Leakage-Resilient Circuits Revisited

-- Optimal Number of Computing Components without Leak-free Hardware

Hong-Sheng Zhou

Virginia Commonwealth University

Joint work with

Dana Dachman-Soled and **Feng-Hao Liu**

University of Maryland


TCC Test-of-Time Award

The *TCC Test of Time Award* recognizes outstanding papers, published in TCC at least eight years ago, making a significant contribution to the influence also in other area of cryptography, theory, and beyond. The inaugural TCC Test of Time Award was given in TCC 2015 for papers pu

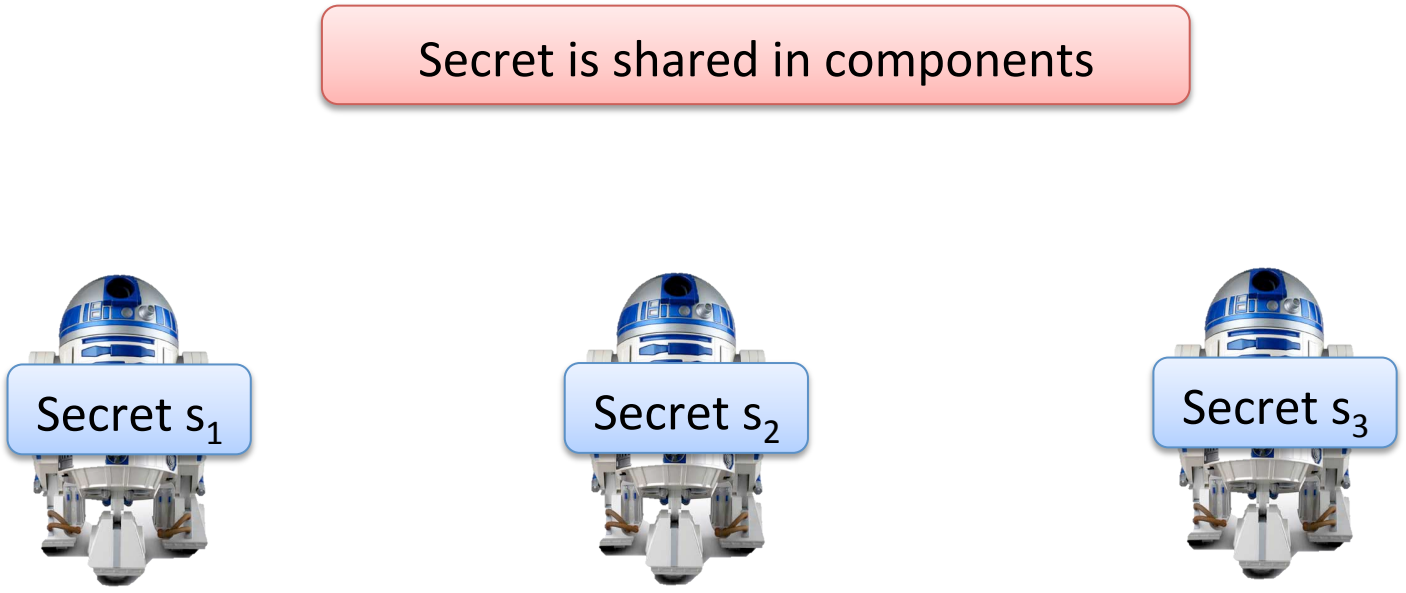
Award Recipients

2015:

- Physically Observable Cryptography by Silvio Micali (MIT) and Leonid Reyzin (Boston University), which was published in TCC 2004 award *for pioneering a mathematical foundation of cryptography in the presence of information leakage in physical systems*.

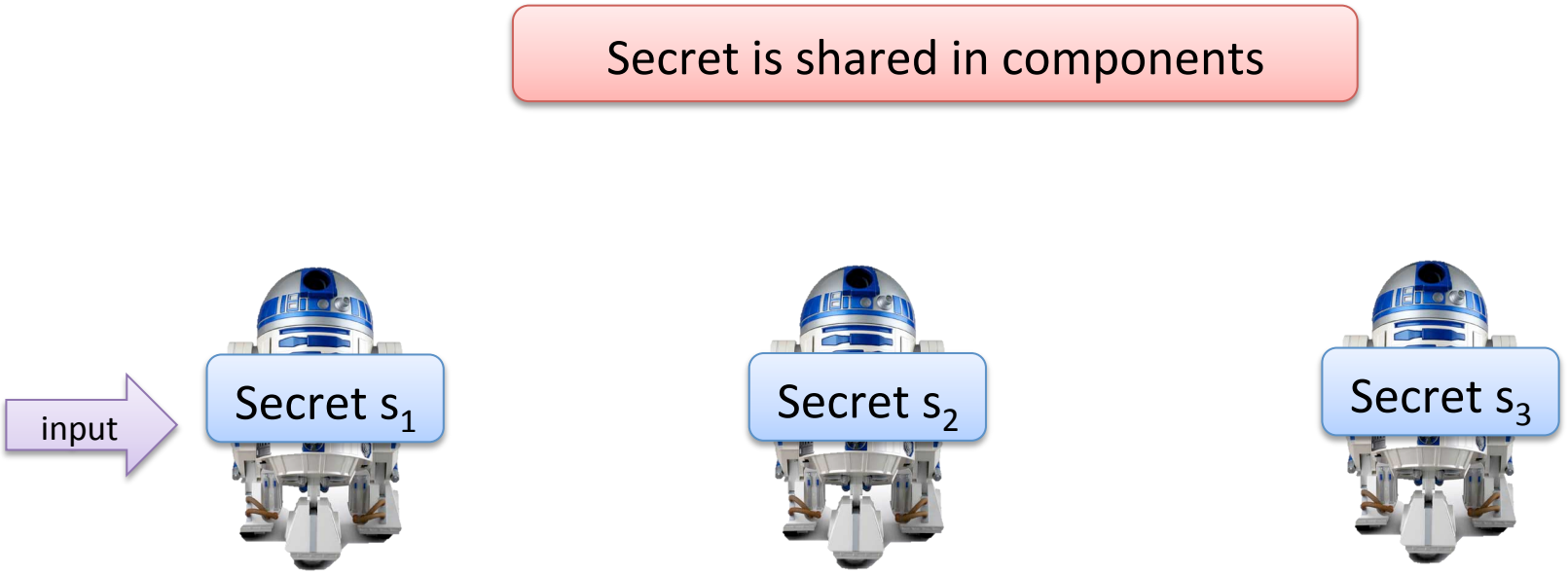
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



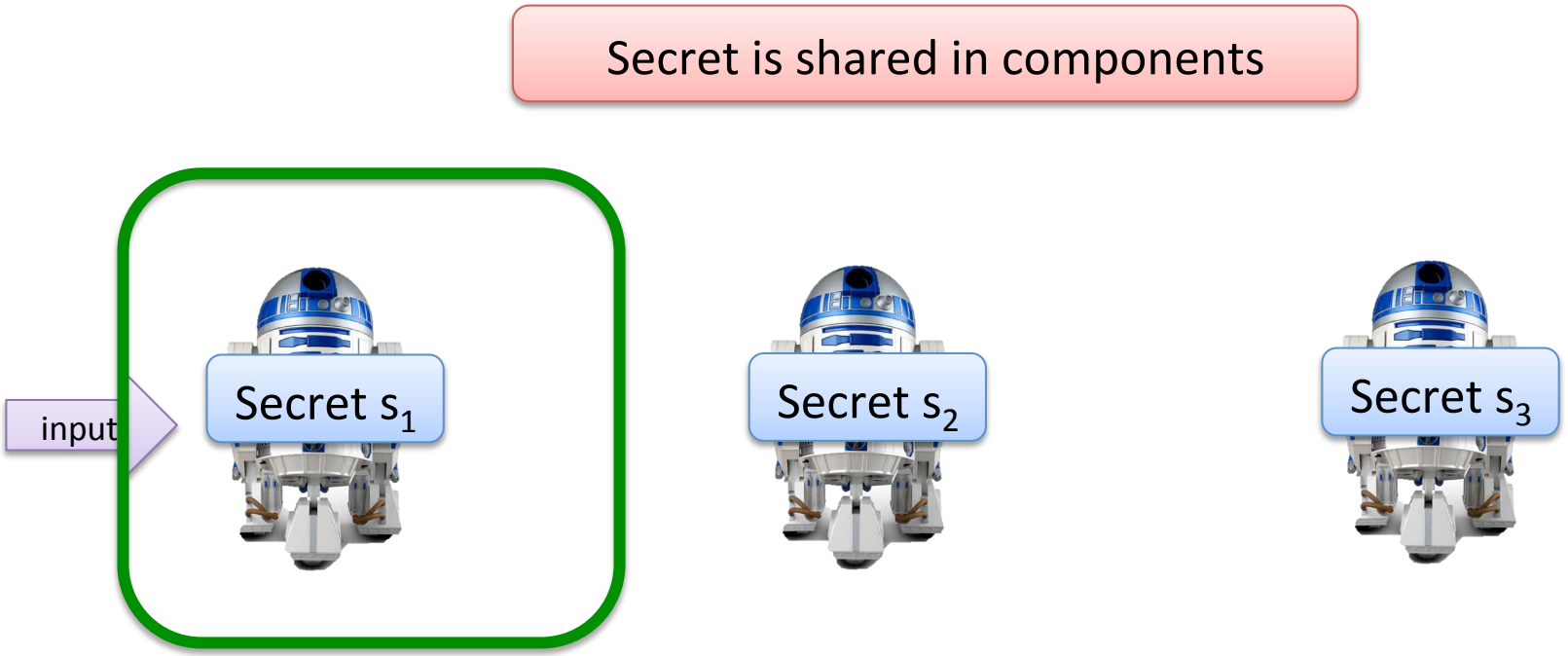
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



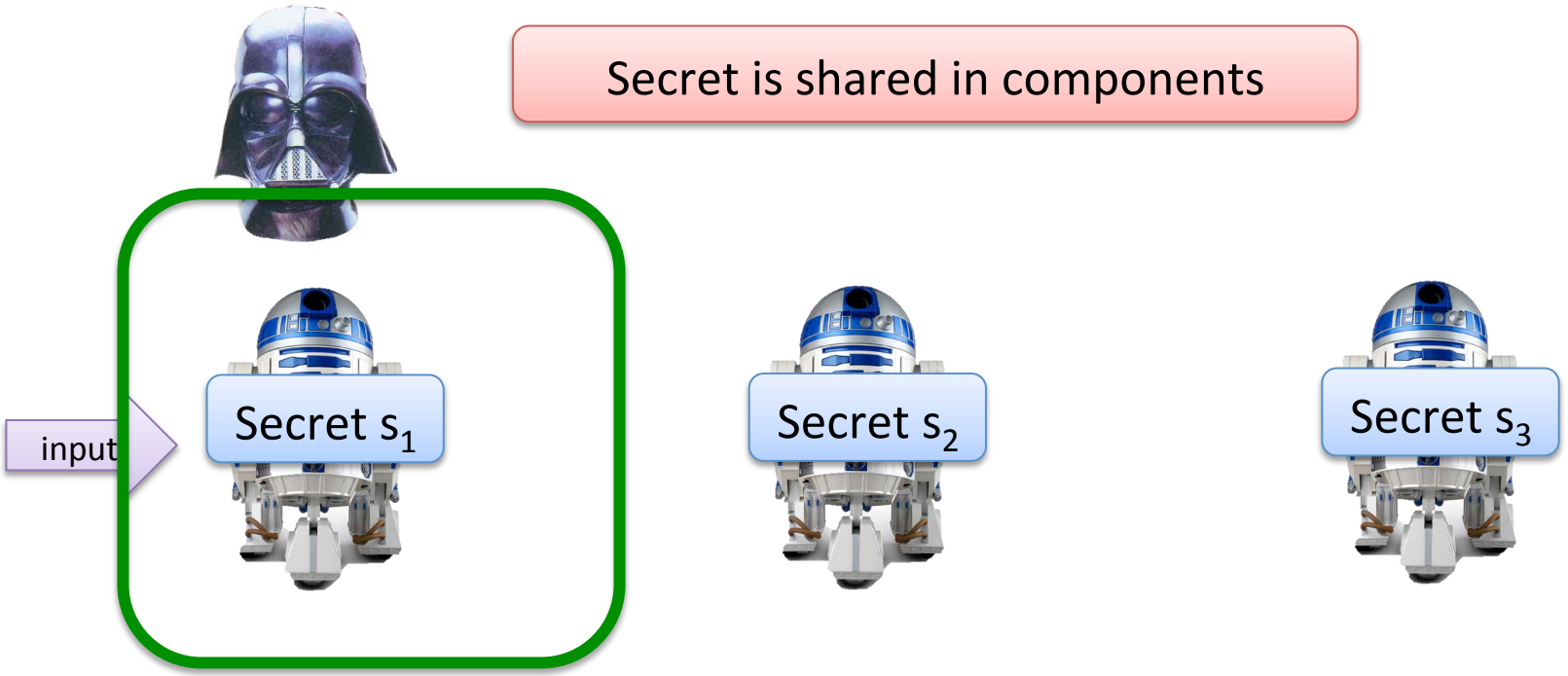
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



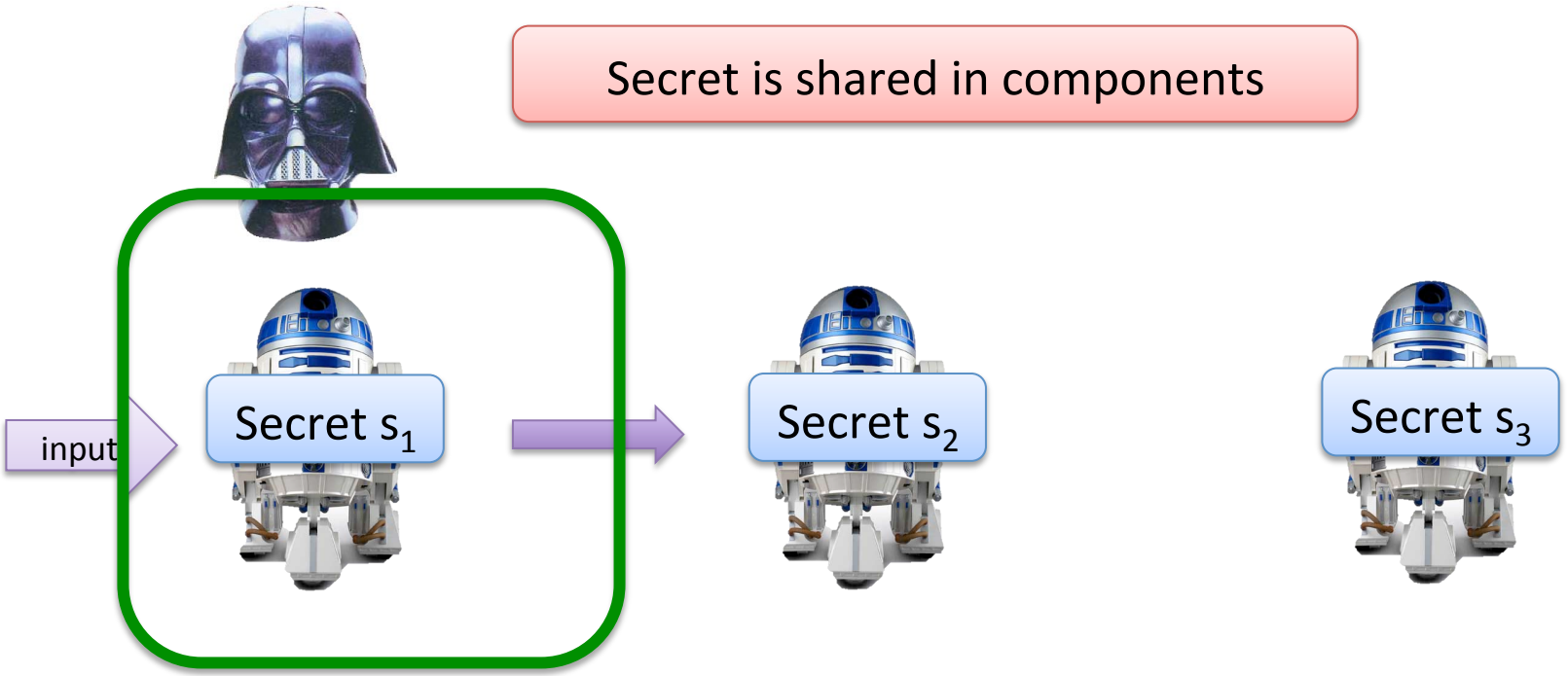
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



