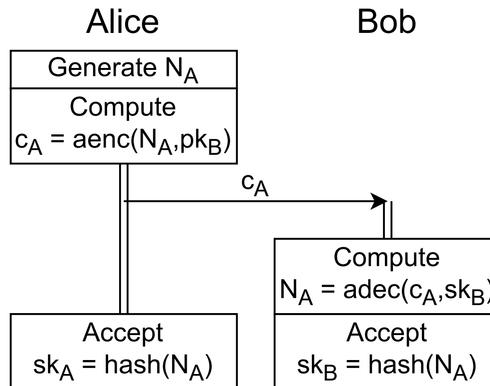

図 6: 二者間鍵交換プロトコル P'.

公開鍵暗号は次のコンストラクタとデストラクタで定義されます.

```
1 (* Asymmetric key encryption *)
2 fun pk(bitstring):bitstring.
3 fun aenc(bitstring, bitstring):bitstring.
4 reduc forall message:bitstring, key:bitstring; adec(aenc(message, pk(key)), key) =
    message.
```

公開鍵 `pk` を用いたメッセージ `m` の暗号文は `aenc(m, pk)`，復号鍵 `sk` を用いた暗号文 `c` の復号結果は `adec(c, sk)` で表されます（正しい復号鍵が使われた時に限り，正しい復号結果を得ることができます）.

演習 3.2. コード 2の ??? 部分を埋めて，プロトコル P' を検証するコードを完成させてください．

Code 2: 二者間鍵交換プロトコル P'

```
 1 (* Setting of channels *)
 2 free c:channel.
 3
 4 (* Hash function *)
 5 fun hash(bitstring):bitstring.
 6
 7 (* Symmetric key encryption *)
 8 fun senc(bitstring, bitstring):bitstring.
 9 reduc forall m:bitstring, k:bitstring; sdec(senc(m, k), k) = m.
10
11 (* Asymmetric key encryption *)
12 fun pk(bitstring):bitstring.
13 fun aenc(bitstring, bitstring):bitstring.
14 reduc forall m:bitstring, k:bitstring; adec(aenc(m, pk(k)), k) = m.
15
16 (* Queries *)
17 free message_for_verification_A:bitstring[private].
18 query attacker(message_for_verification_A).
19 free message_for_verification_B:bitstring[private].
20 query attacker(message_for_verification_B).
21
22 (* Behavior of Alice *)
23 let Alice(pkA:bitstring, pkB:bitstring, skA:bitstring) =
24     new Na:bitstring;
25     let Cipher_A = ???(1) in
26     out(c, Cipher_A);
27     let sk_Alice = hash(Na) in
28     out(c, senc(message_for_verification_A, sk_Alice)).
29
30 (* Behavior of Bob *)
31 let Bob(pkB:bitstring, pkA:bitstring, skB:bitstring) =
32     in(c, Cipher_A_recv:bitstring);
33     let Na = ???(2) in
34     let sk_Bob = hash(Na) in
35     out(c, senc(message_for_verification_B, sk_Bob)).
36     new sskA:bitstring; let spkA = spk(sskA) in out(c, spkA);
37
38 (* Start process *)
39 process new skA:bitstring; let pkA = pk(skA) in out(c, pkA);
40     new skB:bitstring; let pkB = pk(skB) in out(c, pkB);
41 ((!Alice(pkA, pkB, skA)) | !Bob(pkB, pkA, skB))
```

変数名は次の通りです．

- A の公開鍵：pkA，A の秘密鍵：skA（A のサブプロセスの中でのみ使えます）．
- B の公開鍵：pkB，B の秘密鍵：skB（B のサブプロセスの中でのみ使えます）．

参考解答は FWS の Web ページにある code2_complete.pv をご覧ください．以降，参考解答の出力を想定して話を進めていきます．

code2_complete.pv の出力は次のようになります．

- `Query not attacker(message_for_verification_A[]) is true.`
- `Query not attacker(message_for_verification_B[]) is false.`

