Shuffling Against Side-Channel Attacks:
a Comprehensive Study with Cautionary Note

N. Veyrat-Charvillon, M. Medwed,
S. Kerckhof, ***F.-X. Standaert***

UCL Crypto Group, Belgium

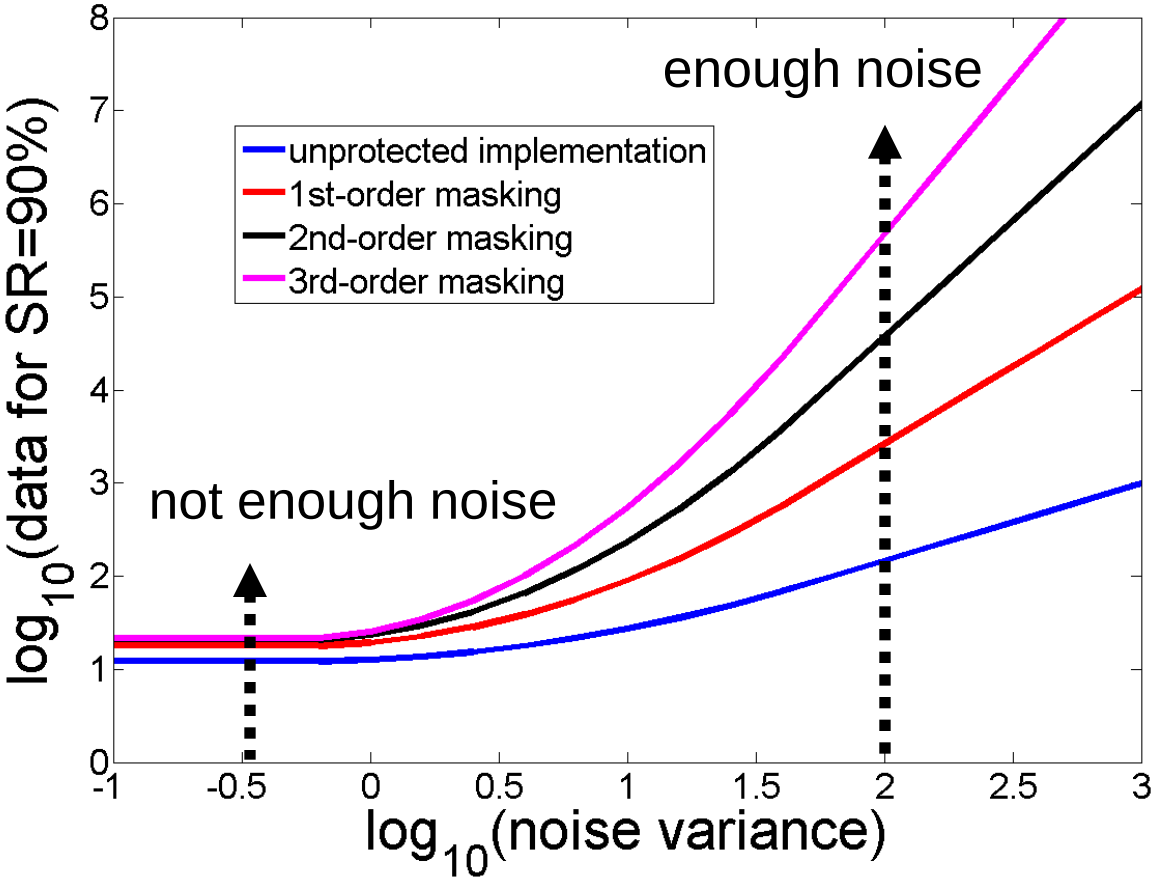
ASIACRYPT 2012, Bejing, China

Motivation (I)

- ASIACRYPT 2010: masking is an effective countermeasure against side-channel attacks

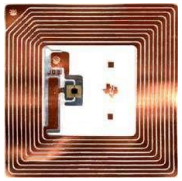
Motivation (I)

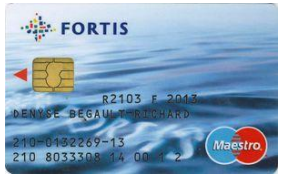
- ASIACRYPT 2010: masking is an effective countermeasure against side-channel attacks
  - *If there is sufficient measurement noise !!!*