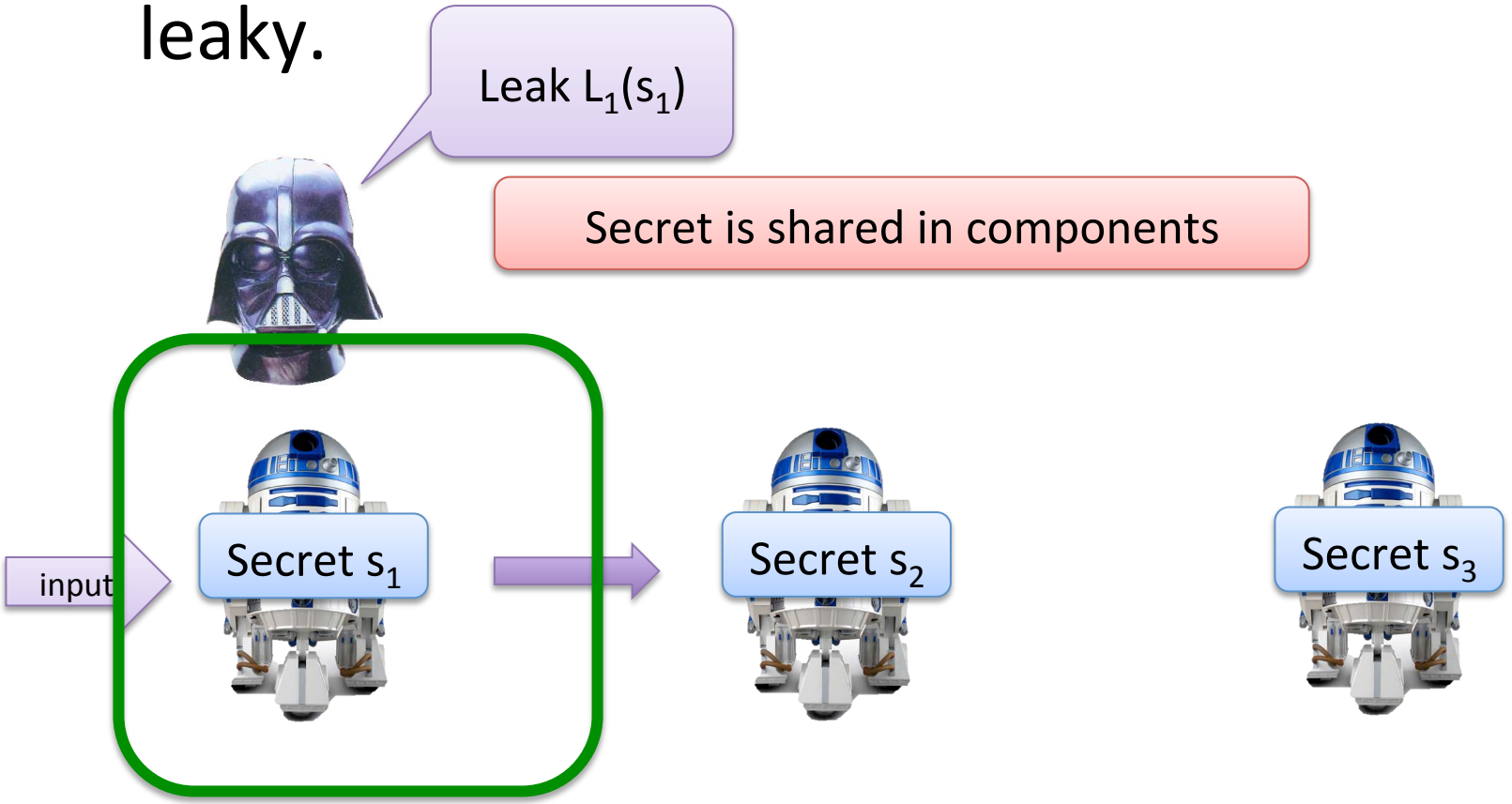
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



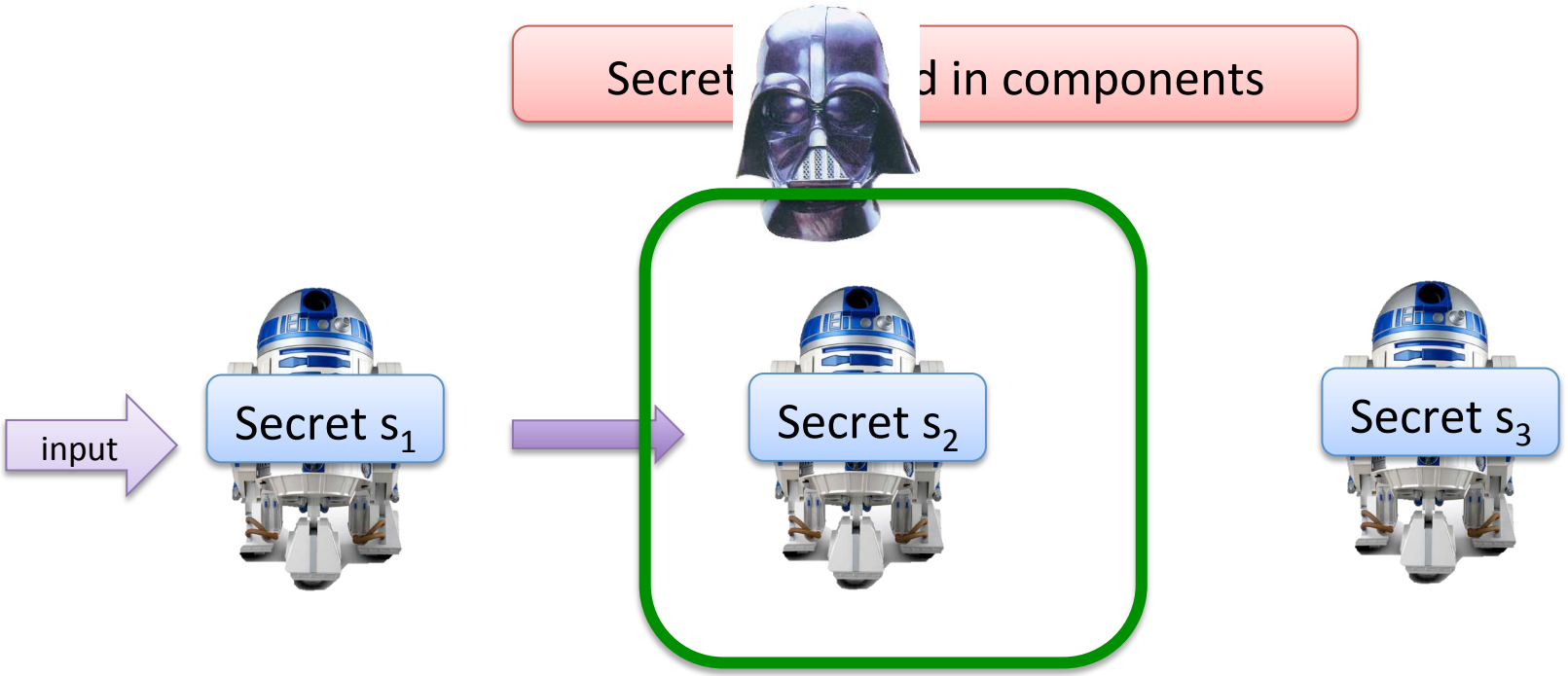
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



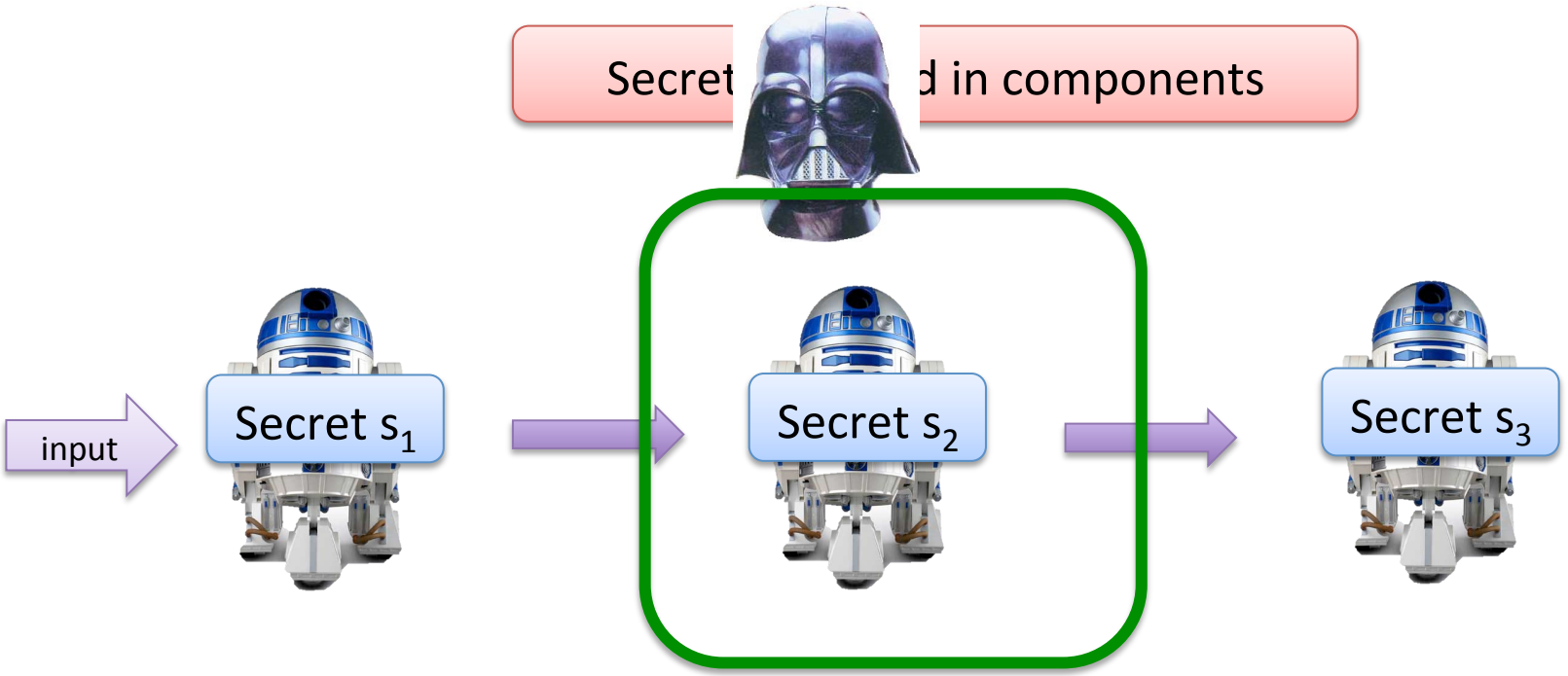
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



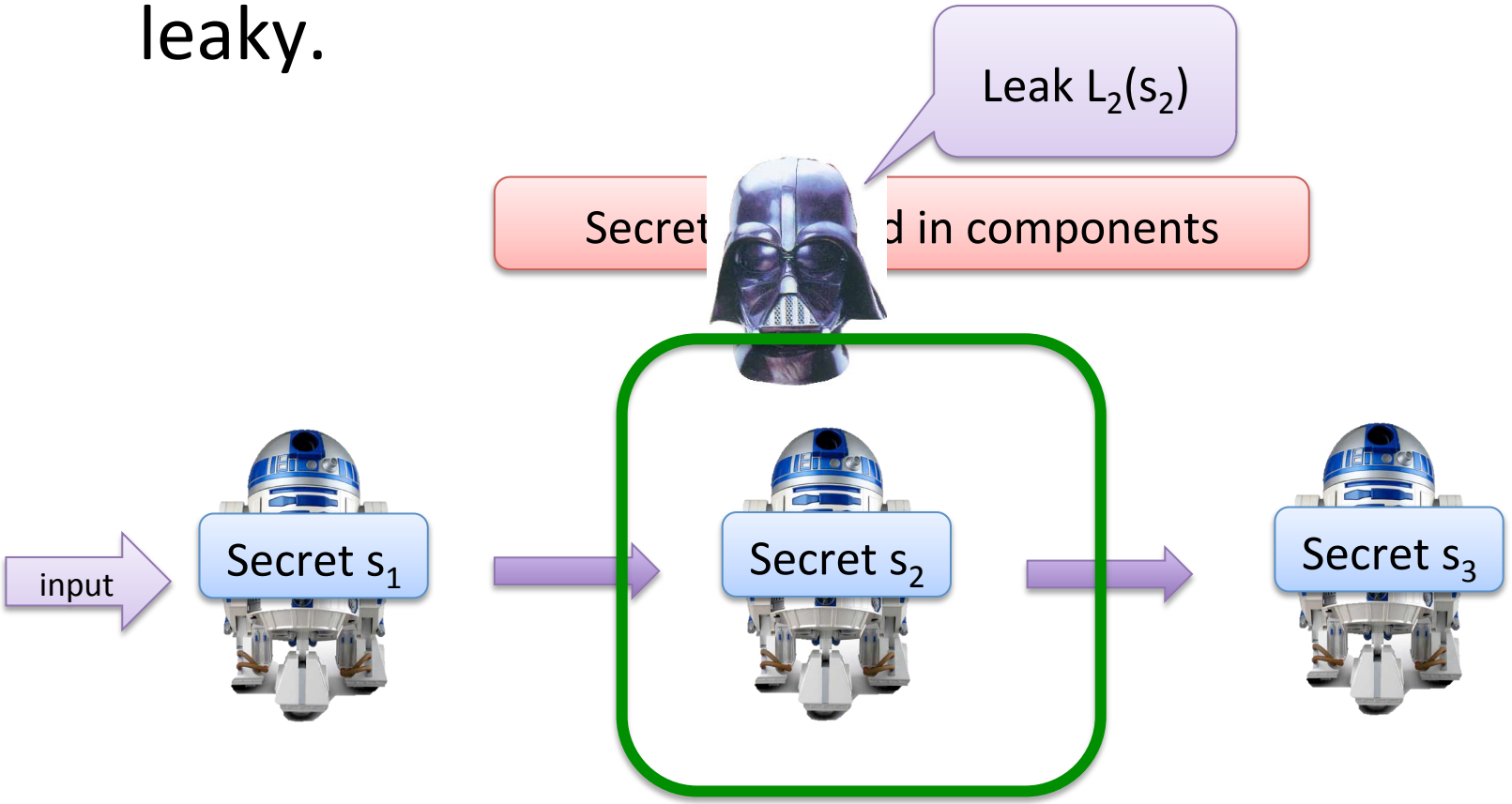
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



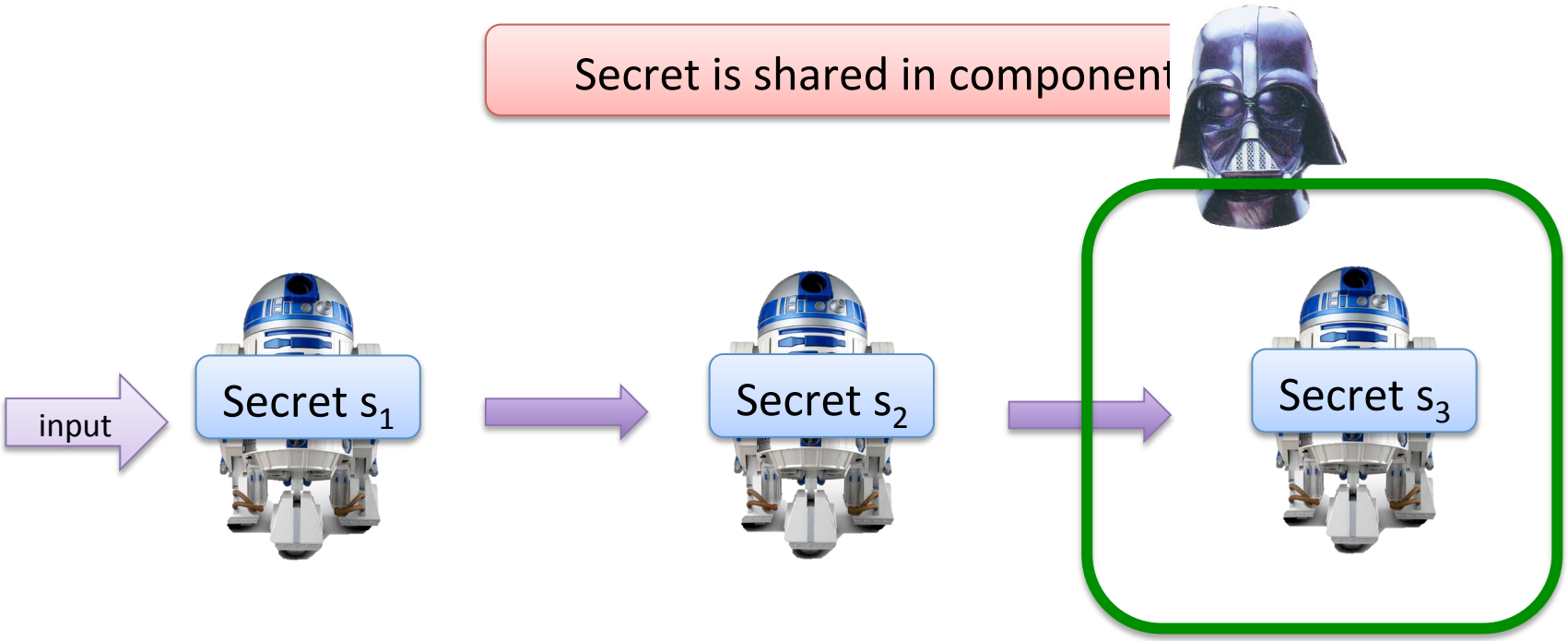
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



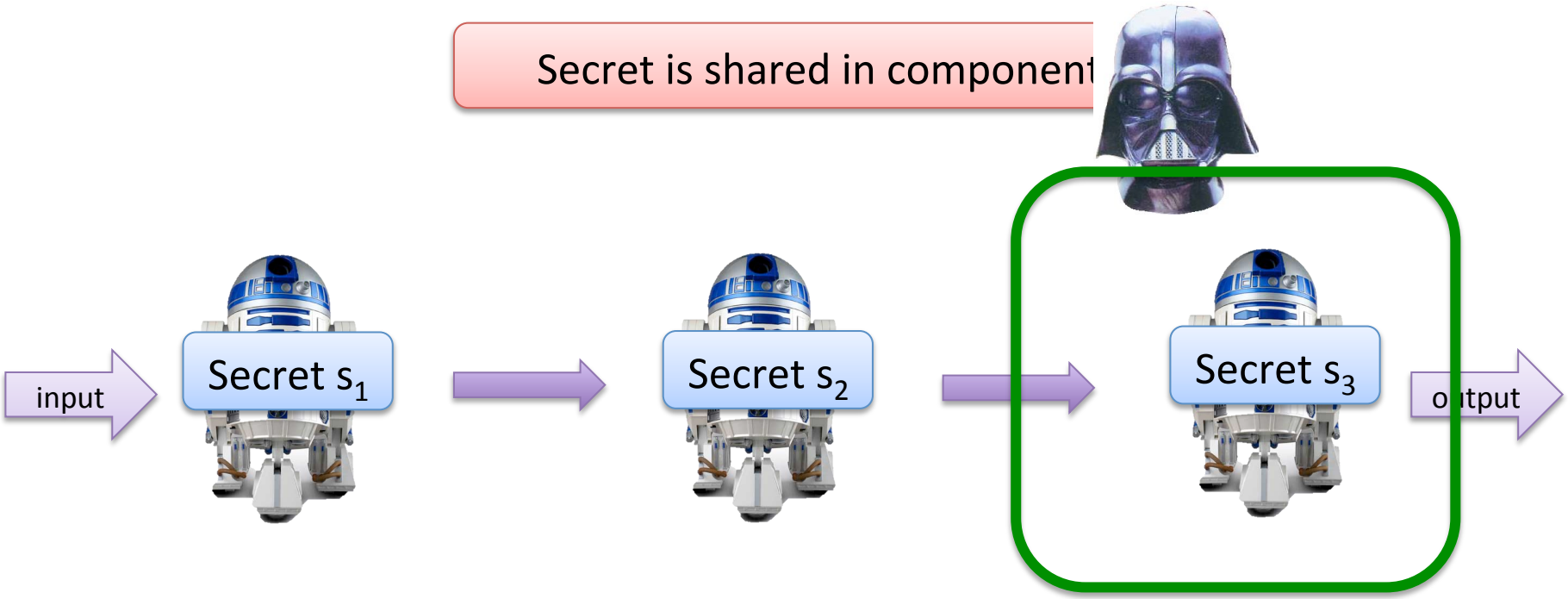
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



