

# On Deniable Authentication against Malicious Verifiers

Rune Fiedler



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Roman Langrehr

Work done at:

**ETH** zürich

Now at:



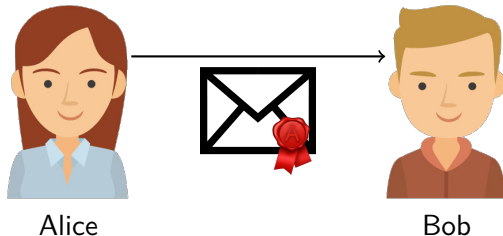
UNIVERSITY OF  
**WATERLOO**

2025-08-06



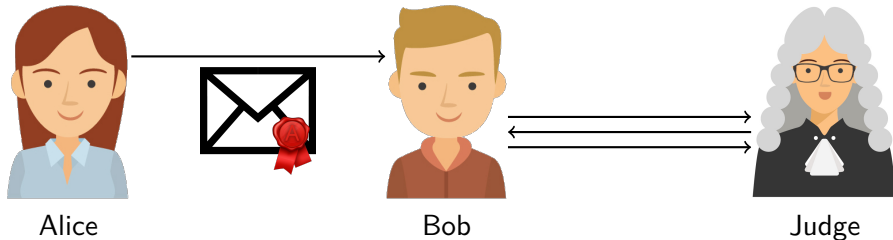


# Deniable authentication [DDN91, DNS98, DG05]



- Bob is convinced by the authenticity of Alice's message.

# Deniable authentication [DDN91, DNS98, DG05]



- Bob is convinced by the authenticity of Alice's message.
- Bob cannot convince anyone else of the authenticity of Alice's message.

# Designated verifier signatures (DVS) [JSI96]



Alice  
 $(pk_A, sk_A)$



Bob  
 $(pk_B, sk_B)$

# Designated verifier signatures (DVS) [JSI96]



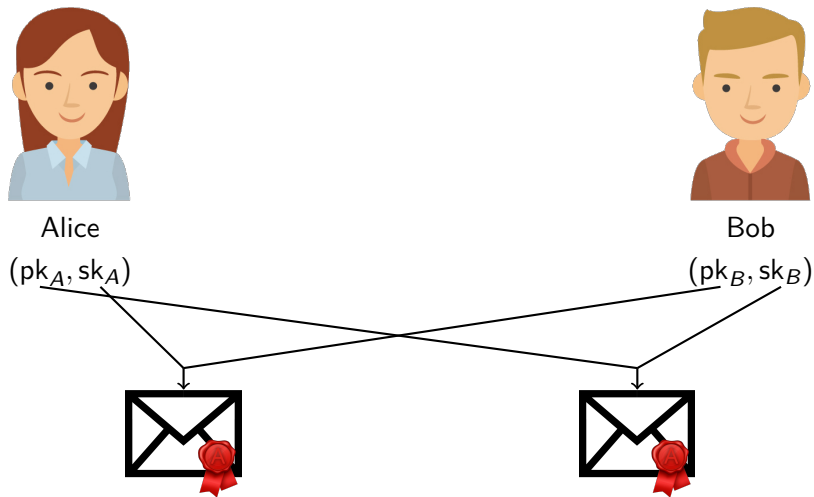
Alice  
 $(pk_A, sk_A)$



Bob  
 $(pk_B, sk_B)$

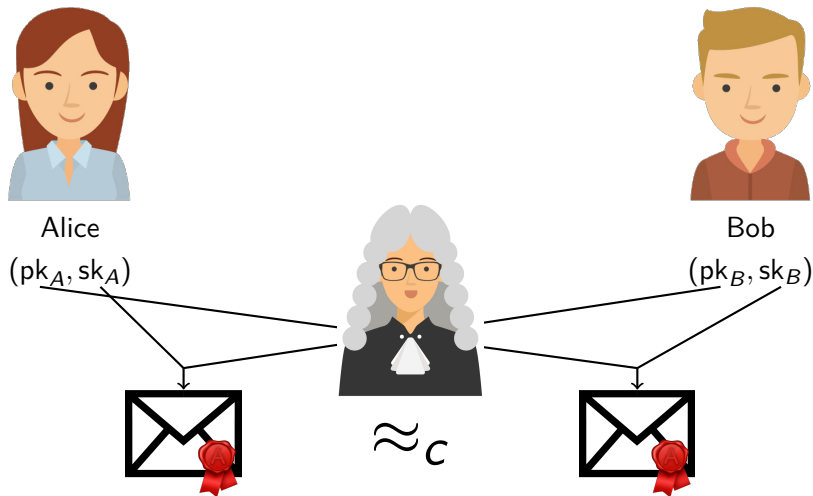


# Designated verifier signatures (DVS) [JSI96]



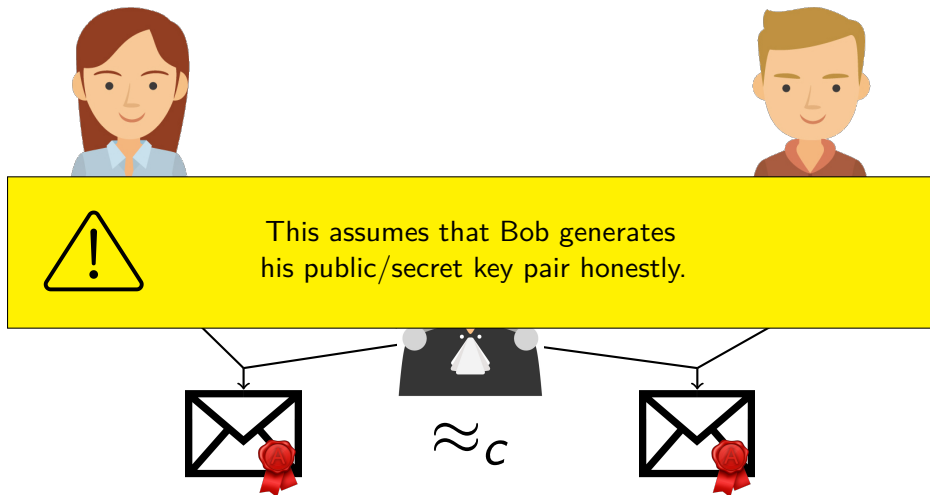


# Designated verifier signatures (DVS) [JSI96]



Alice and Bob can produce identically looking signatures.

# Designated verifier signatures (DVS) [JSI96]



Alice and Bob can produce identically looking signatures.