Motivation (II)

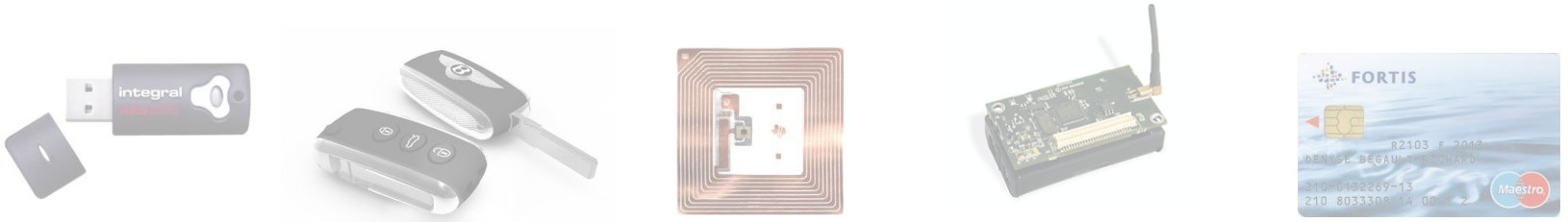
- Problem: how to generate noise in small devices?





Motivation (II)

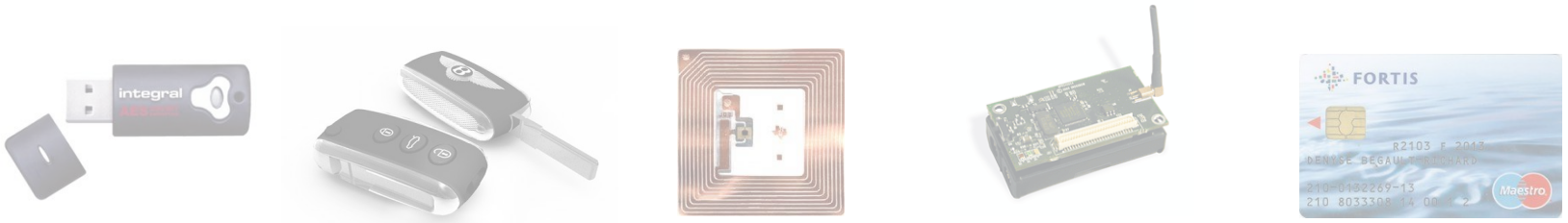
- Problem: how to generate noise in small devices?



- Previous works: time randomization can help

Motivation (II)

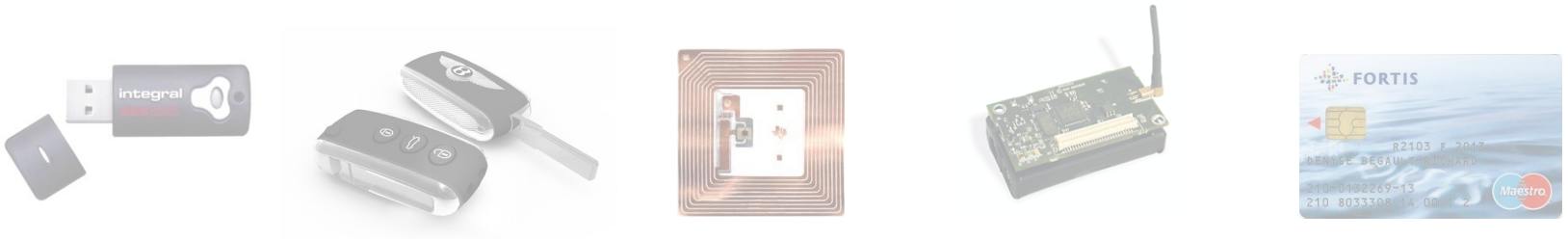
- Problem: how to generate noise in small devices?



- Previous works: time randomization can help
- But we need to be careful with re-synchronization
  - e.g. random insertion of dummy operations is not enough [Durvaux et al./CARDIS2012]

Motivation (II)

- Problem: how to generate noise in small devices?