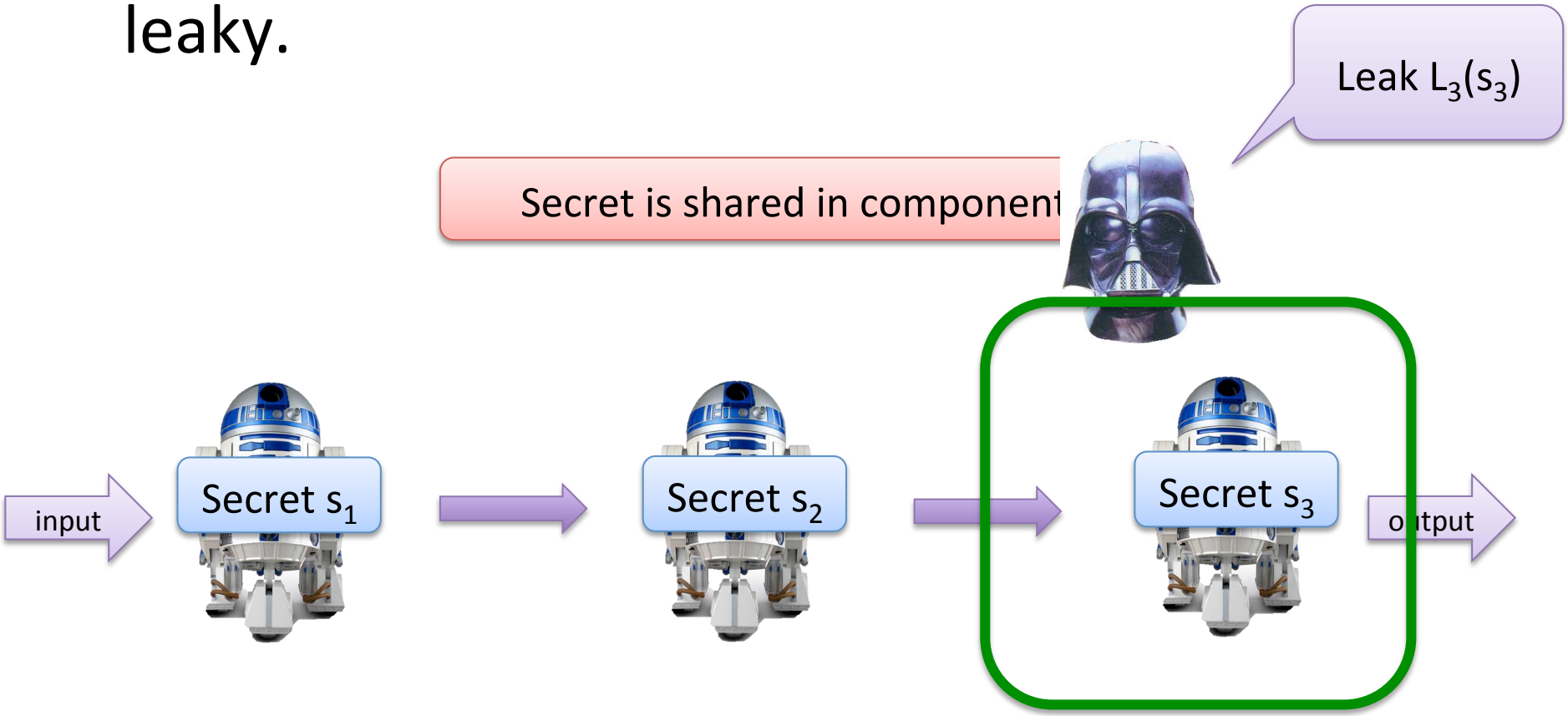
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



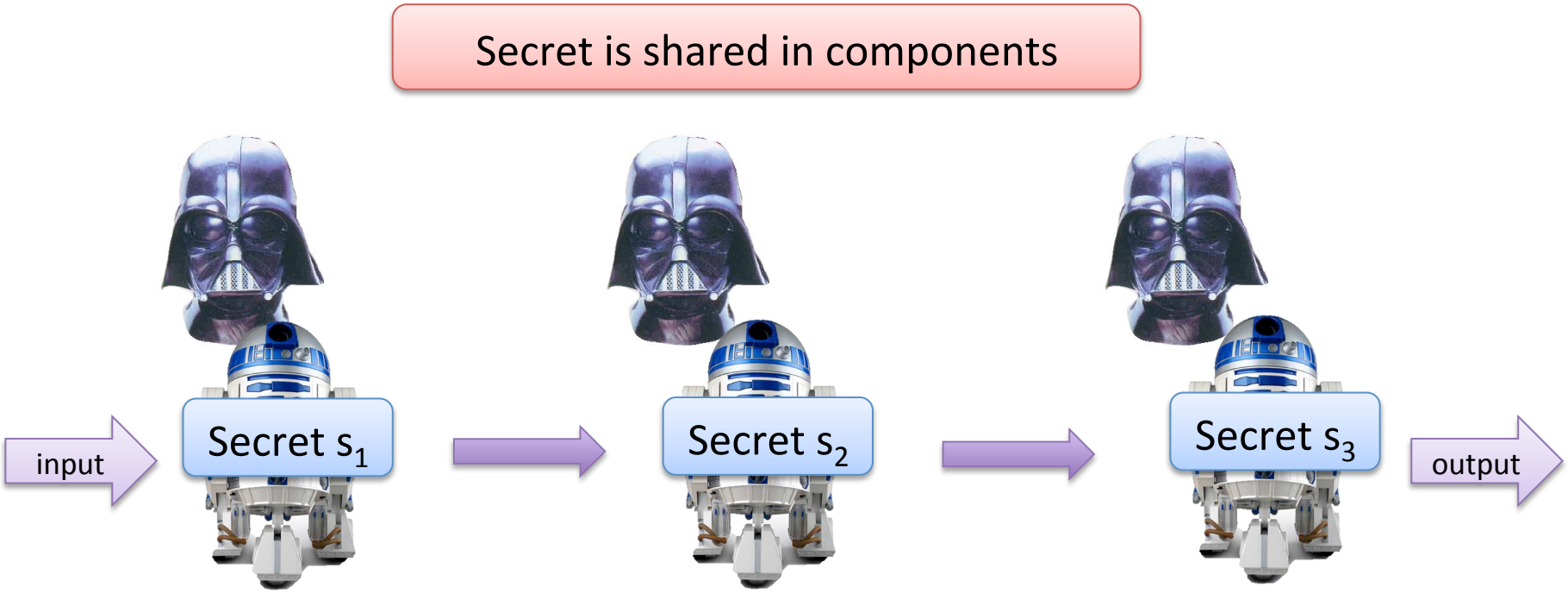
Only Computation Leaks (OCL) [MR]

- Idea: computation is performed by multiple components, where only the active ones are leaky.



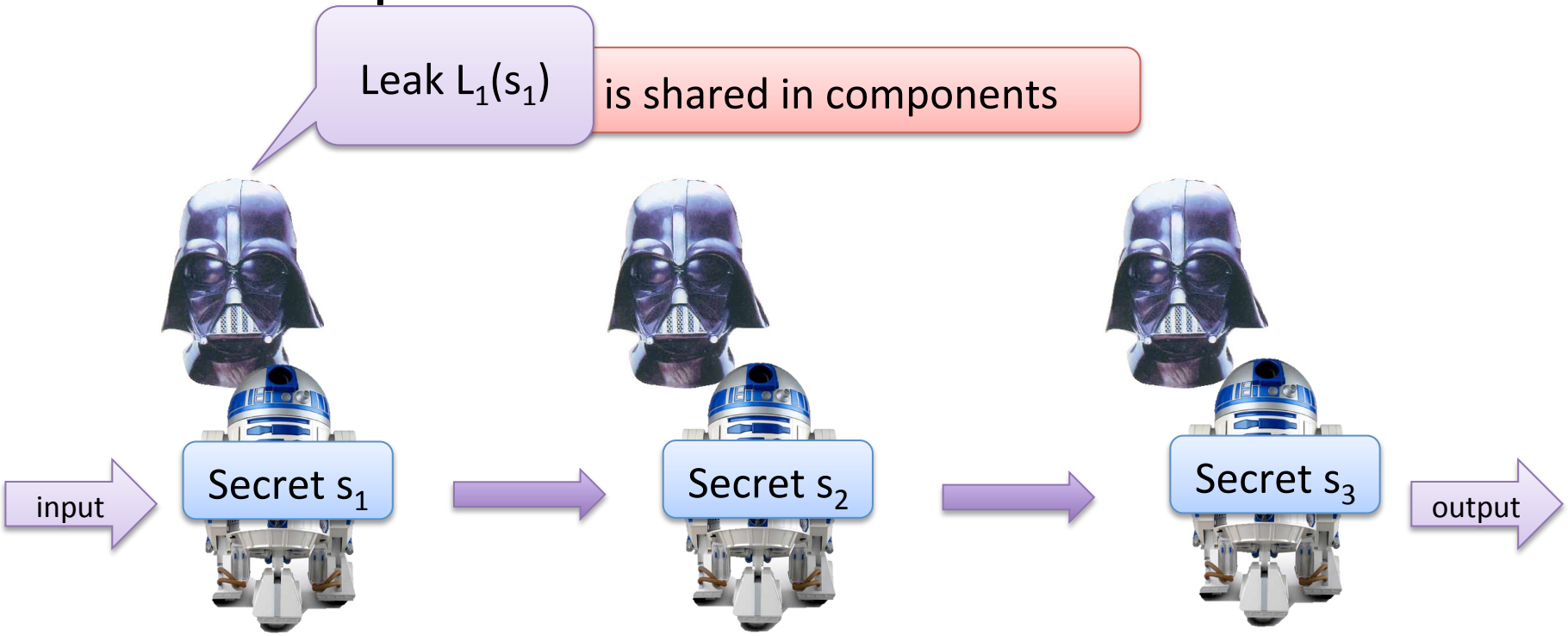
Generalized Model

- Adversary can leak on an arbitrary order of the component.



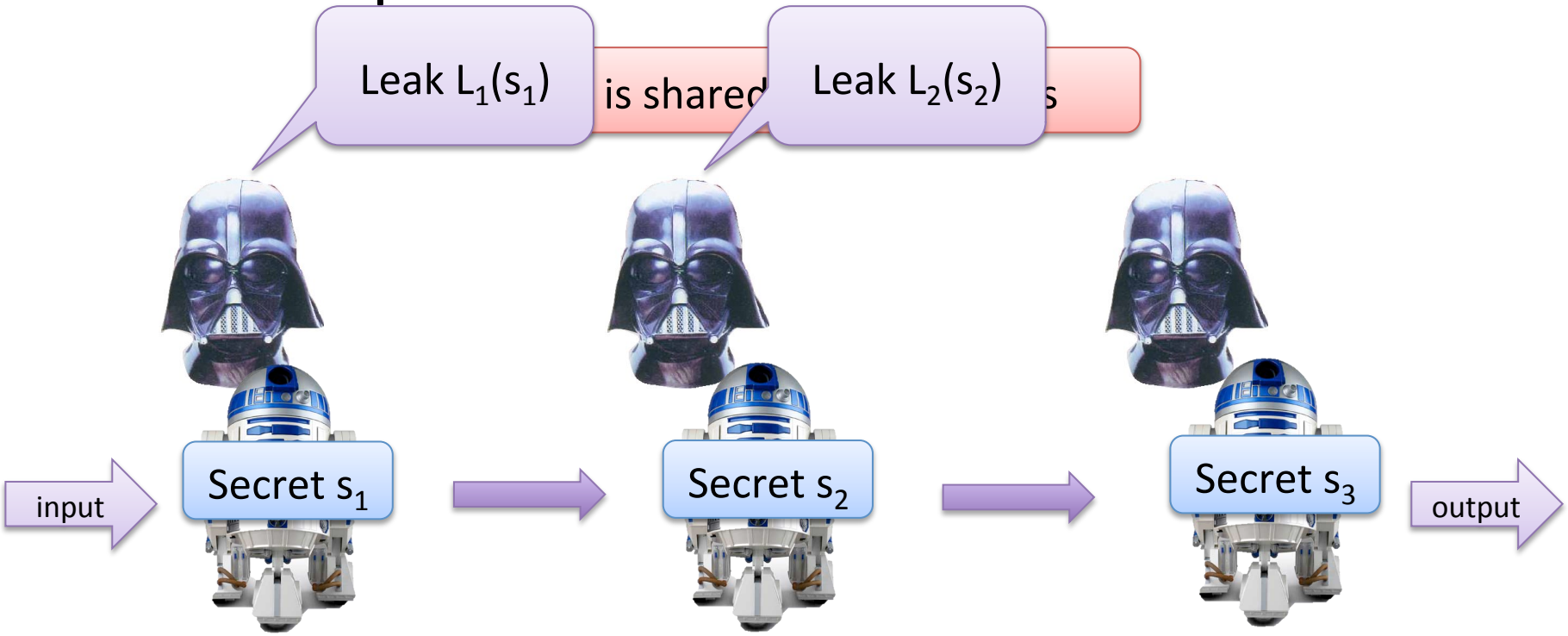
Generalized Model

- Adversary can leak on an arbitrary order of the component.



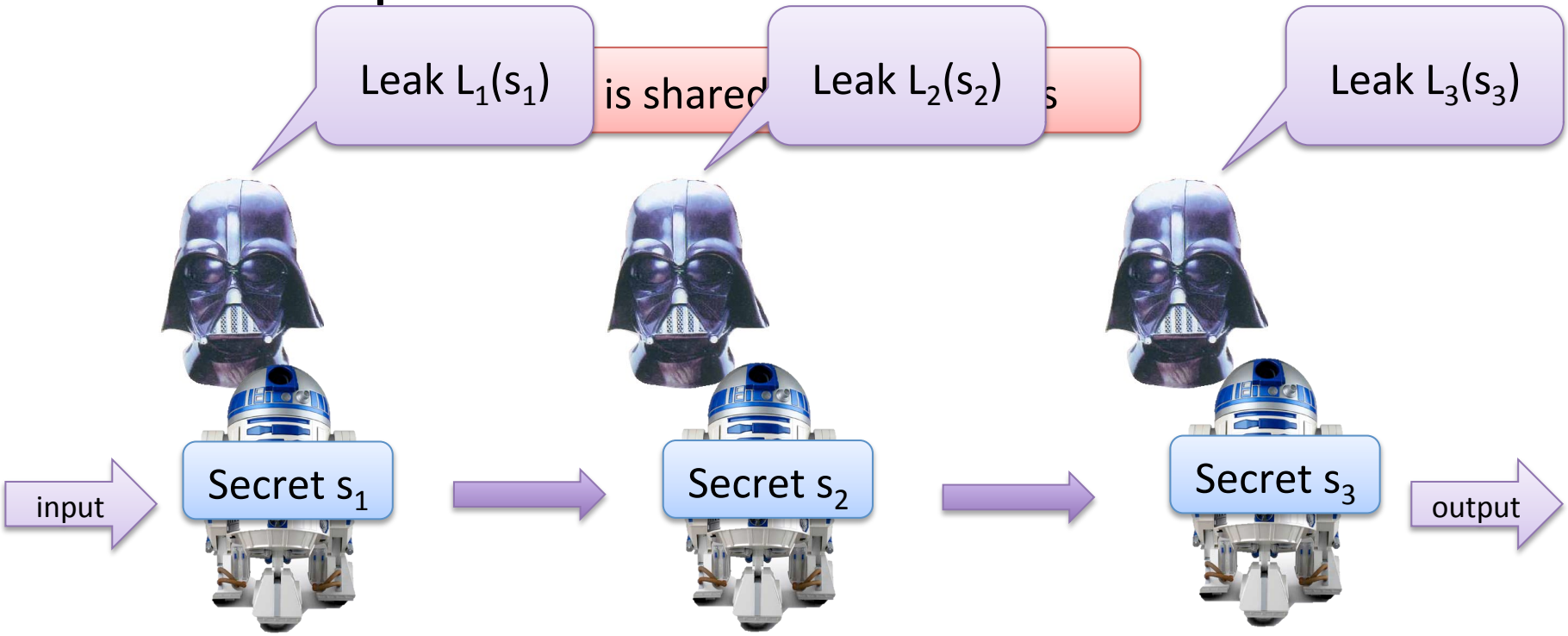
Generalized Model

- Adversary can leak on an arbitrary order of the component.