# Our contributions

- 1 A model for deniable authentication against malicious verifiers for DVS

# Our contributions

- ① A model for deniable authentication against malicious verifiers for DVS
- ② Undeniability of Signal's initial handshake protocols (X3DH and PQXDH) in a similar model for key exchange 📜

# Our contributions

- ① A model for deniable authentication against malicious verifiers for DVS
- ② Undeniability of Signal's initial handshake protocols (X3DH and PQXDH) in a similar model for key exchange 📜
  - Break (Extended) Knowledge of Diffie–Hellman (E)KDH assumption 📜

# Our contributions

- ① A model for deniable authentication against malicious verifiers for DVS
- ② Undeniability of Signal's initial handshake protocols (X3DH and PQXDH) in a similar model for key exchange 📜
  - Break (Extended) Knowledge of Diffie–Hellman (E)KDH assumption 📜
- ③ A construction of DVS in the ROM based on NIZKs

# Our contributions

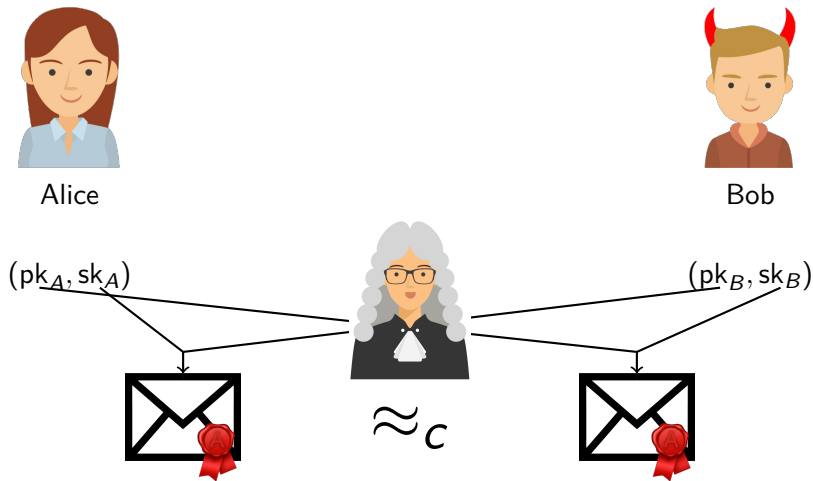
- ① A model for deniable authentication against malicious verifiers for DVS
- ② Undeniability of Signal's initial handshake protocols (X3DH and PQXDH) in a similar model for key exchange 📜
  - Break (Extended) Knowledge of Diffie–Hellman (E)KDH assumption 📜
- ③ A construction of DVS in the ROM based on NIZKs
- ④ An impossibility for DVS in the standard model assuming indistinguishability obfuscation (iO) exists (concrete attack)



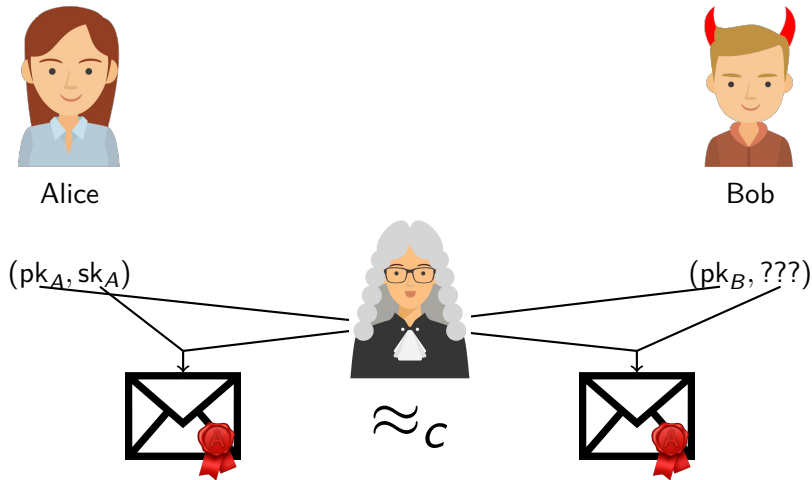
# Our contributions

- ① A model for deniable authentication against malicious verifiers for DVS
- ② Undeniability of Signal's initial handshake protocols (X3DH and PQXDH) in a similar model for key exchange 📜
  - Break (Extended) Knowledge of Diffie–Hellman (E)KDH assumption 📜
- ③ A construction of DVS in the ROM based on NIZKs
- ④ An impossibility for DVS in the standard model assuming indistinguishability obfuscation (iO) exists (concrete attack)
  - Shows uninstantiability of the ROM construction