A のセッション鍵の機密性は `true` になりました．B 側のセッション鍵の機密性に対してのみ攻撃が発見されたようです．攻撃トレースを見てみましょう．

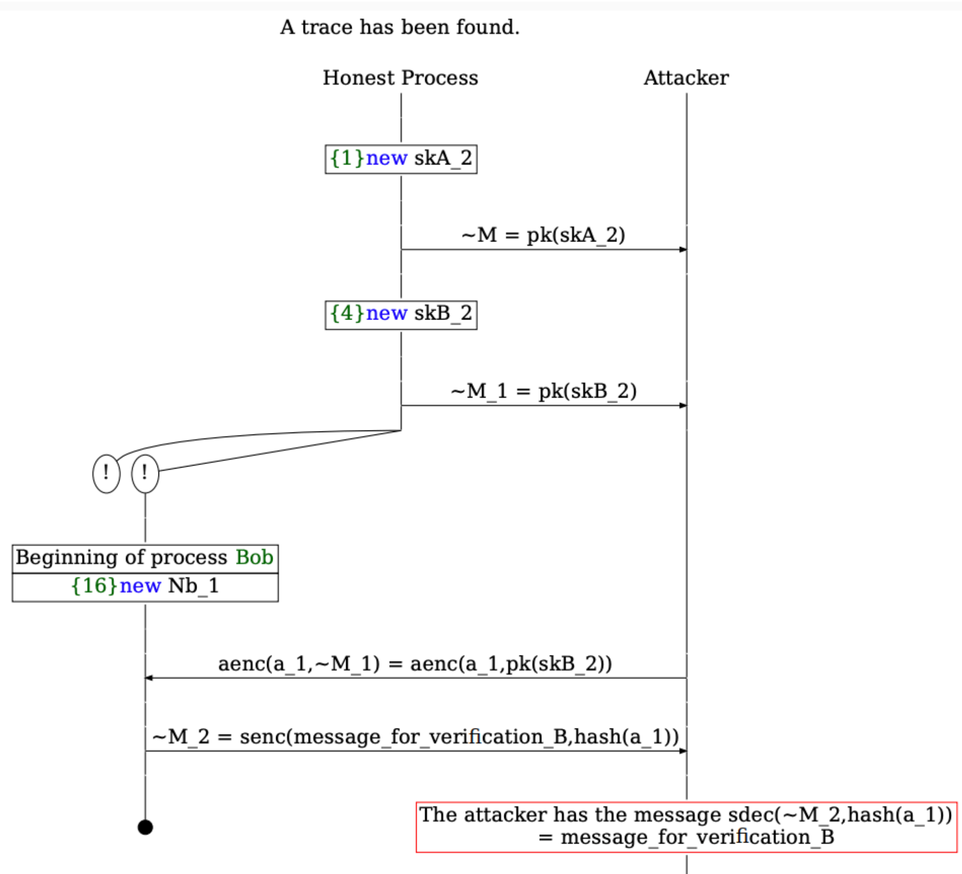

図 7: コード 2 の B のセッション鍵の機密性についての攻撃トレース．

トレースによると，攻撃者が B に対して A になりすましているようです．A から B は公開鍵を用いて（暗黙的に）認証ができているのですが，B から A は何も確かめる術がないため，なりすましが可能になっています．

3.3 電子署名を使ってみよう＆認証要件を検証してみよう

なりすましを防ぐために，電子署名を用いてみましょう．次のプロトコル P'' を考えてみます．

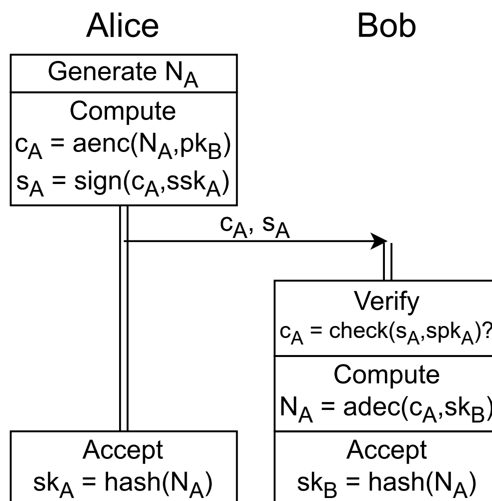

図 8: 二者間鍵交換プロトコル P''.