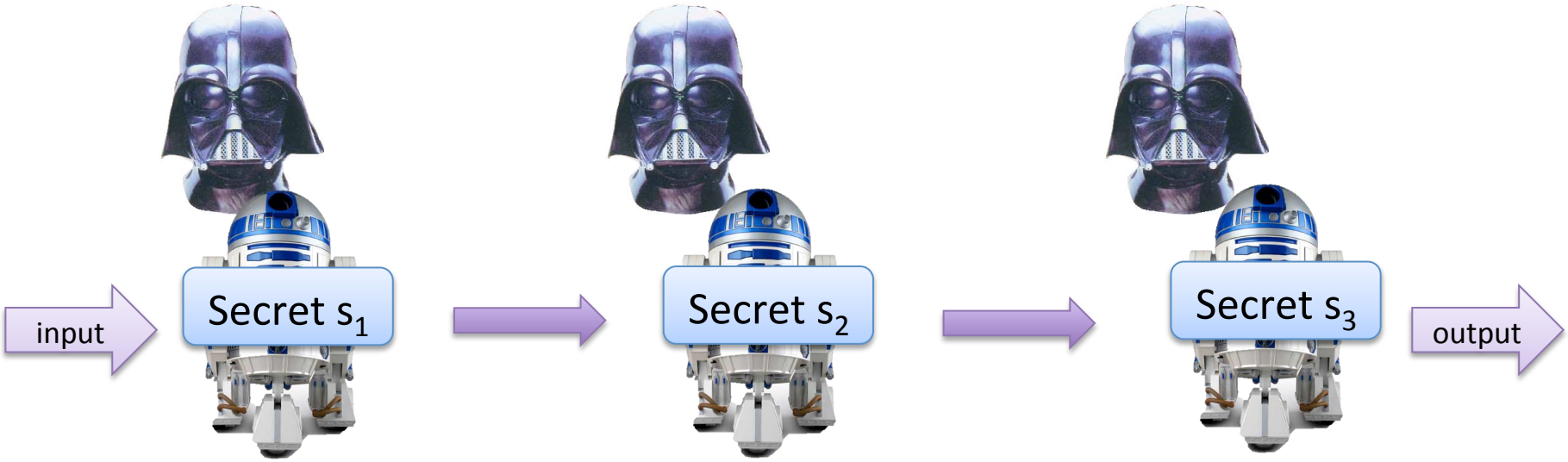
Generalized Model

- Adversary can leak on an arbitrary order of the component.



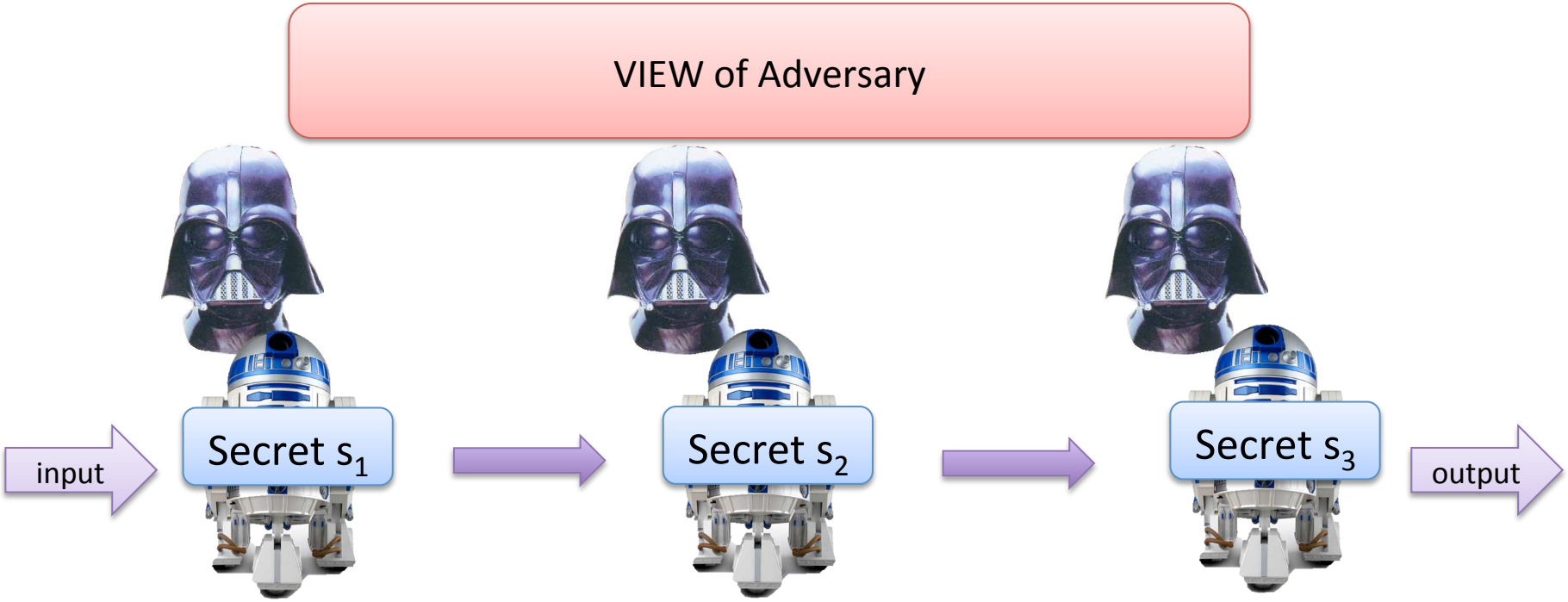
Security

- The adversary learns nothing more than black-box access to the device.



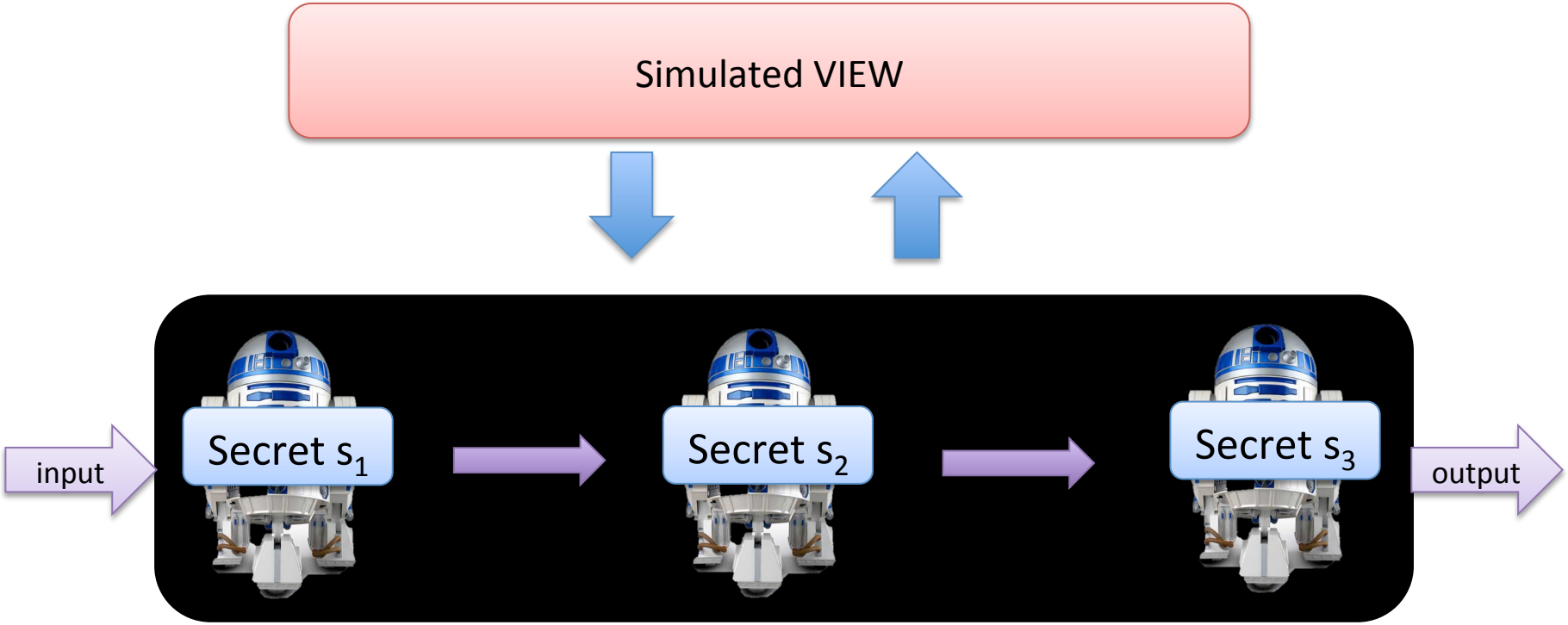
Security

- The adversary learns nothing more than black-box access to the device.



Security

- The adversary learns nothing more than black-box access to the device.



How to measure the “quality” of a construction

- Functionality

How to measure the “quality” of a construction

- Functionality
- Number of components

How to measure the "quality" of a construction

- Functionality
- Number of components
- Extra Secure Hardware

Secure Hardware

Our Goal

- How do we secure general computation?
  - With the optimal number of components
  - Without secure hardware

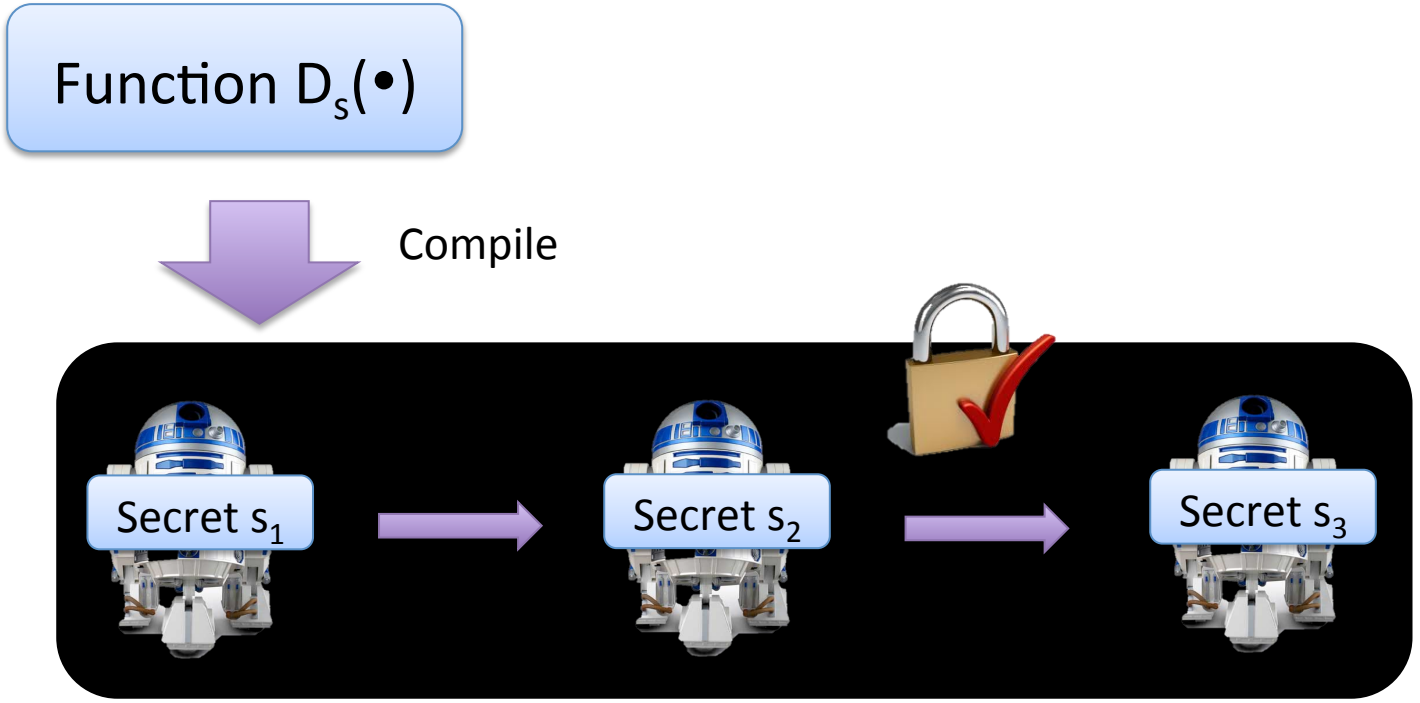
Our Goal

- How do we secure general computation?
  - With the optimal number of components
  - Without secure hardware

Function $D_s(\bullet)$

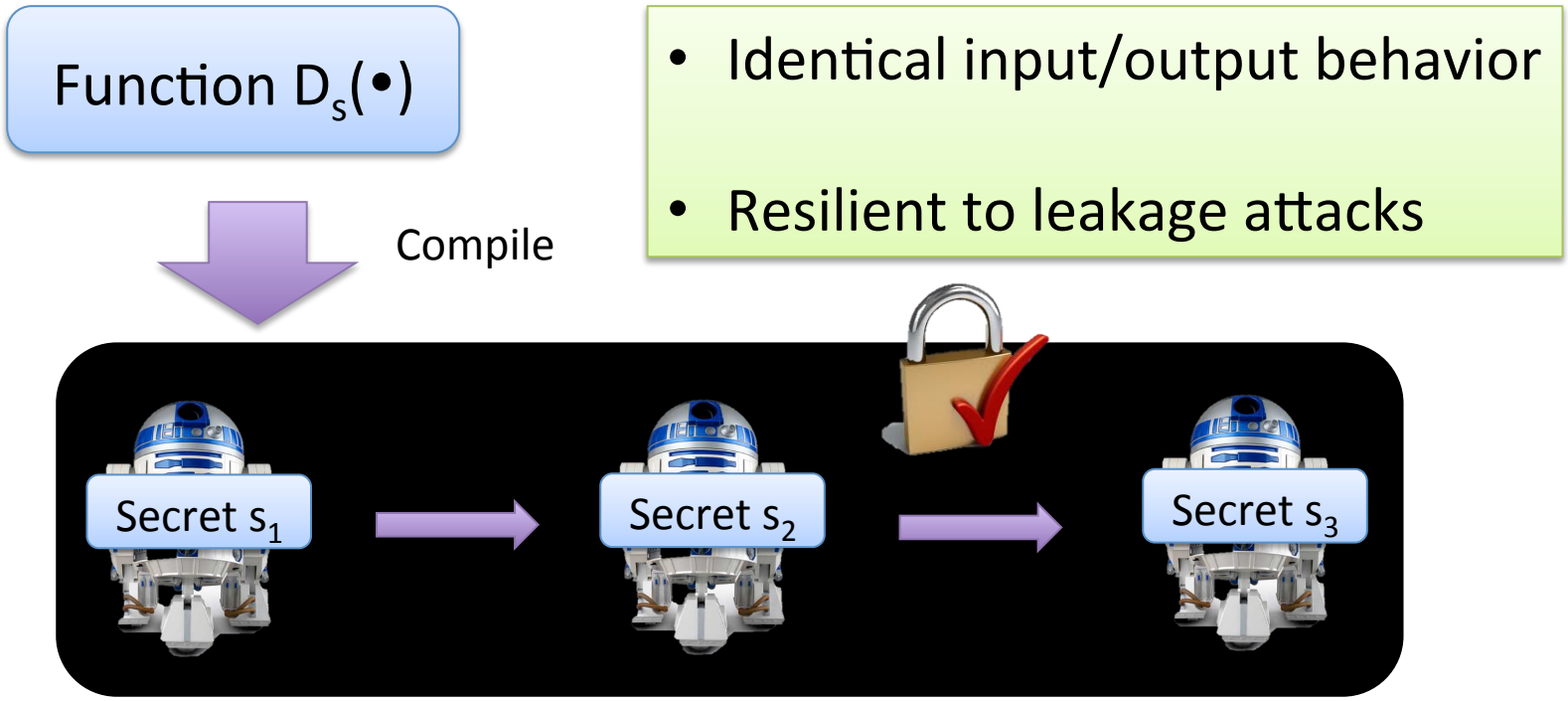
Our Goal

- How do we secure general computation?
  - With the optimal number of component
  - Without secure hardware



Our Goal

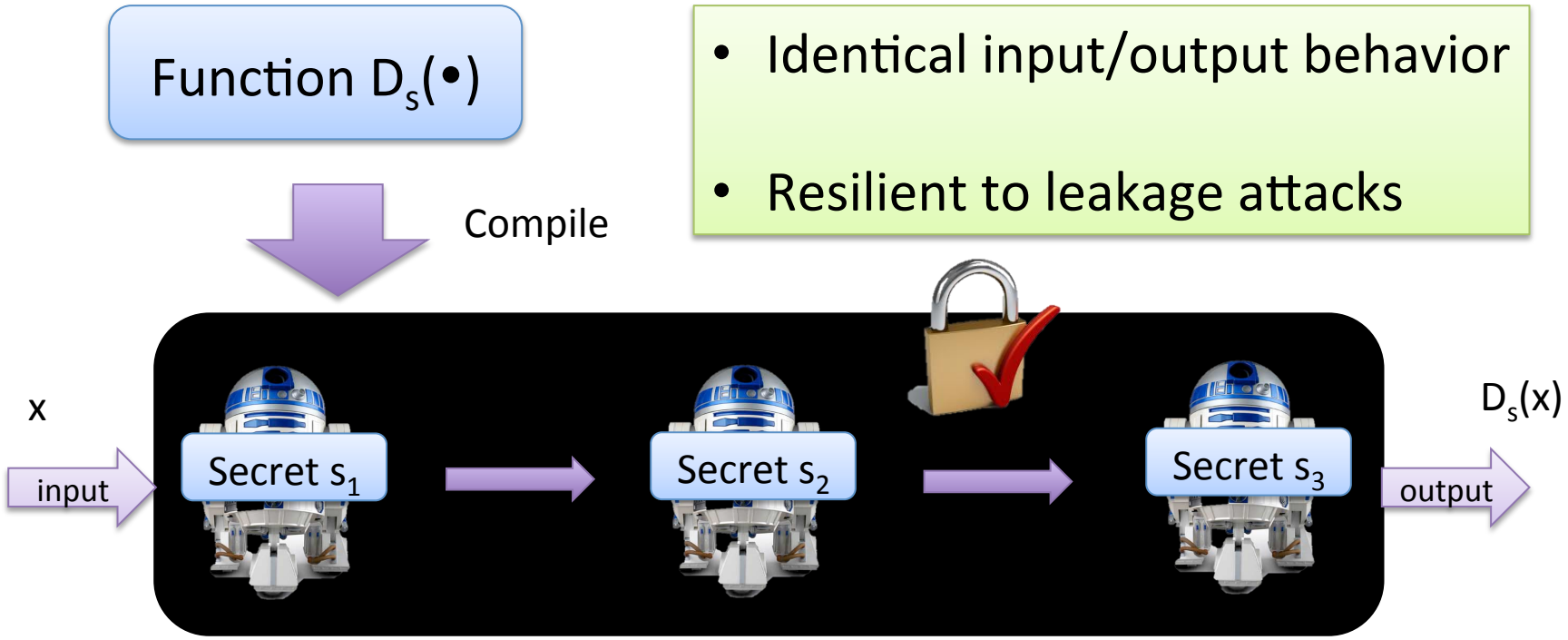
- How do we secure general computation?
  - With the optimal number of component
  - Without secure hardware



- Identical input/output behavior
- Resilient to leakage attacks

Our Goal

- How do we secure general computation?
  - With the optimal number of component
  - Without secure hardware



Previous Result

- It is impossible for one component (without secure hardware) [folklore,GR12]
- For multi-component constructions, we have:

Scheme	Hardware	Components
JV10	**Yes**	2
DF11	**Yes**	2
GR12	No	**|C|**
BDL14	No	**20**

Our Main Result

- Get best of the two: 2 components without hardware!