# Modelling deniability against malicious verifiers

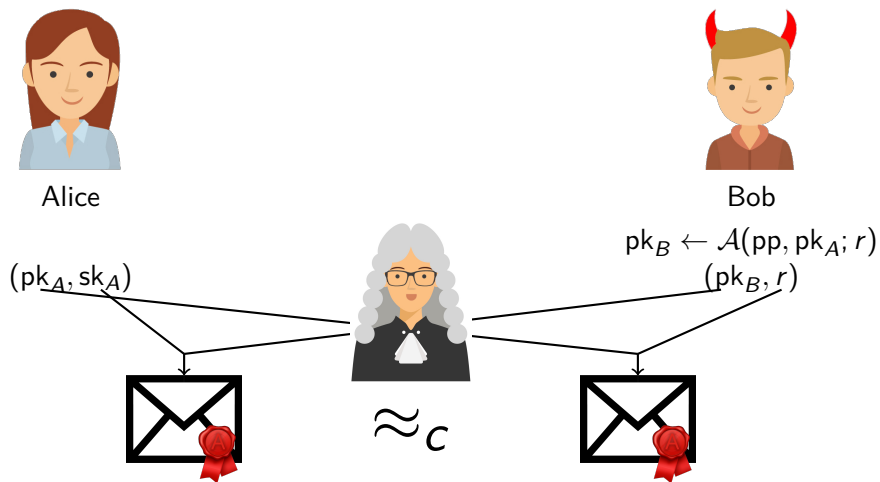


# Modelling deniability against malicious verifiers



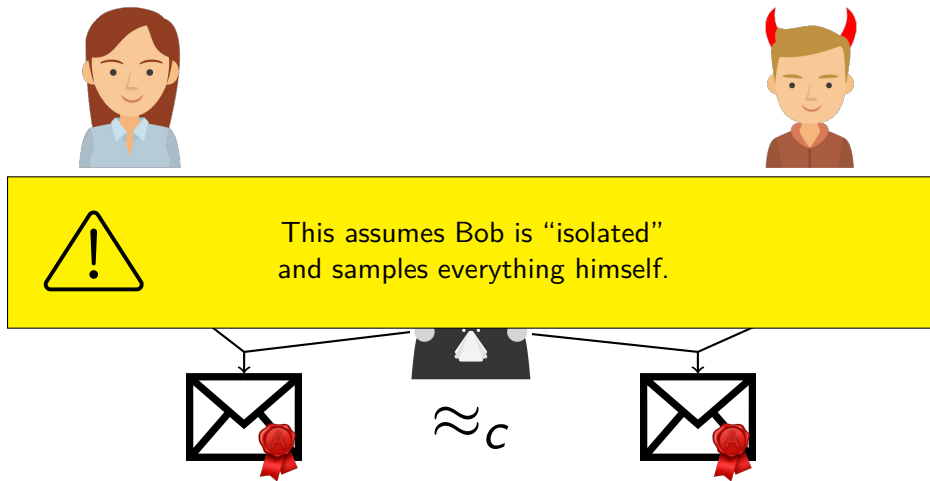
Bob can produce signatures using a secret he is guaranteed to possess.

# Modelling deniability against malicious verifiers: First attempt



Bob can produce signatures using a the random coins used to generate his public key.  
We make no assumption about how Bob generates his public key.

# Modelling deniability against malicious verifiers: First attempt



Bob can produce signatures using a the random coins used to generate his public key.  
We make no assumption about how Bob generates his public key.

# Is this realistic?

CD 83 40 FB 12 2F 79 6C 22 BE E9 20 0D 18 03 72 68 03 C6 40 49 2B 33 75 2B 47 0D FF D2 BA 2A 71 0D 24 3A AD C5 22 F1 B7 D7 0E B1 29 1A  
9E 90 F8 8D 43 96 AA C4 A9 B3 BF 5D 55 25 B3 33 5D 3B D5 24 64 4E DD CA 82 7C 55 9A B1 C8 66 20 70 E7 9A FD 80 C6 34 61 2D CB E3 5A C2  
36 C4 F7 71 E8 C2 07 77 01 C6 02 F3 A9 07 75 B2 75 31 14 A0 7D F0 6B F1 50 A0 73 5F B3 CC AA 99 CC F3 68 74 6E 03 D7 38 B5 FF DC A6 A9  
9C 16 74 B9 9C FA 41 CD 1F 8E 1C 2C 49 6A 8E CC 88 76 CA 0A 5A D5 C5 ED E9 6E AE DB 8C 97 B9 76 39 3A E1 EA E9 C7 81 17 3E F8 41 FB 4C  
D1 BB 53 C2 A0 70 17 95 8F DB D8 6F 2F F6 56 7A AA 3D 70 1D 07 ED 99 98 88 35 30 4C 6A 6B 4A 2A 11 84 01 BF 61 AA A5 C1 20 FF CD F4 26  
0E EE 3F A3 B7 65 DC FE 89 35 9C 99 E5 7A 2F 9F D9 1D A6 B8 A6 09 2F D0 F3 B5 F0 A0 B8 15 D2 51 DC 8D 63 B8 2E BF D1 E9 67 DB 2A 2C 9B  
95 DC D7 3C ED BE D2 87 53 4A 11 1D 99 2D 2B D1 E4 7E 18 7D B0 D1 70 9B A8 2B D5 93 74 F5 E4 7D AE 26 AB 75 C0 AA B6 2E 4E 15 80 29 2A  
36 B3 A7 3D D6 07 6E AF A2 25 62 77 6B B5 EC A8 B5 76 70 BB E7 BA DE 37 91 89 83 05 DD DB DE 9D 12 35 1C 4C AC 96 06 92 A2 04 2A 6B DA  
E7 1D 58 19 BD 5C 88 8A CE 56 D0 F6 8F 91 38 C4 70 8F 47 A0 5E 85 08 14 80 9E ED 32 48 0C 1D C0 BC AB 1C 79 CB E5 AC B4 65 01 B6 BB E0  
F5 CB 48 37 3A C9 F6 8C CF 87 06 46 F8 AC 54 2D 47 37 9F 1B 84 B3 C9 00 FA 89 72 7D 04 0F C2 A3 3D 02 AF 22 8F 6F 4F EE BF 18 58 D9 E5  
15 B7 56 FB E3 C5 BF 83 6E DC 05 97 52 F8 38 F0 7E E5 41 4F 50 5C 6E DA 98 BF CB A3 F6 75 E3 F8 DD 0C 2A 0B 64 47 5C 73 34 42 A5 B3 A9  
FB 41 C1 87 DE 1B 9E 25 6D 2A 45 66 05 8F 2B FA 75

- This is the product of two 2048 bit primes.

# Is this realistic?

CD 83 40 FB 12 2F 79 6C 22 BE E9 20 0D 18 03 72 68 03 C6 40 49 2B 33 75 2B 47 0D FF D2 BA 2A 71 0D 24 3A AD C5 22 F1 B7 D7 0E B1 29 1A  
9E 90 F8 8D 43 96 AA C4 A9 B3 BF 5D 55 25 B3 33 5D 3B D5 24 64 4E DD CA 82 7C 55 9A B1 C8 66 20 70 E7 9A FD 80 C6 34 61 2D CB E3 5A C2  
36 C4 F7 71 E8 C2 07 77 01 C6 02 F3 A9 07 75 B2 75 31 14 A0 7D F0 6B F1 50 A0 73 5F B3 CC AA 99 CC F3 68 74 6E 03 D7 38 B5 FF DC A6 A9  
9C 16 74 B9 9C FA 41 CD 1F 8E 1C 2C 49 6A 8E CC 88 76 CA 0A 5A D5 C5 ED E9 6E AE DB 8C 97 B9 76 39 3A E1 EA E9 C7 81 17 3E F8 41 FB 4C  
D1 BB 53 C2 A0 70 17 95 8F DB D8 6F 2F F6 56 7A AA 3D 70 1D 07 ED 99 98 88 35 30 4C 6A 6B 4A 2A 11 84 01 BF 61 AA A5 C1 20 FF CD F4 26  
0E EE 3F A3 B7 65 DC FE 89 35 9C 99 E5 7A 2F 9F D9 1D A6 B8 A6 09 2F D0 F3 B5 F0 A0 B8 15 D2 51 DC 8D 63 B8 2E BF D1 E9 67 DB 2A 2C 9B  
95 DC D7 3C ED BE D2 87 53 4A 11 1D 99 2D 2B D1 E4 7E 18 7D B0 D1 70 9B A8 2B D5 93 74 F5 E4 7D AE 26 AB 75 C0 AA B6 2E 4E 15 80 29 2A  
36 B3 A7 3D D6 07 6E AF A2 25 62 77 6B B5 EC A8 B5 76 70 BB E7 BA DE 37 91 89 83 05 DD DB DE 9D 12 35 1C 4C AC 96 06 92 A2 04 2A 6B DA  
E7 1D 58 19 BD 5C 88 8A CE 56 D0 F6 8F 91 38 C4 70 8F 47 A0 5E 85 08 14 80 9E ED 32 48 0C 1D C0 BC AB 1C 79 CB E5 AC B4 65 01 B6 BB E0  
F5 CB 48 37 3A C9 F6 8C CF 87 06 46 F8 AC 54 2D 47 37 9F 1B 84 B3 C9 00 FA 89 72 7D 04 0F C2 A3 3D 02 AF 22 8F 6F 4F EE BF 18 58 D9 E5  
15 B7 56 FB E3 C5 BF 83 6E DC 05 97 52 F8 38 F0 7E E5 41 4F 50 5C 6E DA 98 BF CB A3 F6 75 E3 F8 DD 0C 2A 0B 64 47 5C 73 34 42 A5 B3 A9  
FB 41 C1 87 DE 1B 9E 25 6D 2A 45 66 05 8F 2B FA 75