spk と ssk は署名用の公開鍵（検証鍵）と秘密鍵 (署名鍵) のペアです．電子署名は次のコンストラクタとデストラクタで定義されます．

```
1 (* Signatures *)
2 fun spk(bitstring):bitstring.
3 fun sign(bitstring, bitstring):bitstring.
4 reduc forall message:bitstring, key:bitstring; check(sign(message, key), spk(key)) =
      message.
```

署名鍵 `ssk` を用いたメッセージ `m` への署名は `sign(m, ssk)`，署名検証鍵 `spk` を用いた署名 `s` の検証結果は `check(s, spk)` で表されます（正しい署名の場合はメッセージが返ってきます）．

さらに，攻撃者によるなりすましが可能かどうかを直接検査するために，認証要件を検証してみましょう．認証要件はイベントという概念を用いて定義できます．このプロトコルの例では，B から A への認証要件は「B がセッション鍵 $\mathbf{hash}(N_A)$ を受理したならば，それよりも前の段階で A は B とナンス N_A を用いた鍵交換を行おうとしている」という対応関係を見ることにより，検証を実現します．ProVerif では，

```
1     event(e1) ==> event(e2)
```

と書くことで,「イベント e_1 が起きているならば，それよりも前にイベント e_2 が起きている」という現象を検証できます．

演習 3.3. コード3の ??? 部分を埋めて，プロトコル P'' のセッション鍵の機密性と B から A への認証要件を検証するコードを完成させてください．

Code 3: 二者間鍵交換プロトコル P''

```
1 (* Setting of channels *)
2 free c:channel.
3
4 (* Hash function *)
5 fun hash(bitstring):bitstring.
6
7 (* Symmetric key encryption *)
8 fun senc(bitstring, bitstring):bitstring.
9 reduc forall m:bitstring, k: bitstring; sdec(senc(m, k), k) = m.
10
11 (* Asymmetric key encryption *)
12 fun pk(bitstring):bitstring.
13 fun aenc(bitstring, bitstring):bitstring.
14 reduc forall m: bitstring, key:bitstring; adec(aenc(m, pk(key)), key) = m.
15
16 (* Signatures *)
17 fun spk(bitstring):bitstring.
18 fun sign(bitstring, bitstring):bitstring.
19 reduc forall m:bitstring, key:bitstring; check(sign(m, key), spk(key)) = m.
20
21 (* Queries and Events *)
22 free message_for_verification_A:bitstring[private].
23 query attacker(message_for_verification_A).
24 free message_for_verification_B:bitstring[private].
25 query attacker(message_for_verification_B).
26
27 event Accept_B(bitstring).
28 event Send_A(bitstring).
29 query x:bitstring; event(Accept_B(x)) ==> event(Send_A(x)).
30
31 (* Behavior of Alice *)
32 let Alice(pkA:bitstring, pkB:bitstring, spkA:bitstring, spkB:bitstring, skA:
       bitstring, sskA:bitstring) =
33     new Na: bitstring;
34     event Send_A(hash(Na));
35     let Cipher_A = aenc(Na, pkB) in
36     let Sig_A = ???(3) in
37     out(c, (Cipher_A, Sig_A));
38     let sk_Alice = hash(Na) in
39     out(c, senc(message_for_verification_A, sk_Alice)).
40
41 (* Behavior of Bob *)
42 let Bob(pkB:bitstring, pkA:bitstring, spkB:bitstring, spkA:bitstring, skB:
       bitstring, sskB:bitstring) =
43     in(c, (Cipher_A_recv:bitstring, Sig_A_recv:bitstring));
44     if ???(4) = ???(5) then
45     let Na = adec(Cipher_A_recv, skB) in
46     let sk_Bob = hash(Na) in
47     event Accept(sk_Bob);
48     out(c, senc(message_for_verification_B, sk_Bob)).
49
50 (* Start process *)
51 process new skA:bitstring; let pkA = pk(skA) in out(c, pkA);
52     new skB:bitstring; let pkB = pk(skB) in out(c, pkB);
53     new sskA:bitstring; let spkA = spk(sskA) in out(c, spkA);
54     new sskB:bitstring; let spkB = spk(sskB) in out(c, spkB);
55 ((!Alice(pkA,pkB,spkA,spkB,skA,sskA)) | !Bob(pkB,pkA,spkB,spkA,skB,sskB))
```

新しく定義された変数名は次の通りです．

- A の署名検証鍵：spkA，A の署名鍵：sskA（A のサブプロセスの中でのみ使えます）．
- B の署名検証鍵：spkB，B の署名鍵：sskB（B のサブプロセスの中でのみ使えます）．

こちらも参考解答は code3_complete.pv をご覧ください [4]．code3_complete.pv の出力は次のようになります.