Our Main Result

- Get best of the two: 2 components without hardware!

- A modular approach:
  - Generic way to replace hardware in previous schemes [JV, DF]

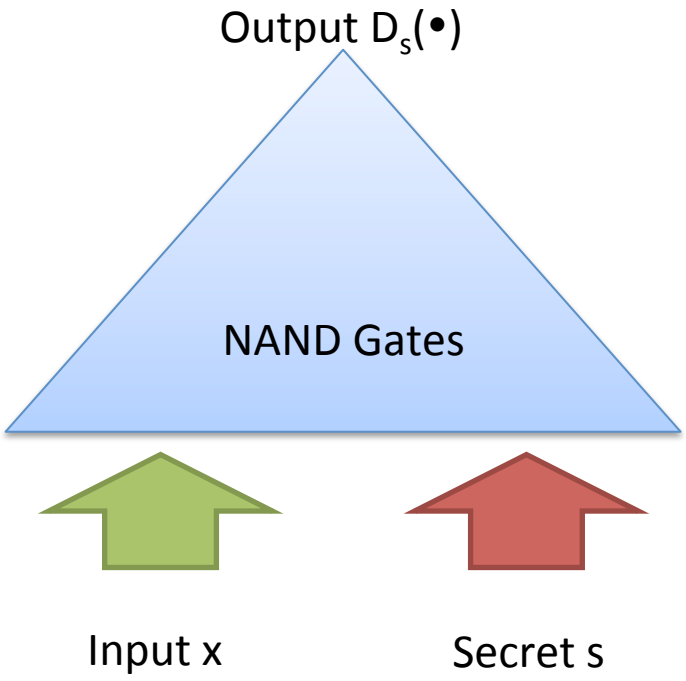
Scheme	Hardware	Components
JV-based	**No**	**2**
DF-based	**No**	**2**

Roadmap

- A generic design paradigm
  - Step1: design a hardware-based scheme
  - Step2: get rid of the hardware
    - Hardware replacement theorem
    - Implement sampling functionality

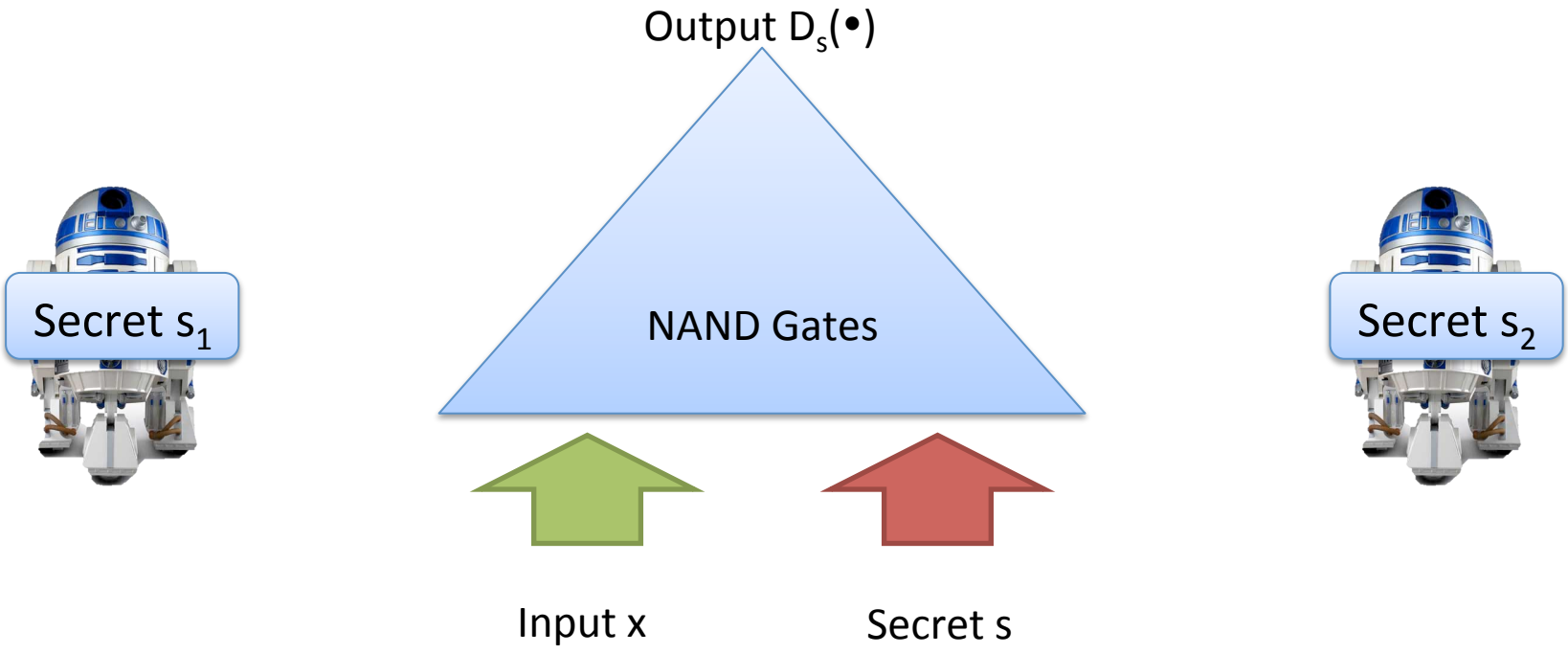
Original Dziembowski-Faust Scheme

- Given any $D_s(\bullet)$, we can express it as a circuit of NAND gates



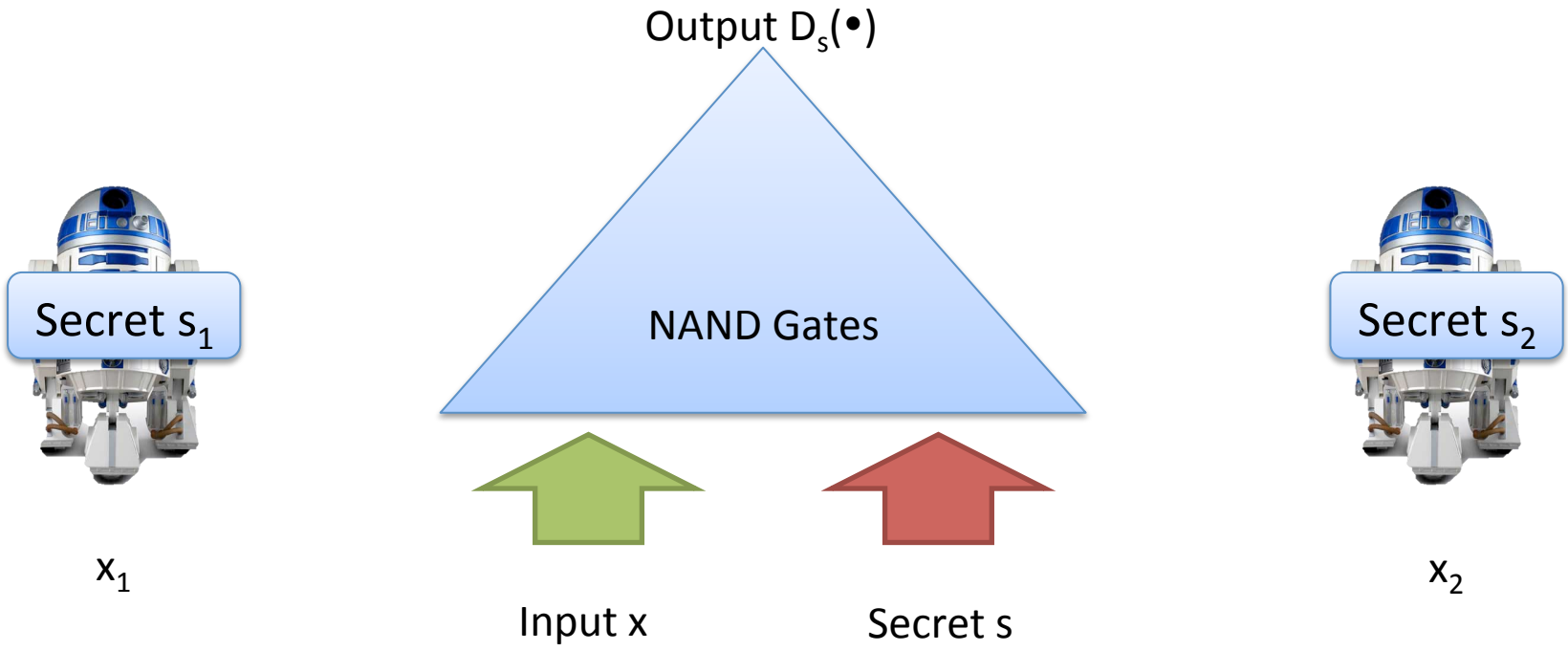
Original Dziembowski-Faust Scheme

- Given any $D_s(\bullet)$, we can express it as a circuit of NAND gates
  - Initially, secret share s



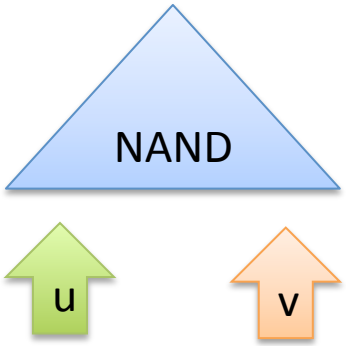
Original Dziembowski-Faust Scheme

- Given any $D_s(\bullet)$, we can express it as a circuit of NAND gates
  - Initially, secret share s
  - On input x, secret share x