- This is the product of two 2048 bit primes.
- There is no algorithm to sample such a number without knowing the primes (as far as I know)

# Is this realistic?

CD 83 40 FB 12 2F 79 6C 22 BE E9 20 0D 18 03 72 68 03 C6 40 49 2B 33 75 2B 47 0D FF D2 BA 2A 71 0D 24 3A AD C5 22 F1 B7 D7 0E B1 29 1A  
9E 90 F8 8D 43 96 AA C4 A9 B3 BF 5D 55 25 B3 33 5D 3B D5 24 64 4E DD CA 82 7C 55 9A B1 C8 66 20 70 E7 9A FD 80 C6 34 61 2D CB E3 5A C2  
36 C4 F7 71 E8 C2 07 77 01 C6 02 F3 A9 07 75 B2 75 31 14 A0 7D F0 6B F1 50 A0 73 5F B3 CC AA 99 CC F3 68 74 6E 03 D7 38 B5 FF DC A6 A9  
9C 16 74 B9 9C FA 41 CD 1F 8E 1C 2C 49 6A 8E CC 88 76 CA 0A 5A D5 C5 ED E9 6E AE DB 8C 97 B9 76 39 3A E1 EA E9 C7 81 17 3E F8 41 FB 4C  
D1 BB 53 C2 A0 70 17 95 8F DB D8 6F 2F F6 56 7A AA 3D 70 1D 07 ED 99 98 88 35 30 4C 6A 6B 4A 2A 11 84 01 BF 61 AA A5 C1 20 FF CD F4 26  
0E EE 3F A3 B7 65 DC FE 89 35 9C 99 E5 7A 2F 9F D9 1D A6 B8 A6 09 2F D0 F3 B5 F0 A0 B8 15 D2 51 DC 8D 63 B8 2E BF D1 E9 67 DB 2A 2C 9B  
95 DC D7 3C ED BE D2 87 53 4A 11 1D 99 2D 2B D1 E4 7E 18 7D B0 D1 70 9B A8 2B D5 93 74 F5 E4 7D AE 26 AB 75 C0 AA B6 2E 4E 15 80 29 2A  
36 B3 A7 3D D6 07 6E AF A2 25 62 77 6B B5 EC A8 B5 76 70 BB E7 BA DE 37 91 89 83 05 DD DB DE 9D 12 35 1C 4C AC 96 06 92 A2 04 2A 6B DA  
E7 1D 58 19 BD 5C 88 8A CE 56 D0 F6 8F 91 38 C4 70 8F 47 A0 5E 85 08 14 80 9E ED 32 48 0C 1D C0 BC AB 1C 79 CB E5 AC B4 65 01 B6 BB E0  
F5 CB 48 37 3A C9 F6 8C CF 87 06 46 F8 AC 54 2D 47 37 9F 1B 84 B3 C9 00 FA 89 72 7D 04 0F C2 A3 3D 02 AF 22 8F 6F 4F EE BF 18 58 D9 E5  
15 B7 56 FB E3 C5 BF 83 6E DC 05 97 52 F8 38 F0 7E E5 41 4F 50 5C 6E DA 98 BF CB A3 F6 75 E3 F8 DD 0C 2A 0B 64 47 5C 73 34 42 A5 B3 A9  
FB 41 C1 87 DE 1B 9E 25 6D 2A 45 66 05 8F 2B FA 75

- This is the product of two 2048 bit primes.
- There is no algorithm to sample such a number without knowing the primes (as far as I know)
- But I don't know the primes...