- `Query not attacker(message_for_verification_A[]) is true.`
- `Query not attacker(message_for_verification_B[]) is true.`
- `Query event(Accept_B(x)) ==> event(Send_A(x)) is true.`

A と B のセッション鍵の機密性および B から A への認証性について `true` が出力されることが確認できました.

3.4 クエリの種類

補足として，ProVerif の扱える安全性要件について説明します．扱える要件は次の 3 つの性質です.

- 到達可能性（reachability）．ある特定のイベントがプロセスの組み合わせによって発生しうるかどうかを検証します．秘密情報の機密性やプロトコルが最後まで到達可能かどうかなどの要件を検証できます.
- 対応表明性（correspondence assertions）．あるイベント e が起きたときに，前提となるイベント e' が必ず発生しているかどうかを検証します．認証要件などを検証できます.
- 観測等価性（observational equivalence）．2 つのプロセスを実行した結果が外部から見て識別可能かどうかを検証します．プライバシーの要件などを検証できます.

上の例では，機密性の検証を到達可能性のクエリを用いて，認証要件の検証を対応表明性のクエリを用いて行いました.

4 発展演習

お時間がある方はどうぞ.

演習 4.1. プロトコル P' を電子署名を用いずにセッション鍵の機密性を満たす（攻撃が発見されない）ように修正し，検証してみましょう．ただし，セッション鍵は $\mathbf{hash}(N_A)$ とします.

演習 4.2. codeDH.pv の???部分を埋めて DH 鍵交換プロトコルを検証するコードを作成し，どのような攻撃が発見されるか見てみましょう.

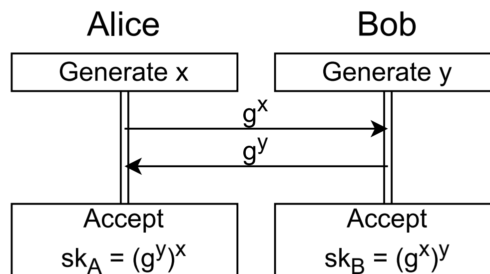

図 9: DH 鍵交換プロトコル.

[4]（小噺その 3）P'' もワンパスのシンプルなプロトコルなため，ProVerif のコードは 53 行ほどです．複雑なプロトコルになるとどのくらいになるのでしょうか．例えば TLS 1.3 の検証 [KBB17] の github ディレクトリを見てみると，コードの行数（最大）は 722 行のようです．なお，IETF の TLS1.3 の文書 [Res18] は 160 ページです.

5 おわりに

本ハンズオンでは，簡単な（認証付き）鍵交換プロトコルを例に，ProVerif による検証方法について紹介しました．時間の都合上，細かい文法や記述方法については扱いませんでした．まだまだ ProVerif の能力の 1/10 も伝えられていないため，興味を持っていただけた方はマニュアルや既存の検証例などを見てもらえればと思います．ハンズオンにご参加いただきありがとうございました．

References

[Bla21] Bruno Blanchet. Proverif: Cryptographic protocol verifier in the formal model, 2021. https://prosecco.gforge.inria.fr/personal/bblanche/proverif/. Accessed: 2023-10-27.

[BMS19] Colin Boyd, Anish Mathuria, and Douglas Stebila. *Protocols for authentication and key establishment Second Edition*. Springer, 2019.

[BSCS21] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. Proverif 2.04: Automatic cryptographic protocol verifier, user manual and tutorial, 2021.

[DY83] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[KBB17] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach. In *2017 IEEE European symposium on security and privacy (EuroS&P)*, pages 435–450. IEEE, 2017.

[MB21] Sreekanth Malladi and Bruno Blanchet. Online demo for proverif, 2021. http://proverif20.paris.inria.fr/index.php. Accessed: 2023-10-27.

[Res18] Eric Rescorla. Rfc 8446: The transport layer security (tls) protocol version 1.3, 2018.

ⒸFWS 準備委員会