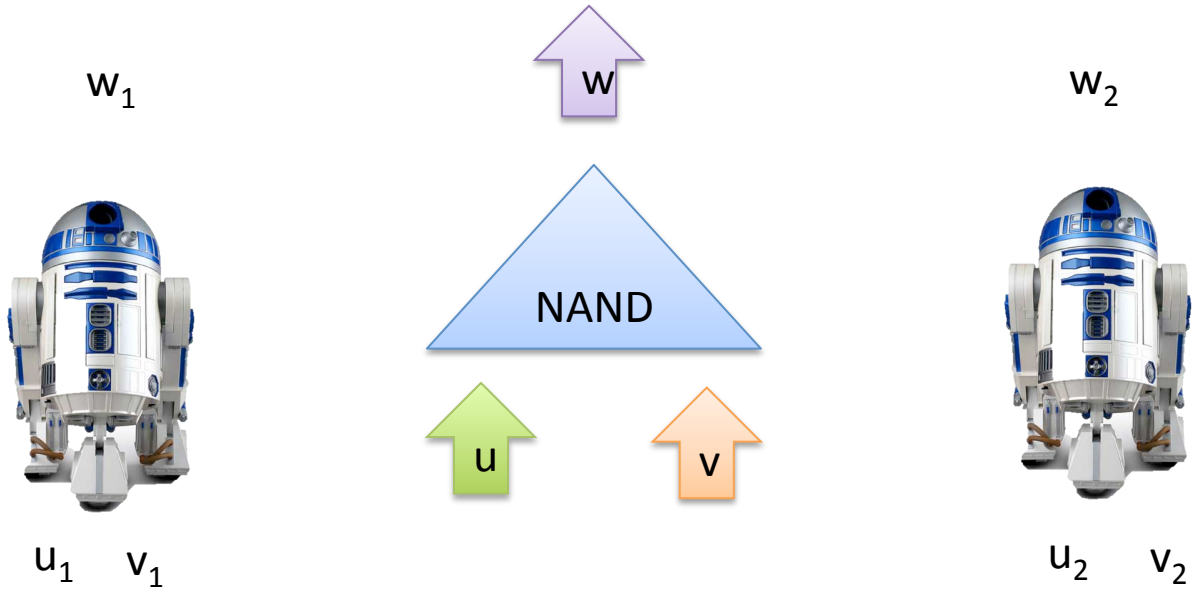
Original Dziembowski-Faust Scheme

u_1 v_1



u_2 v_2

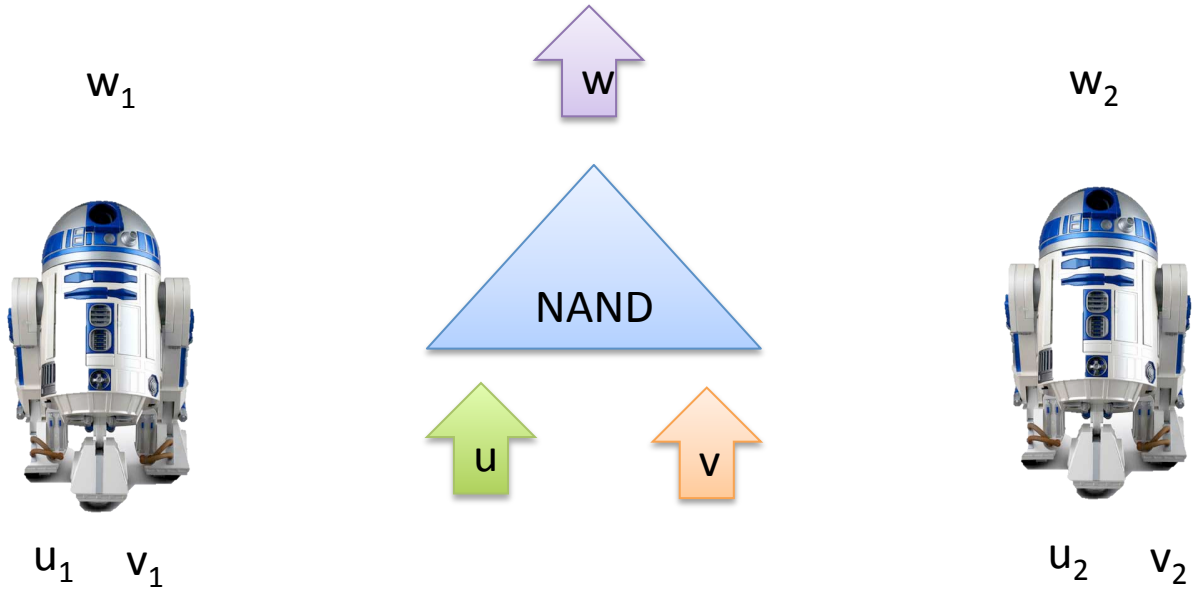
Original Dziembowski-Faust Scheme



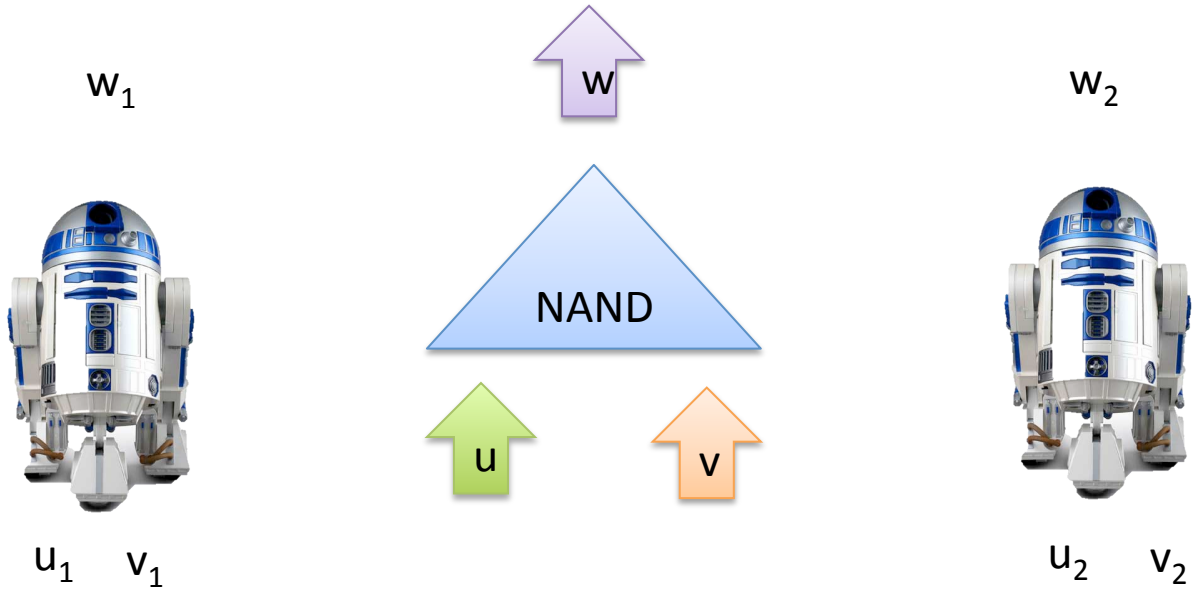
Original Dziembowski-Faust Scheme

Great property of shares

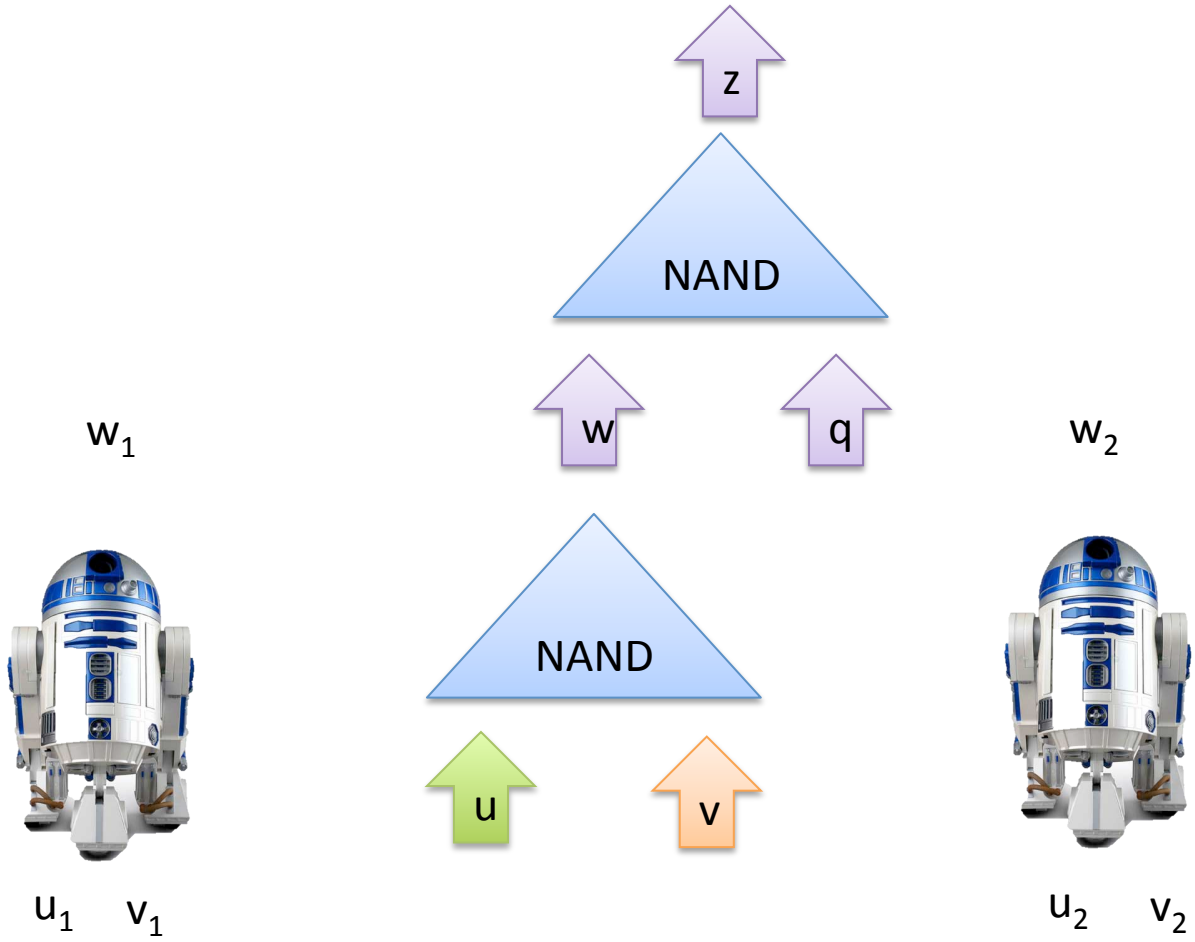
- Independent leakage on shares cannot reveal the underlying value!
- u is hidden given $L_1(u_1)$, $L_2(u_2)$, for bounded length functions



Original Dziembowski-Faust Scheme

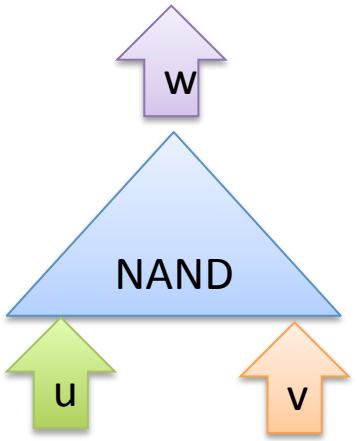


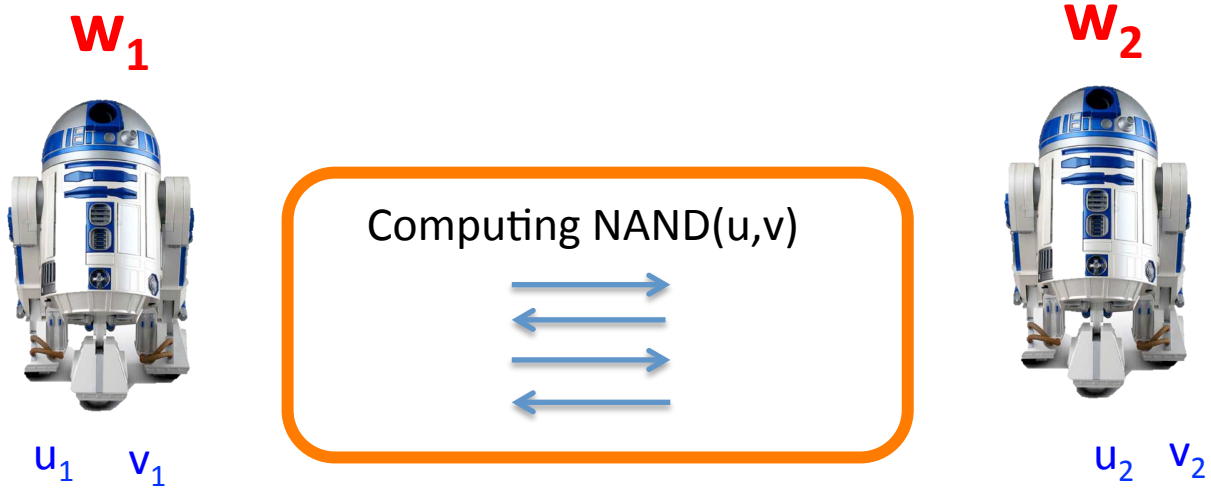
Original Dziembowski-Faust Scheme



Original Dziembowski-Faust Scheme

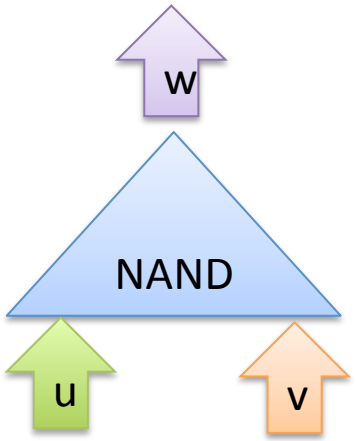
- DF designed a scheme
  - A Protocol for NAND

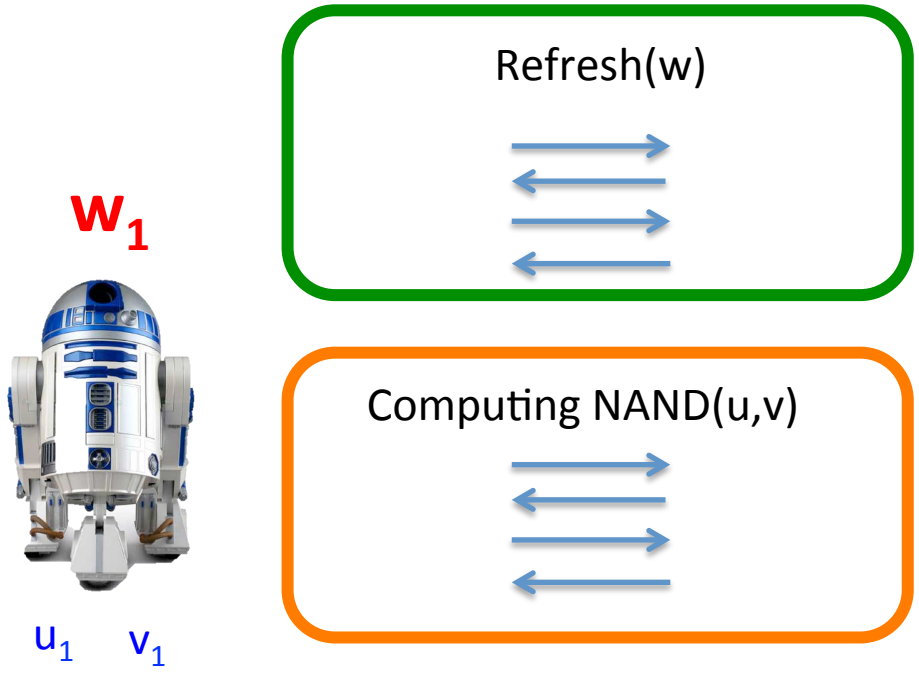




Original Dziembowski-Faust Scheme

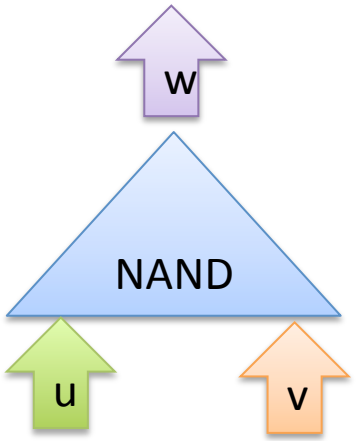
- DF designed a scheme
  - A Protocol for NAND
  - A Protocol Refresh

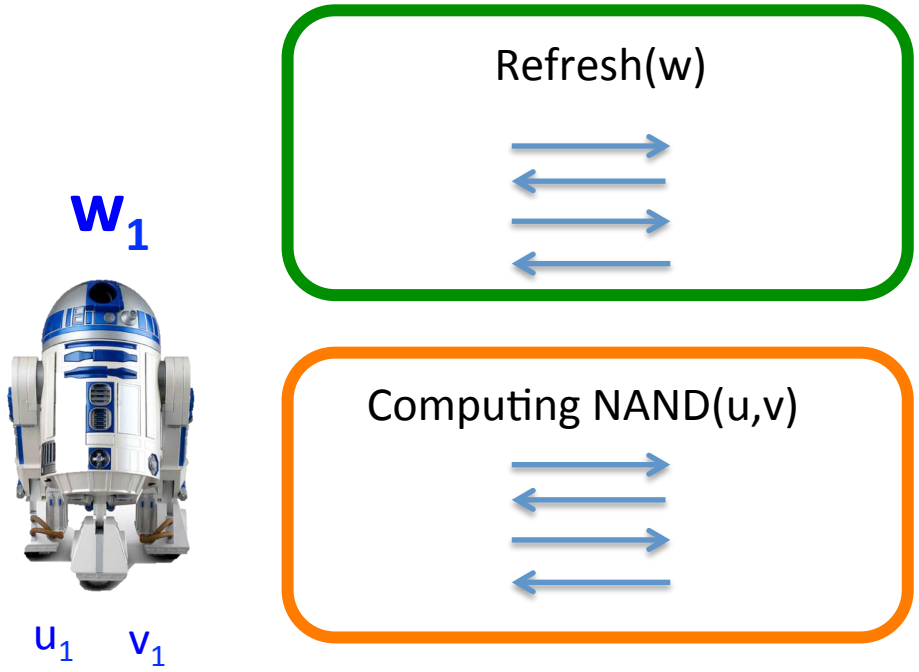




Original Dziembowski-Faust Scheme

- DF designed a scheme
  - A Protocol for NAND
  - A Protocol Refresh



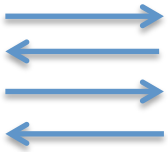


Original Dziembowski-Faust Scheme

- Refresh needs hardware

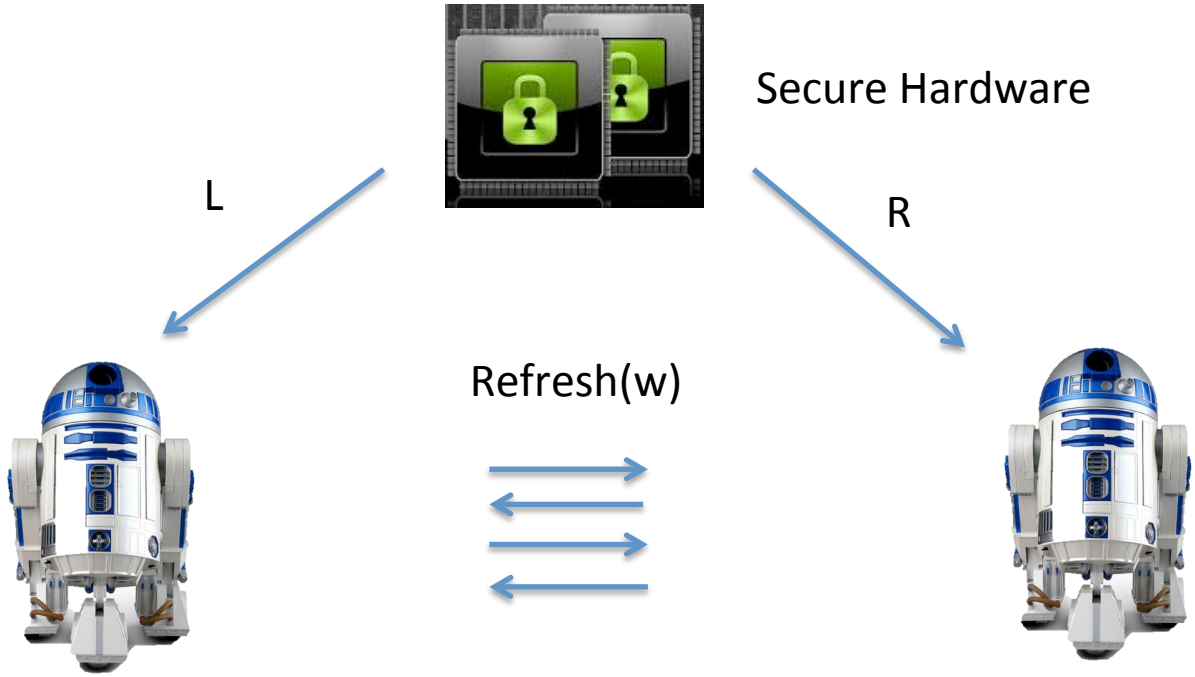
Secure Hardware

Refresh(w)



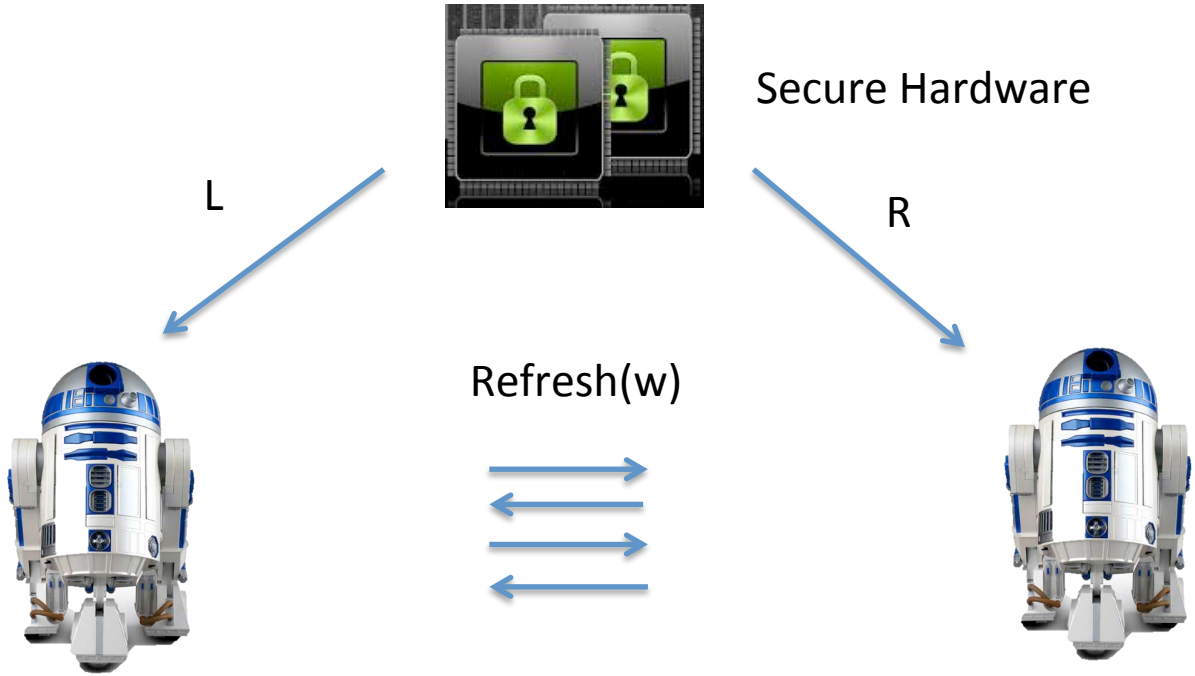
Original Dziembowski-Faust Scheme

- Refresh needs hardware



Original Dziembowski-Faust Scheme

- Refresh needs hardware



- L and R are vectors such that <L, R> = 0
- It is fine to leak on L and R separately, but NOT jointly