# Is this realistic?

```
CD 83 40 FB 12 2F 79 6C 22 BE E9 20 0D 18 03 72 68 03 C6 40 49 2B 33 75 2B 47 0D FF D2 BA 2A 71 0D 24 3A AD C5 22 F1 B7 D7 0E B1 29 1A
9E 90 F8 8D 43 96 AA C4 A9 B3 BF 5D 55 25 B3 33 5D 3B D5 24 64 4E DD CA 82 7C 55 9A B1 C8 66 20 70 E7 9A FD 80 C6 34 61 2D CB E3 5A C2
36 C4 F7 71 E8 C2 07 77 01 C6 02 F3 A9 07 75 B2 75 31 14 A0 7D F0 6B F1 50 A0 73 5F B3 CC AA 99 CC F3 68 74 6E 03 D7 38 B5 FF DC A6 A9
9C 16 74 B9 9C FA 41 CD 1F 8E 1C 2C 49 6A 8E CC 88 76 CA 0A 5A D5 C5 ED E9 6E AE DB 8C 97 B9 76 39 3A E1 EA E9 C7 81 17 3E F8 41 FB 4C
D1 BB 53 C2 A0 70 17 95 8F DB D8 6F 2F F6 56 7A AA 3D 70 1D 07 ED 99 98 88 35 30 4C 6A 6B 4A 2A 11 84 01 BF 61 AA A5 C1 20 FF CD F4 26
0E EE 3F A3 B7 65 DC FE 89 35 9C 99 E5 7A 2F 9F D9 1D A6 B8 A6 09 2F D0 F3 B5 F0 A0 B8 15 D2 51 DC 8D 63 B8 2E BF D1 E9 67 DB 2A 2C 9B
95 DC D7 3C ED BE D2 87 53 4A 11 1D 99 2D 2B D1 E4 7E 18 7D B0 D1 70 9B A8 2B D5 93 74 F5 E4 7D AE 26 AB 75 C0 AA B6 2E 4E 15 80 29 2A
36 B3 A7 3D D6 07 6E AF A2 25 62 77 6B B5 EC A8 B5 76 70 BB E7 BA DE 37 91 89 83 05 DD DB DE 9D 12 35 1C 4C AC 96 06 92 A2 04 2A 6B DA
E7 1D 58 19 BD 5C 88 8A CE 56 D0 F6 8F 91 38 C4 70 8F 47 A0 5E 85 08 14 80 9E ED 32 48 0C 1D C0 BC AB 1C 79 CB E5 AC B4 65 01 B6 BB E0
F5 CB 48 37 3A C9 F6 8C CF 87 06 46 F8 AC 54 2D 47 37 9F 1B 84 B3 C9 00 FA 89 72 7D 04 0F C2 A3 3D 02 AF 22 8F 6F 4F EE BF 18 58 D9 E5
15 B7 56 FB E3 C5 BF 83 6E DC 05 97 52 F8 38 F0 7E E5 41 4F 50 5C 6E DA 98 BF CB A3 F6 75 E3 F8 DD 0C 2A 0B 64 47 5C 73 34 42 A5 B3 A9
FB 41 C1 87 DE 1B 9E 25 6D 2A 45 66 05 8F 2B FA 75
```

- This is the product of two 2048 bit primes.
- There is no algorithm to sample such a number without knowing the primes (as far as I know)
- But I don't know the primes...
- ...because it is the public key of the iacr.org webserver.

## Fixing the definition

We have to model everything Bob might know without generating it himself:

# Fixing the definition

We have to model everything Bob might know without generating it himself:

- Auxiliary input: Everything independent of the scheme's public parameters  $pp$ .

# Fixing the definition

We have to model everything Bob might know without generating it himself:

- Auxiliary input: Everything independent of the scheme's public parameters  $pp$ .
- Public keys of other verifiers 📜

# Fixing the definition

We have to model everything Bob might know without generating it himself:

- Auxiliary input: Everything independent of the scheme's public parameters  $pp$ .
- Public keys of other verifiers 📜
- Public keys of senders

# Fixing the definition

We have to model everything Bob might know without generating it himself:

- Auxiliary input: Everything independent of the scheme's public parameters  $pp$ .
- Public keys of other verifiers 📜
- Public keys of senders
- Valid signatures (for other messages or for other users)

# Fixing the definition

We have to model everything Bob might know without generating it himself:

- Auxiliary input: Everything independent of the scheme's public parameters  $pp$ .
- Public keys of other verifiers 📜
- Public keys of senders
- Valid signatures (for other messages or for other users)
- ...

# Fixing the definition

We have to model everything Bob might know without generating it himself:

- Auxiliary input: Everything independent of the scheme's public parameters  $pp$ .
- Public keys of other verifiers 📜
- Public keys of senders
- Valid signatures (for other messages or for other users)
- ...

These are given to Bob as input or oracle access.



# Fixing the definition

We have to model everything Bob might know without generating it himself:

- Auxiliary input: Everything independent of the scheme's public parameters  $pp$ .
  - Public keys of other verifiers 📄
  - Public keys of senders
  - Valid signatures (for other messages or for other users)
  - ...
- } Ignored here for simplicity.

These are given to Bob as input or oracle access.

# Malicious source hiding

A DVS is malicious source hiding iff

$\forall$  Adversaries  $\mathcal{A}$  (Bob, generates  $\text{pk}_B$ )

# Malicious source hiding

A DVS is malicious source hiding iff

$\forall$  Adversaries  $\mathcal{A}$  (Bob, generates  $\text{pk}_B$ )

$\exists$  Extractor  $\mathcal{E}$  (forges signatures, same inputs/random tape as Bob)

# Malicious source hiding

A DVS is malicious source hiding iff

- $\forall$  Adversaries  $\mathcal{A}$  (Bob, generates  $\text{pk}_B$ )
- $\exists$  Extractor  $\mathcal{E}$  (forges signatures, same inputs/random tape as Bob)
- $\forall$  Distinguisher  $\mathcal{D}$  (Judge)

# Malicious source hiding

A DVS is malicious source hiding iff

- $\forall$  Adversaries  $\mathcal{A}$  (Bob, generates  $\text{pk}_B$ )
- $\exists$  Extractor  $\mathcal{E}$  (forges signatures, same inputs/random tape as Bob)
- $\forall$  Distinguisher  $\mathcal{D}$  (Judge)
- $\forall \text{ aux} \in \{0, 1\}^*$  (given to  $\mathcal{A}$  and  $\mathcal{E}$ )

# Malicious source hiding

A DVS is malicious source hiding iff

$\forall$  Adversaries  $\mathcal{A}$  (Bob, generates  $pk_B$ )

$\exists$  Extractor  $\mathcal{E}$  (forges signatures, same inputs/random tape as Bob)

$\forall$  Distinguisher  $\mathcal{D}$  (Judge)

$\forall \text{ aux} \in \{0, 1\}^*$  (given to  $\mathcal{A}$  and  $\mathcal{E}$ )

$\mathcal{D}$  cannot distinguish real (Alice's) signatures  
from faked ( $\mathcal{E}$ 's) signatures.