Roadmap

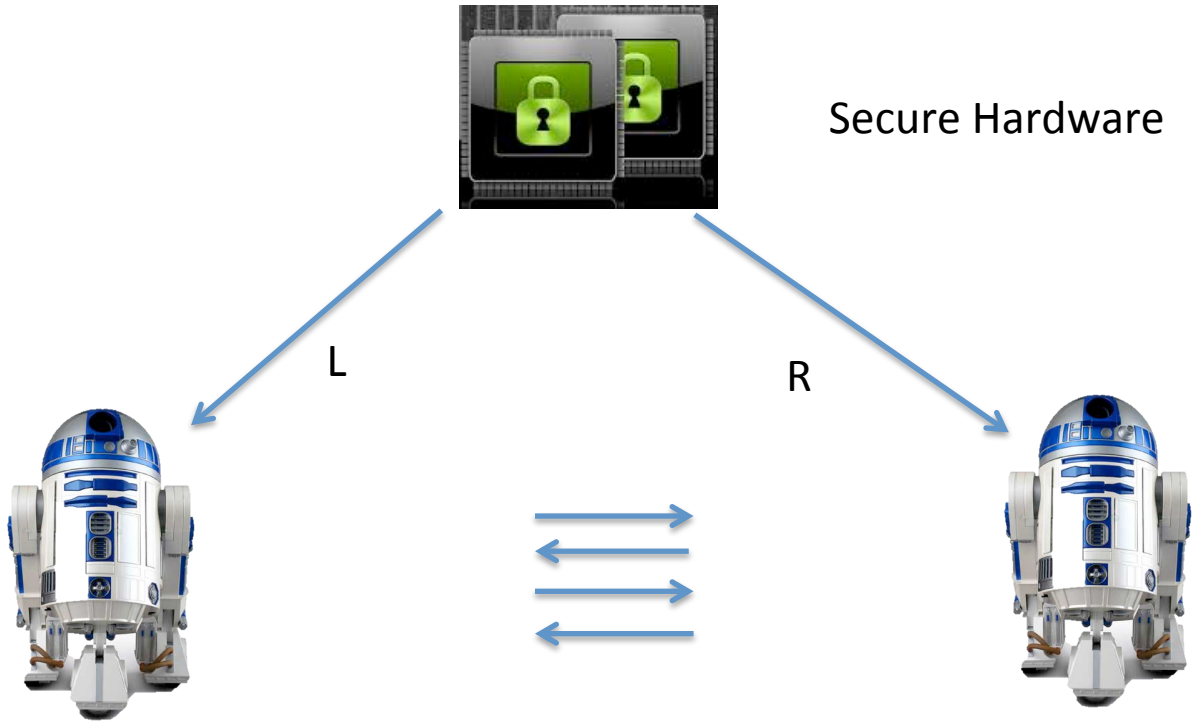
- A generic design paradigm
  - Step1: design a hardware-based scheme
  - Step2: get rid of the hardware
    - Hardware replacement theorem
    - Implement sampling functionality

Roadmap

- A generic design paradigm
  - Step1: design a hardware-based scheme
  - Step2: get rid of the hardware
    - Hardware replacement theorem
    - Implement sampling functionality

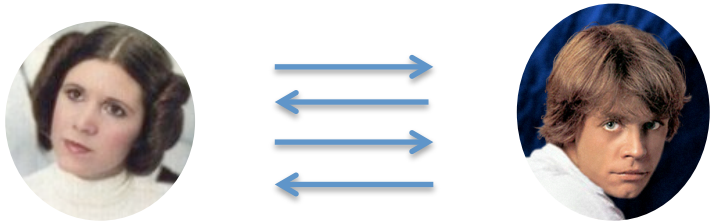
Hardware Replacement Theorem

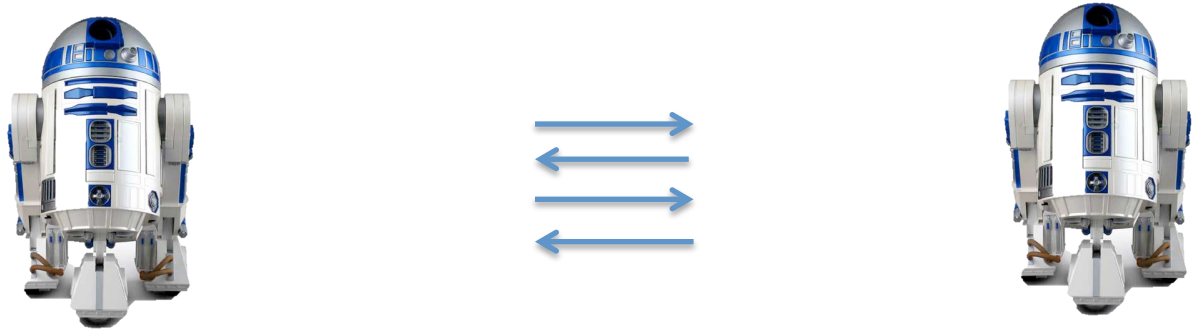
- Given any hardware-based scheme



Hardware Replacement Theorem

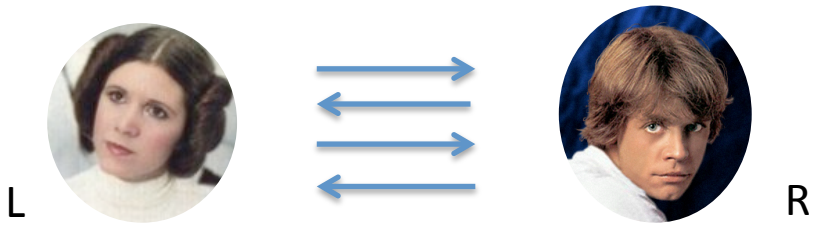
- Given any hardware-based scheme

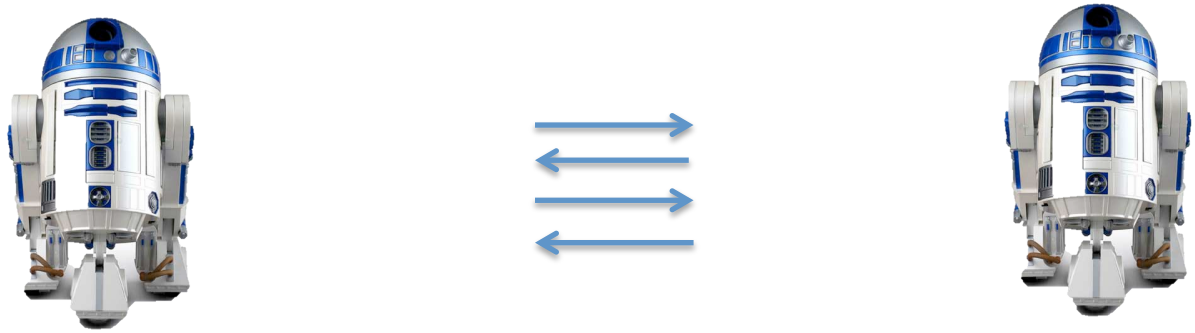




Hardware Replacement Theorem

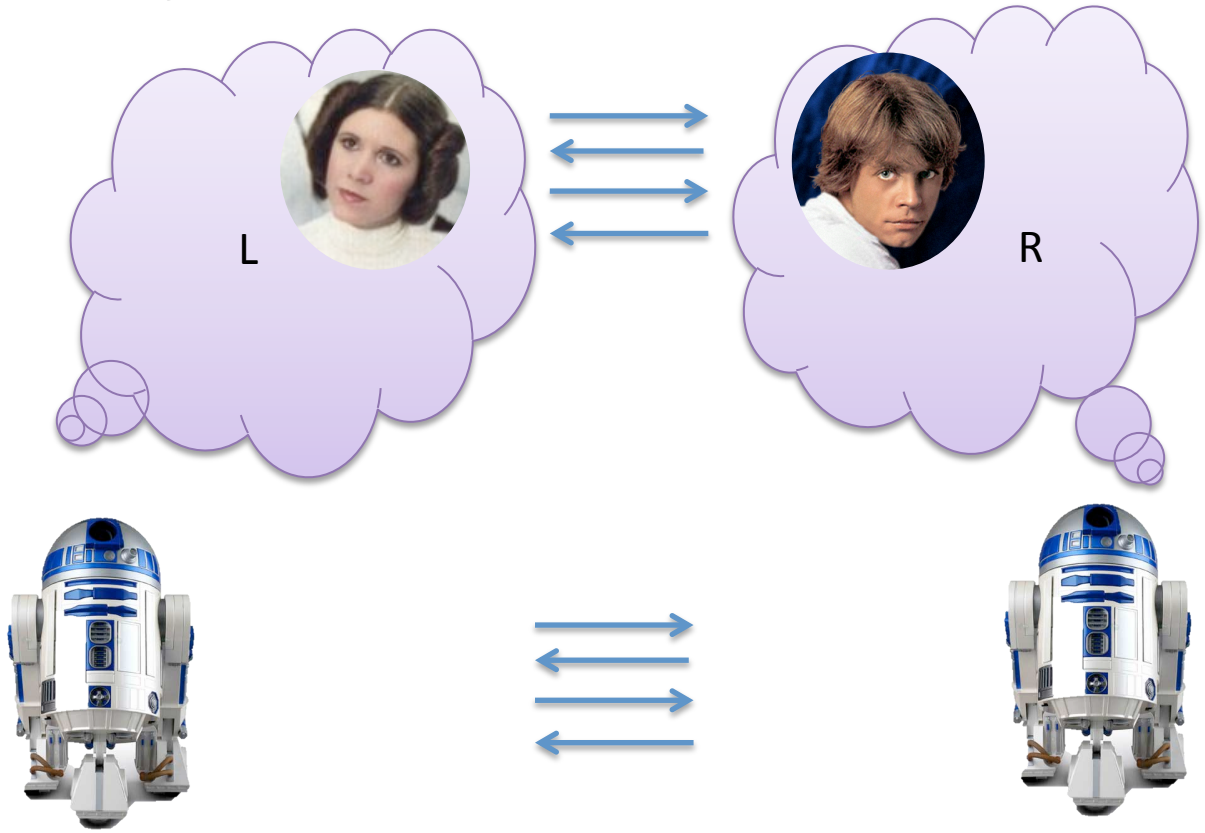
- Given any hardware-based scheme





Hardware Replacement Theorem

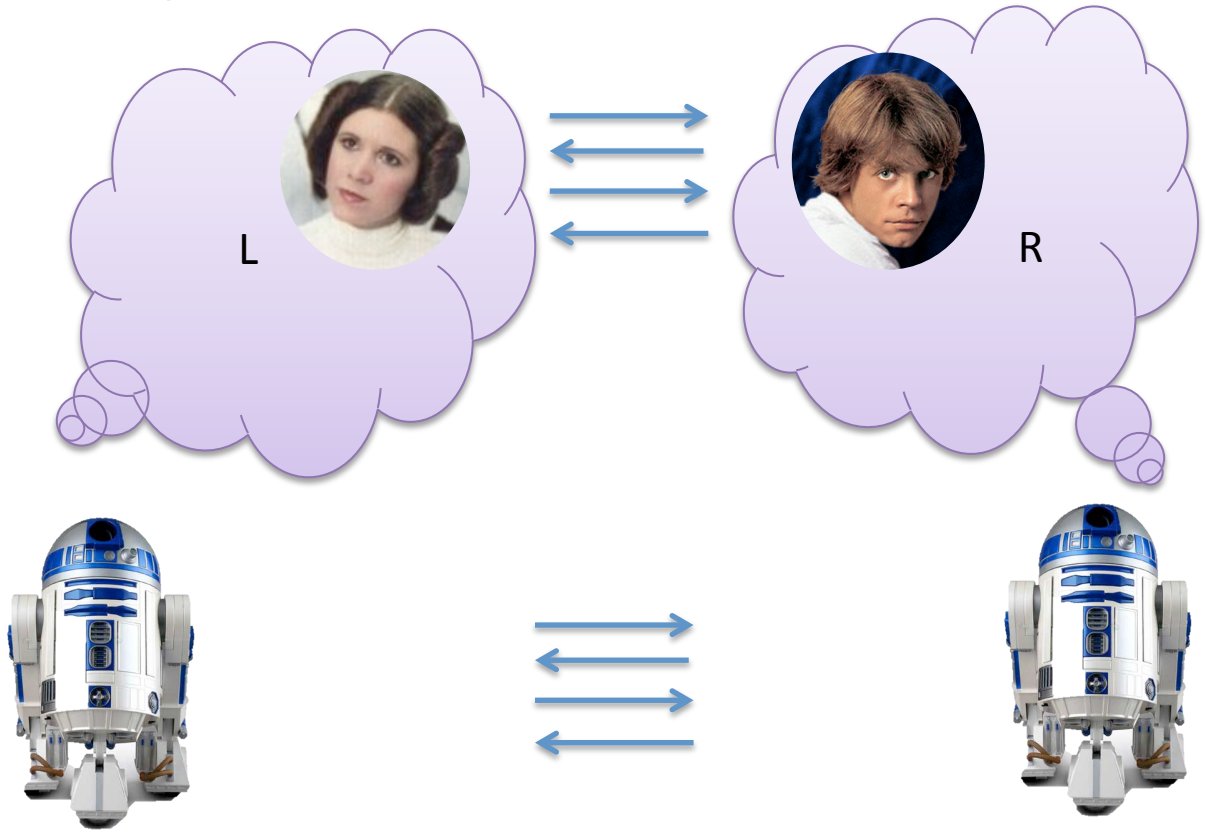
- Given any hardware-based scheme



Similar to UC, leaky UC [BCH12]

Hardware Replacement Theorem

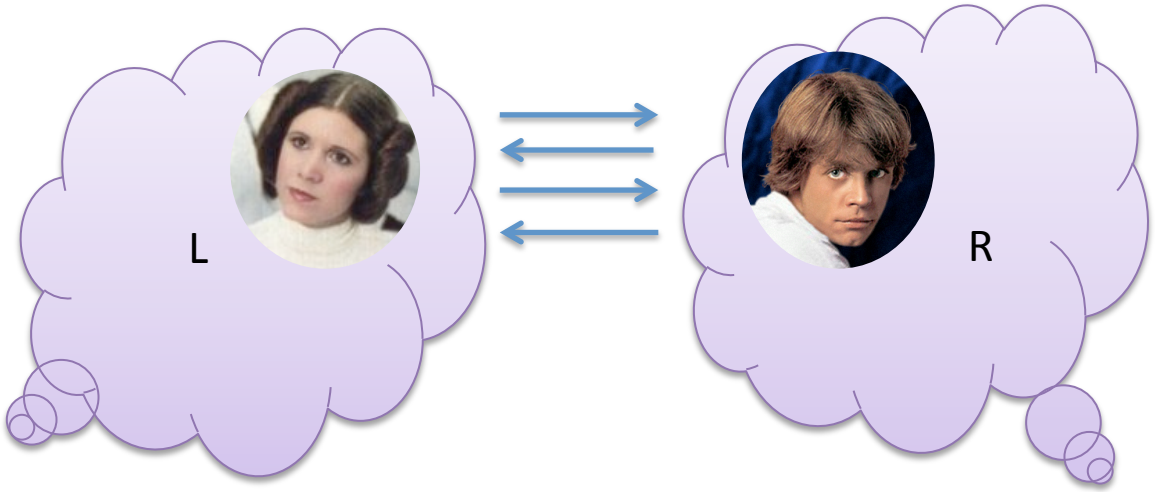
- Given any hardware-based scheme

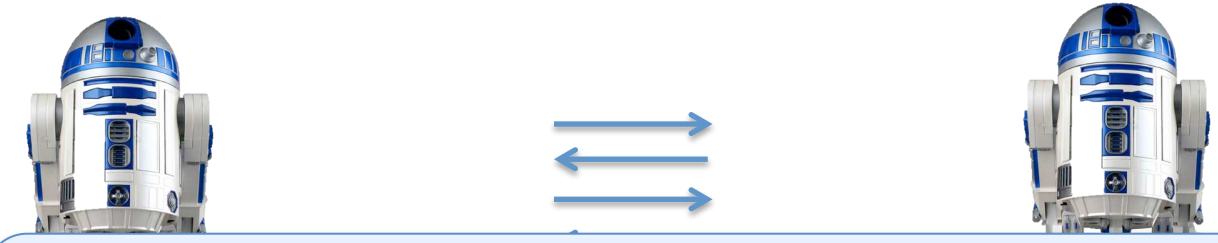


Similar to UC, leaky UC [BCH12]

Hardware Replacement Theorem

- Given any hardware-based scheme





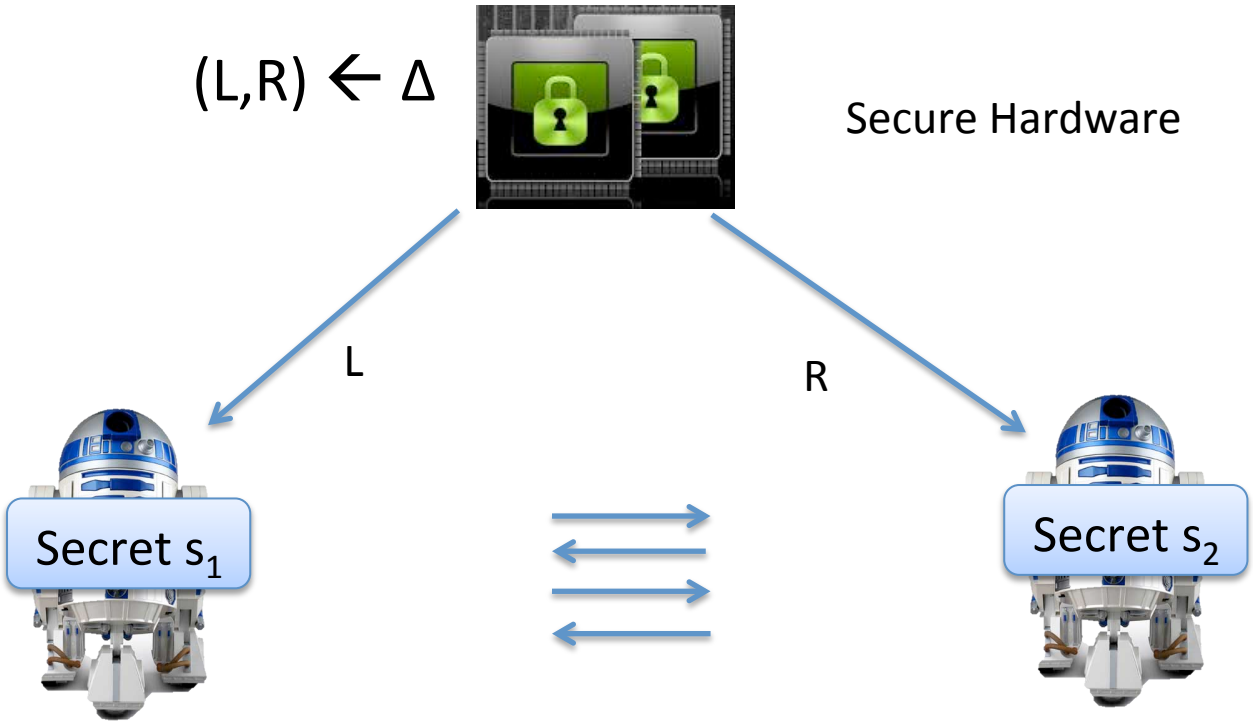
Challenge: Need to make sure Adv can not learn joint leakage of L and R !