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]



- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📜

# Non-solutions

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📜
- Use NIZKPoKs to prove knowledge of secret keys (black-box extraction)

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📜
- Use NIZKPoKs to prove knowledge of secret keys (black-box extraction)
  - Extractor would need to know a trapdoor for the crs

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📜
- Use NIZKPoKs to prove knowledge of secret keys (black-box extraction)
  - Extractor would need to know a trapdoor for the crs
  - In reality nobody knows this trapdoor

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📜
- Use NIZKPoKs to prove knowledge of secret keys (black-box extraction)
  - Extractor would need to know a trapdoor for the crs
  - In reality nobody knows this trapdoor
  - ⇒ Unrealistic to assume that Bob knows it

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📄
- Use NIZKPoKs to prove knowledge of secret keys (black-box extraction)
  - Extractor would need to know a trapdoor for the crs
  - In reality nobody knows this trapdoor
  - ⇒ Unrealistic to assume that Bob knows it
- Use NIZKPoKs to prove knowledge of secret keys (white-box extraction, used for SNARKs)

- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📄
- Use NIZKPoKs to prove knowledge of secret keys (black-box extraction)
  - Extractor would need to know a trapdoor for the crs
  - In reality nobody knows this trapdoor
  - ⇒ Unrealistic to assume that Bob knows it
- Use NIZKPoKs to prove knowledge of secret keys (white-box extraction, used for SNARKs)
  - Cannot be secure with auxiliary inputs assuming iO exists [BCPR14]



- Knowledge type assumptions (like knowledge of exponent assumption (KEA))
  - Most of them (like KEA) are not secure with auxiliary inputs assuming iO exists [BCPR14]
  - (E)KDH assumptions were designed to circumvent this problem...
  - ...but we show that they are insecure against very simple auxiliary inputs (no iO needed) 📌
- Use NIZKPoKs to prove knowledge of secret keys (black-box extraction)
  - Extractor would need to know a trapdoor for the crs
  - In reality nobody knows this trapdoor
  - ⇒ Unrealistic to assume that Bob knows it
- Use NIZKPoKs to prove knowledge of secret keys (white-box extraction, used for SNARKs)
  - Cannot be secure with auxiliary inputs assuming iO exists [BCPR14]

# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

```
crs  $\leftarrow$  GenNIZK()  
(pkB, skB)  $\leftarrow$  VKGen(pp; r) (Traditional DVS)  
 $\pi \leftarrow$  Prove(crs, pkB, skB)  
return ((pkB, crs,  $\pi$ ), skB)
```

# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

```
crs  $\leftarrow$  GenNIZK()  
(pkB, skB)  $\leftarrow$  VKGen(pp; r) (Traditional DVS)  
 $\pi \leftarrow$  Prove(crs, pkB, skB)  
return ((pkB, crs,  $\pi$ ), skB)
```

Problem: How can we force a malicious Bob to sample crs randomly?

# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

```
crs  $\leftarrow$  GenNIZK()  
(pkB, skB)  $\leftarrow$  VKGen(pp; r) (Traditional DVS)  
 $\pi \leftarrow$  Prove(crs, pkB, skB)  
return ((pkB, crs,  $\pi$ ), skB)
```

Problem: How can we force a malicious Bob to sample crs randomly?

Solution: Generate it via random oracle (requires random string as crs)

# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

$$\begin{aligned} \text{crs} &\leftarrow \text{Gen}_{\text{NIZK}}() \\ (\text{pk}_B, \text{sk}_B) &\leftarrow \text{VKGen}(\text{pp}; r) \\ \pi &\leftarrow \text{Prove}(\text{crs}, \text{pk}_B, \text{sk}_B) \\ \mathbf{return} &((\text{pk}_B, \text{crs}, \pi), \text{sk}_B) \end{aligned} \quad \text{(Traditional DVS)}$$

Problem: How can we force a malicious Bob to sample crs randomly?

Solution: Generate it via random oracle (requires random string as crs)

$$\begin{aligned} \text{crs} &\leftarrow H(\text{pp}, \text{id}) \\ \Rightarrow \mathcal{E} &\text{ can run } \mathcal{A} \text{ internally and program this query to a crs with known trapdoor} \end{aligned}$$

# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

$$\begin{aligned} \text{crs} &\leftarrow \text{Gen}_{\text{NIZK}}() \\ (\text{pk}_B, \text{sk}_B) &\leftarrow \text{VKGen}(\text{pp}; r) \\ \pi &\leftarrow \text{Prove}(\text{crs}, \text{pk}_B, \text{sk}_B) \\ \mathbf{return} &((\text{pk}_B, \text{crs}, \pi), \text{sk}_B) \end{aligned} \quad \text{(Traditional DVS)}$$

Problem: How can we force a malicious Bob to sample crs randomly?

Solution: Generate it via random oracle (requires random string as crs)

$$\text{crs} \leftarrow H(\text{pp}, \text{id})$$

$\Rightarrow \mathcal{E}$  can run  $\mathcal{A}$  internally and program this query to a crs with known trapdoor

Problem:  $\mathcal{A}$  could output a different public key when we change a ROM query

# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

$$\begin{aligned} \text{crs} &\leftarrow \text{Gen}_{\text{NIZK}}() \\ (\text{pk}_B, \text{sk}_B) &\leftarrow \text{VKGen}(\text{pp}; r) \\ \pi &\leftarrow \text{Prove}(\text{crs}, \text{pk}_B, \text{sk}_B) \\ \textbf{return } &((\text{pk}_B, \text{crs}, \pi), \text{sk}_B) \end{aligned} \quad \text{(Traditional DVS)}$$

Problem: How can we force a malicious Bob to sample crs randomly?

Solution: Generate it via random oracle (requires random string as crs)

$$\text{crs} \leftarrow H(\text{pp}, \text{id})$$

$\Rightarrow \mathcal{E}$  can run  $\mathcal{A}$  internally and program this query to a crs with known trapdoor

Problem:  $\mathcal{A}$  could output a different public key when we change a ROM query

Solution: Hash also the public key (Fiat–Shamir like trick):



# Construction in the ROM

Basic idea: If Bob generates the crs himself, he can know the trapdoor.

$$\begin{aligned} \text{crs} &\leftarrow \text{Gen}_{\text{NIZK}}() \\ (\text{pk}_B, \text{sk}_B) &\leftarrow \text{VKGen}(\text{pp}; r) \\ \pi &\leftarrow \text{Prove}(\text{crs}, \text{pk}_B, \text{sk}_B) \\ \textbf{return } &((\text{pk}_B, \text{crs}, \pi), \text{sk}_B) \end{aligned} \quad \text{(Traditional DVS)}$$

Problem: How can we force a malicious Bob to sample crs randomly?

Solution: Generate it via random oracle (requires random string as crs)

$$\text{crs} \leftarrow H(\text{pp}, \text{id})$$

$\Rightarrow \mathcal{E}$  can run  $\mathcal{A}$  internally and program this query to a crs with known trapdoor

Problem:  $\mathcal{A}$  could output a different public key when we change a ROM query

Solution: Hash also the public key (Fiat–Shamir like trick):

$$\begin{aligned} (\text{pk}_B, \text{sk}_B) &\leftarrow \text{VKGen}(\text{pp}; r) \\ \text{crs} &\leftarrow H(\text{pp}, \text{id}, \text{pk}_B) \\ \pi &\leftarrow \text{Prove}(\text{crs}, \text{pk}_B, \text{sk}_B) \\ \textbf{return } &((\text{pk}_B, \pi), \text{sk}_B) \end{aligned}$$

# Impossibility result (inspired by [BCPR14])

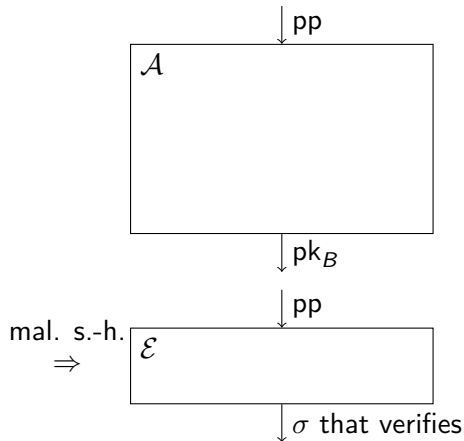
## Theorem

*Assuming  $iO$  exists, no DVS can be simultaneously malicious source-hiding and unforgeable.*

# Impossibility result (inspired by [BCPR14])

## Theorem

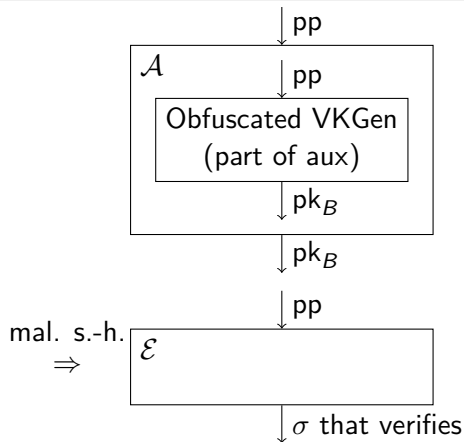
*Assuming  $iO$  exists, no DVS can be simultaneously malicious source-hiding and unforgeable.*



# Impossibility result (inspired by [BCPR14])

## Theorem

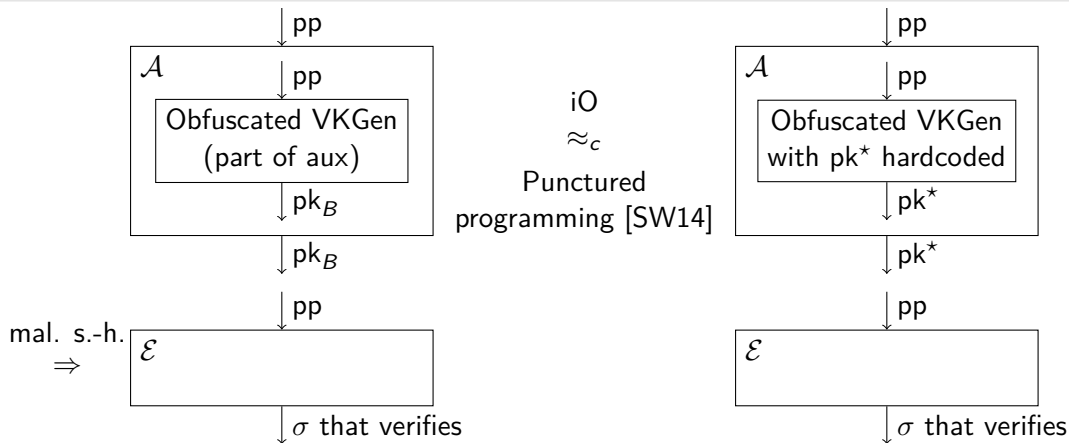
*Assuming  $iO$  exists, no DVS can be simultaneously malicious source-hiding and unforgeable.*



# Impossibility result (inspired by [BCPR14])

## Theorem

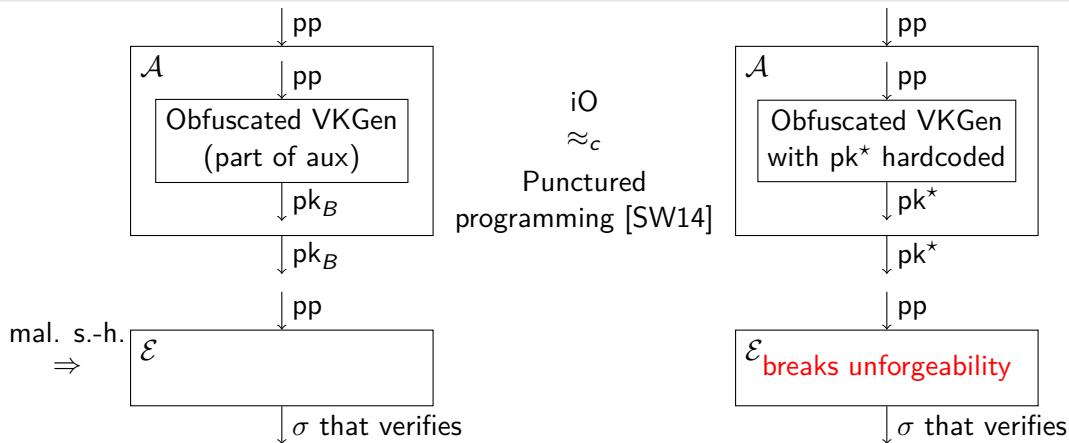
*Assuming  $iO$  exists, no DVS can be simultaneously malicious source-hiding and unforgeable.*



# Impossibility result (inspired by [BCPR14])

## Theorem

*Assuming  $iO$  exists, no DVS can be simultaneously malicious source-hiding and unforgeable.*



# ROM construction vs. impossibility result

- An obfuscated program cannot access the RO

# ROM construction vs. impossibility result

- An obfuscated program cannot access the RO
- In the ROM construction it is impossible to generate a valid verifier pk without access to RO.



# ROM construction vs. impossibility result

- An obfuscated program cannot access the RO
- In the ROM construction it is impossible to generate a valid verifier pk without access to RO.
- When we instantiate the RO with a concrete hash function, the impossibility result applies

# ROM construction vs. impossibility result

- An obfuscated program cannot access the RO
  - In the ROM construction it is impossible to generate a valid verifier pk without access to RO.
  - When we instantiate the RO with a concrete hash function, the impossibility result applies
- ⇒ The ROM construction is insecure when instantiated.