Roadmap

- A generic design paradigm
  - Step1: design a hardware-based scheme
  - Step2: get rid of the hardware
    - Hardware replacement theorem
    - Implement sampling functionality

Sampling Functionality

- Let Δ be some distribution that samples (L, R)



How to Implement Sampling Functionality

How to Implement Sampling Functionality

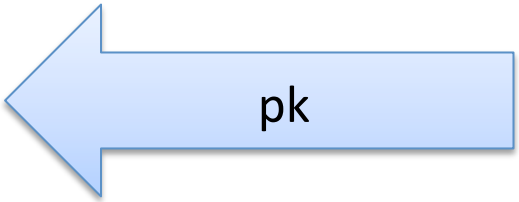
- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ
- Simple idea: let one party samples Δ and use encryption to protect the communication

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ
- Simple idea: let one party samples Δ and use encryption to protect the communication



(pk,sk) <- Gen

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ
- Simple idea: let one party samples Δ and use encryption to protect the communication

pk

Enc(pk,R)

1. Sample coins
2. Compute (L, R) = Δ(coins)
3. Compute Enc(pk, R)

(pk,sk) <- Gen

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ

Can obtain joint leakage on L and R if can leak on coins

Big Issue!!!

pk

Enc(pk,R)

1. Sample coins
2. Compute (L, R) = Δ(coins)
3. Compute Enc(pk, R)

(pk,sk) <- Gen

[JL00,CHK05]

Receiver Non-committing Encryption

pk

c = Enc(pk,m; r)

Sender

Receiver

(pk,sk) <- Gen

[SW14]

Univeral Deniable Encryption

Public Parameter PP

pk

c = Enc(PP, pk,m; r)

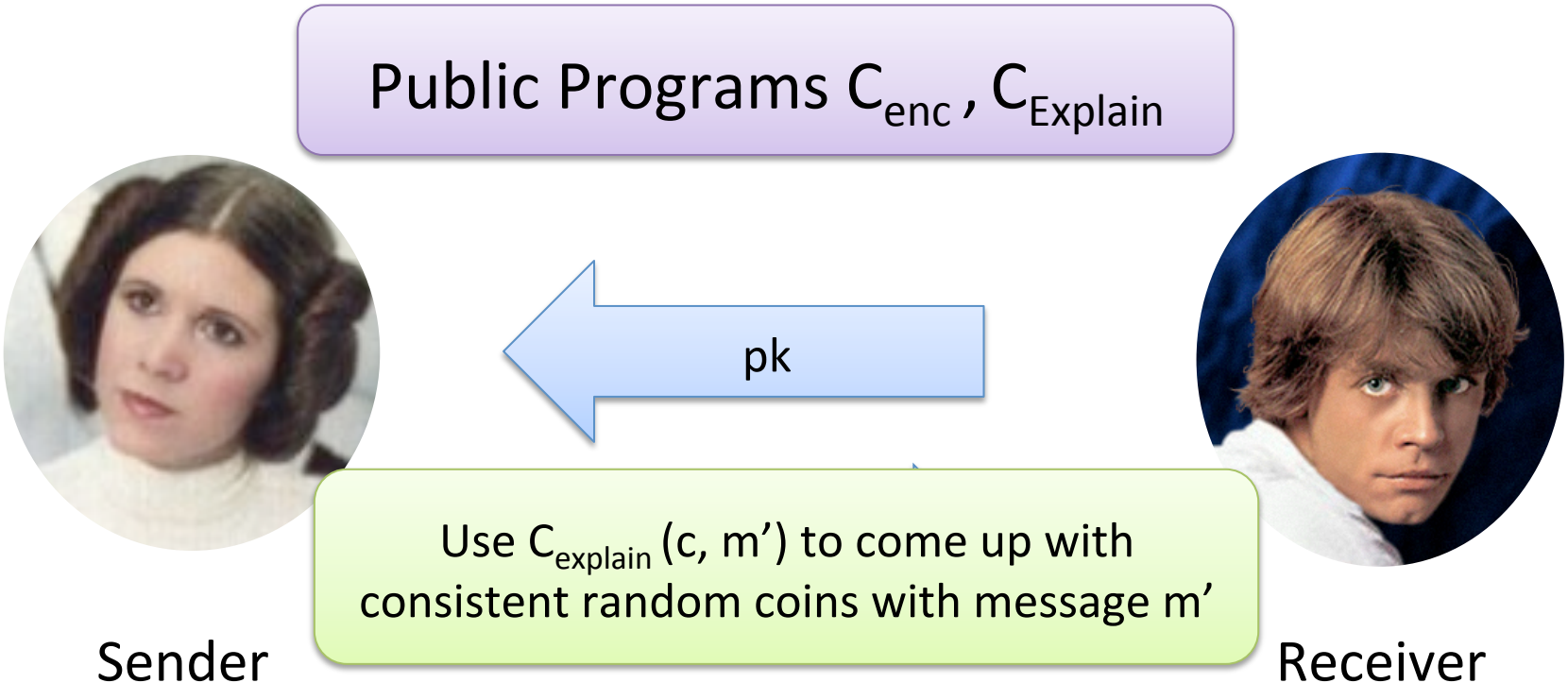
Sender

Receiver

For any m' in the message space, sender can produce a fake opening r' that is consistent with the transcript, i.e. c= Enc(PP, pk,m'; r')

Sahai-Waters' Transformation

- Theorem [SW] Given any encryption E, there exists an upgraded E* that is deniable.



How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ

pk

c_2 = Enc(pk,R)

(pk,sk) <- Gen

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ

Public Program C_{enc} : on input (pk, r),

1. Sample (L, R) ← Δ
2. Output (c_1,c_2) = (L, Enc(pk, R))

pk

c_2 = Enc(pk,R)

(pk,sk) <- Gen

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ

Public Program C_{enc} : on input (pk, r),

1. Sample (L, R) ← Δ
2. Output (c_1,c_2) = (L, Enc(pk, R))

pk

c_2 = Enc(pk,R)

1. Sample r
2. Run Program C_{enc}(pk, r) and obtain (c_1,c_2) = (L, Enc(pk, R))

(pk,sk) <- Gen

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ

1. Sample r
2. Run Program C_{enc}(pk, r) and obtain (c_1,c_2) = (L, Enc(pk, R))

pk

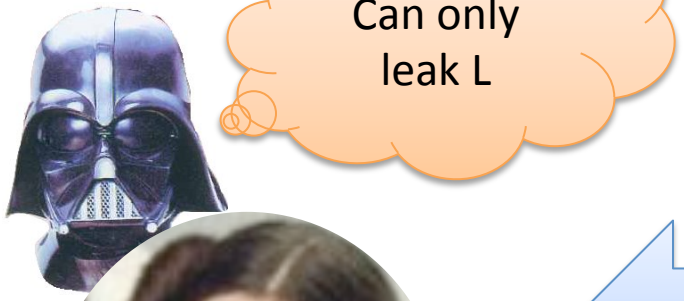
c_2 = Enc(pk,R)

Can only leak R

(pk,sk) <- Gen

How to Implement Sampling Functionality

- Let Δ be a distribution that hardware samples, i.e. (L,R)←Δ



Can only leak L

Can only leak R

pk

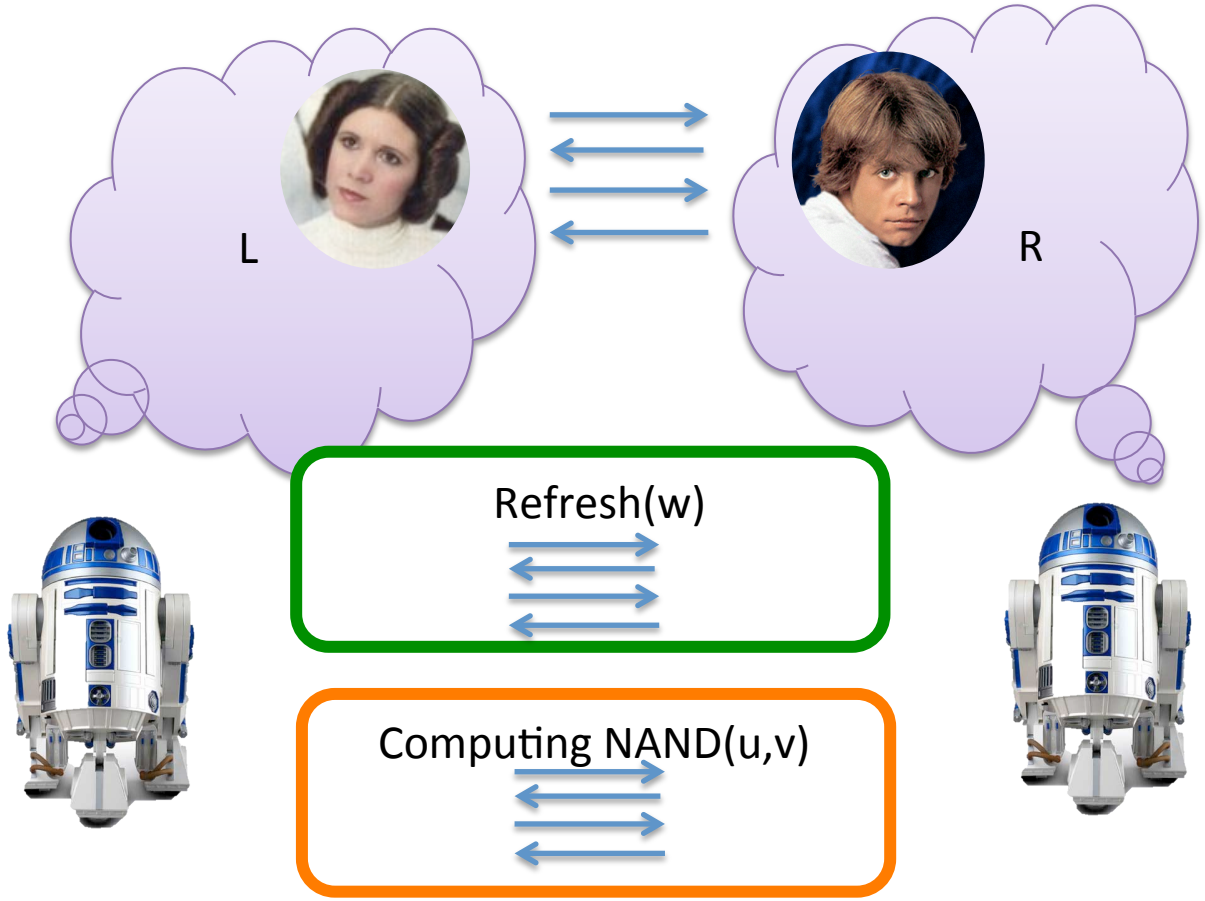
1. Sample r
2. Run Program C_{enc}(pk, r) and obtain (c_1,c_2) = (L, Enc(pk, R))

c_2 = Enc(pk,R)

(pk,sk) <- Gen

Final Scheme

- Replace hardware in DF by the protocol



Conclusion

- A generic design paradigm
  - Step1: design a hardware-based scheme
  - Step2: get rid of the hardware

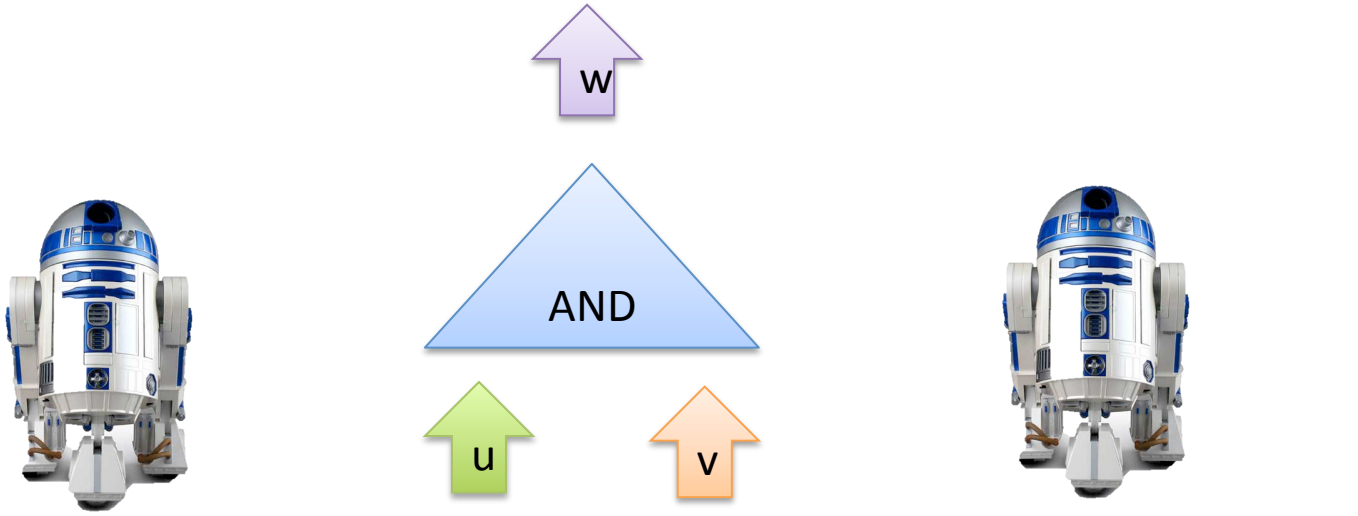
- New techniques to protect computation

Questions?

Thanks!

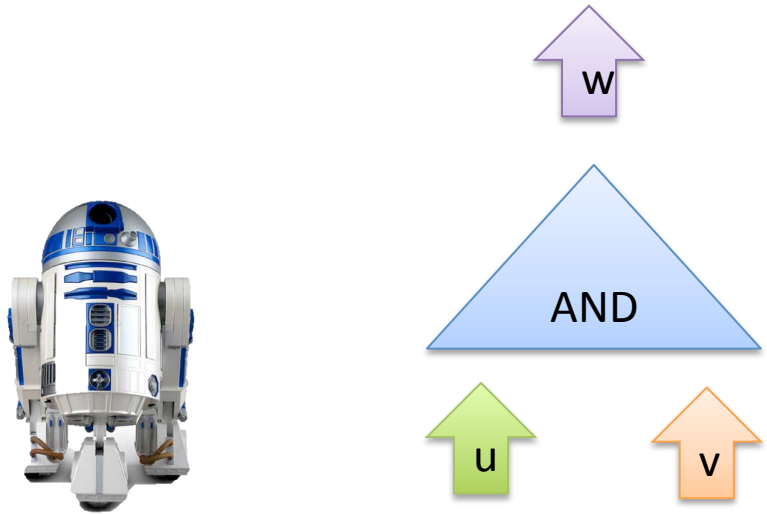
More Concretely

- The sharing is an inner product scheme
  - $u = (u_1, u_2)$ such that their inner product is u



More Concretely

- The sharing is an inner product scheme
  - $u = (u_1, u_2)$ such that their inner product is u
- Getting partial information of the shares does not leak the inner product



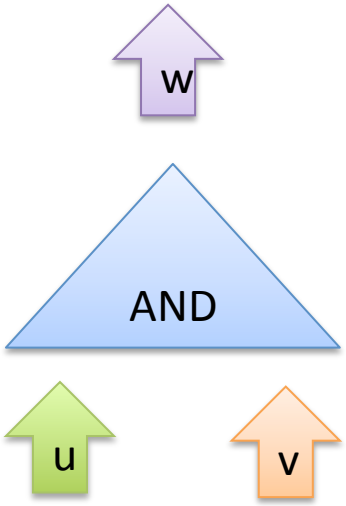
$u_1 = (110)$ $v_1 = (100)$ $u_2 = (111)$ $v_2 = (101)$

More Concretely

- The sharing is an inner product scheme
  - $u = (u_1, u_2)$ such that their inner product is u
- $\langle u_1 \otimes v_1, u_1 \otimes v_2 \rangle = \langle u_1, u_2 \rangle * \langle v_1, v_2 \rangle$

$w_1 = (100\ 100\ 000)$

$u_1 = (110)$ $v_1 = (100)$



$w_2 = (101\ 101\ 101)$

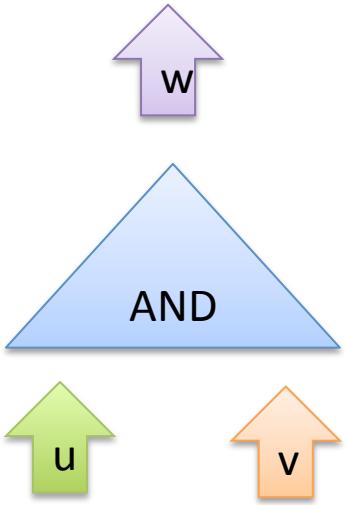
$u_2 = (111)$ $v_2 = (101)$

However…

- Dimension blows up…
- Shares of w are not fresh …

w_1 = (100 100 000)

u_1 = (110) v_1 = (100)



w_2 = (101 101 101)

u_2= (111) v_2= (101)