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient
  - Hard to use for E2EE messaging when users can be offline

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient
  - Hard to use for E2EE messaging when users can be offline
- Restricting auxiliary input

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient
  - Hard to use for E2EE messaging when users can be offline
- Restricting auxiliary input
  - Realistic restrictions (e.g. only efficiently samplable auxiliary input) do not circumvent our impossibility result

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient
  - Hard to use for E2EE messaging when users can be offline
- Restricting auxiliary input
  - Realistic restrictions (e.g. only efficiently samplable auxiliary input) do not circumvent our impossibility result
  - Some schemes require publishing an obfuscated circuit that suffices for our attack [HJK<sup>+</sup>16].



# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient
  - Hard to use for E2EE messaging when users can be offline
- Restricting auxiliary input
  - Realistic restrictions (e.g. only efficiently samplable auxiliary input) do not circumvent our impossibility result
  - Some schemes require publishing an obfuscated circuit that suffices for our attack [HJK<sup>+</sup>16].
  - For concrete schemes, iO might not be needed (e.g. X3DH/PQXDH/(E)KDH)

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient
  - Hard to use for E2EE messaging when users can be offline
- Restricting auxiliary input
  - Realistic restrictions (e.g. only efficiently samplable auxiliary input) do not circumvent our impossibility result
  - Some schemes require publishing an obfuscated circuit that suffices for our attack [HJK<sup>+</sup>16].
  - For concrete schemes, iO might not be needed (e.g. X3DH/PQXDH/(E)KDH)
- Private verification

# Future work: Circumventing the impossibility result

Possible approaches:

- Interactive protocols
  - Interactive proof of knowledge seems to be sufficient
  - Hard to use for E2EE messaging when users can be offline
- Restricting auxiliary input
  - Realistic restrictions (e.g. only efficiently samplable auxiliary input) do not circumvent our impossibility result
  - Some schemes require publishing an obfuscated circuit that suffices for our attack [HJK<sup>+</sup>16].
  - For concrete schemes, iO might not be needed (e.g. X3DH/PQXDH/(E)KDH)
- Private verification
  - Seems to be the most promising direction

## Future work: Different models for deniability against malicious verifiers

- E.g. not relying on the verifier's randomness for simulation

## Future work: Different models for deniability against malicious verifiers


- E.g. not relying on the verifier's randomness for simulation
- ... which is also not meaningful against quantum adversaries

## Future work: Different models for deniability against malicious verifiers

- E.g. not relying on the verifier's randomness for simulation
- ... which is also not meaningful against quantum adversaries
- E.g. time-deniable signatures have no problem with malicious verifiers



Protecting Alice in this scenario is difficult.

 ePrint: 2025/470

# References I



Nir Bitansky, Ran Canetti, Omer Paneth, and Alon Rosen.

On the existence of extractable one-way functions.

In David B. Shmoys, editor, 46th Annual ACM Symposium on Theory of Computing, pages 505–514, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

doi:10.1145/2591796.2591859.



Danny Dolev, Cynthia Dwork, and Moni Naor.

Non-malleable cryptography (extended abstract).

In 23rd Annual ACM Symposium on Theory of Computing, pages 542–552, New Orleans, LA, USA, May 6–8, 1991. ACM Press.

doi:10.1145/103418.103474.



# References II

 Mario Di Raimondo and Rosario Gennaro.

New approaches for deniable authentication.

In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, ACM CCS 2005: 12th Conference on Computer and Communications Security, pages 112–121, Alexandria, Virginia, USA, November 7–11, 2005. ACM Press.

doi:10.1145/1102120.1102137.

 Cynthia Dwork, Moni Naor, and Amit Sahai.

Concurrent zero-knowledge.

In 30th Annual ACM Symposium on Theory of Computing, pages 409–418, Dallas, TX, USA, May 23–26, 1998. ACM Press.

doi:10.1145/276698.276853.

# References III



Dennis Hofheinz, Tibor Jager, Dakshita Khurana, Amit Sahai, Brent Waters, and Mark Zhandry.

How to generate and use universal samplers.

In Jung Hee Cheon and Tsuyoshi Takagi, editors, Advances in Cryptology – ASIACRYPT 2016, Part II, volume 10032 of Lecture Notes in Computer Science, pages 715–744, Hanoi, Vietnam, December 4–8, 2016. Springer Berlin Heidelberg, Germany.  
doi:10.1007/978-3-662-53890-6\_24.



Markus Jakobsson, Kazue Sako, and Russell Impagliazzo.

Designated verifier proofs and their applications.

In Ueli M. Maurer, editor, Advances in Cryptology – EUROCRYPT'96, volume 1070 of Lecture Notes in Computer Science, pages 143–154, Saragossa, Spain, May 12–16, 1996. Springer Berlin Heidelberg, Germany.  
doi:10.1007/3-540-68339-9\_13.



Amit Sahai and Brent Waters.

How to use indistinguishability obfuscation: deniable encryption, and more.

In David B. Shmoys, editor, 46th Annual ACM Symposium on Theory of Computing, pages 475–484, New York, NY, USA, May 31 – June 3, 2014. ACM Press.

doi:10.1145/2591796.2591825.

# Pictures

Alice, Bob, judge, seal, warning sign, and papyrus scroll: freepik.com

Mail: Icon made by SimpleIcon from [www.flaticon.com](http://www.flaticon.com)

Scenes: AI generated (Microsoft copilot)

- Verifiers have unique identities

# Identities

- Verifiers have unique identities
- Generate  $pk/sk$  w.r.t. identity

# Identities

- Verifiers have unique identities
- Generate  $pk/sk$  w.r.t. identity
- Sign w.r.t. verifier's identity and public key

- Verifiers have unique identities
- Generate pk/sk w.r.t. identity
- Sign w.r.t. verifier's identity and public key
  - Sign can just return  $\perp$  if the public key doesn't match the identity.



- Verifiers have unique identities
- Generate pk/sk w.r.t. identity
- Sign w.r.t. verifier's identity and public key
  - Sign can just return  $\perp$  if the public key doesn't match the identity.
  - Avoids key copying attack.

# Identities

- Verifiers have unique identities
- Generate pk/sk w.r.t. identity
- Sign w.r.t. verifier's identity and public key
  - Sign can just return  $\perp$  if the public key doesn't match the identity.
  - Avoids key copying attack.

Identities can be avoided if we assume that public keys are unique (can be enforced by a PKI).