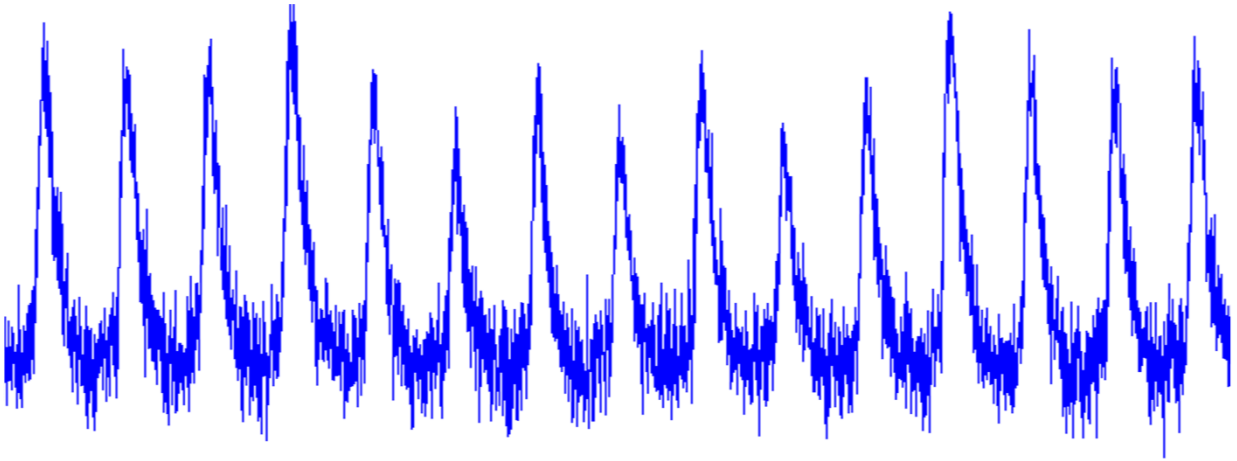


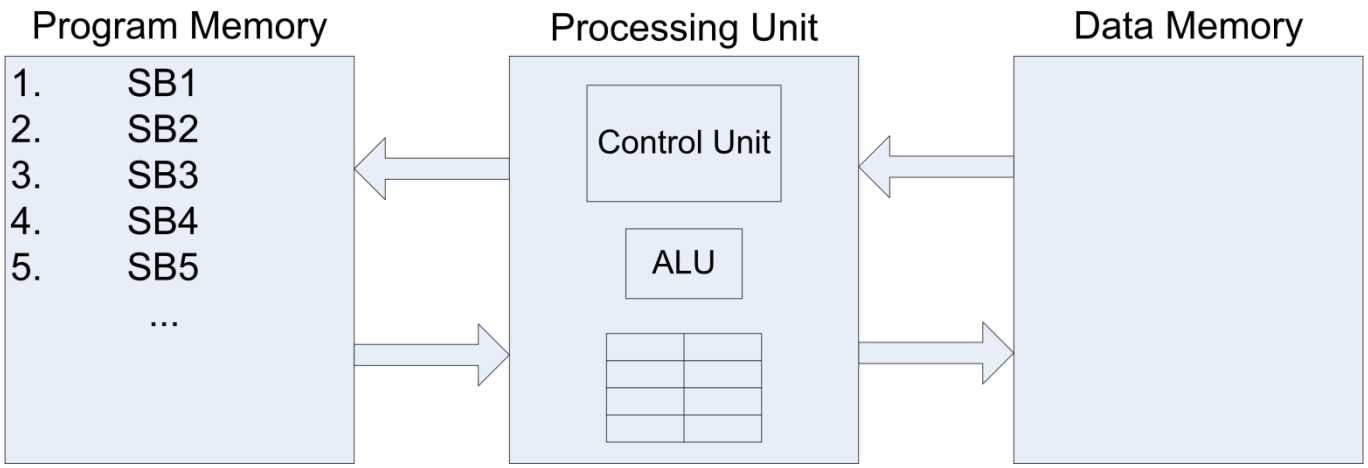
- Previous works: time randomization can help
- But we need to be careful with re-synchronization
  - e.g. random insertion of dummy operations is not enough [Durvaux et al./CARDIS2012]

- Best known option so far: shuffling operations

Intuition (I)

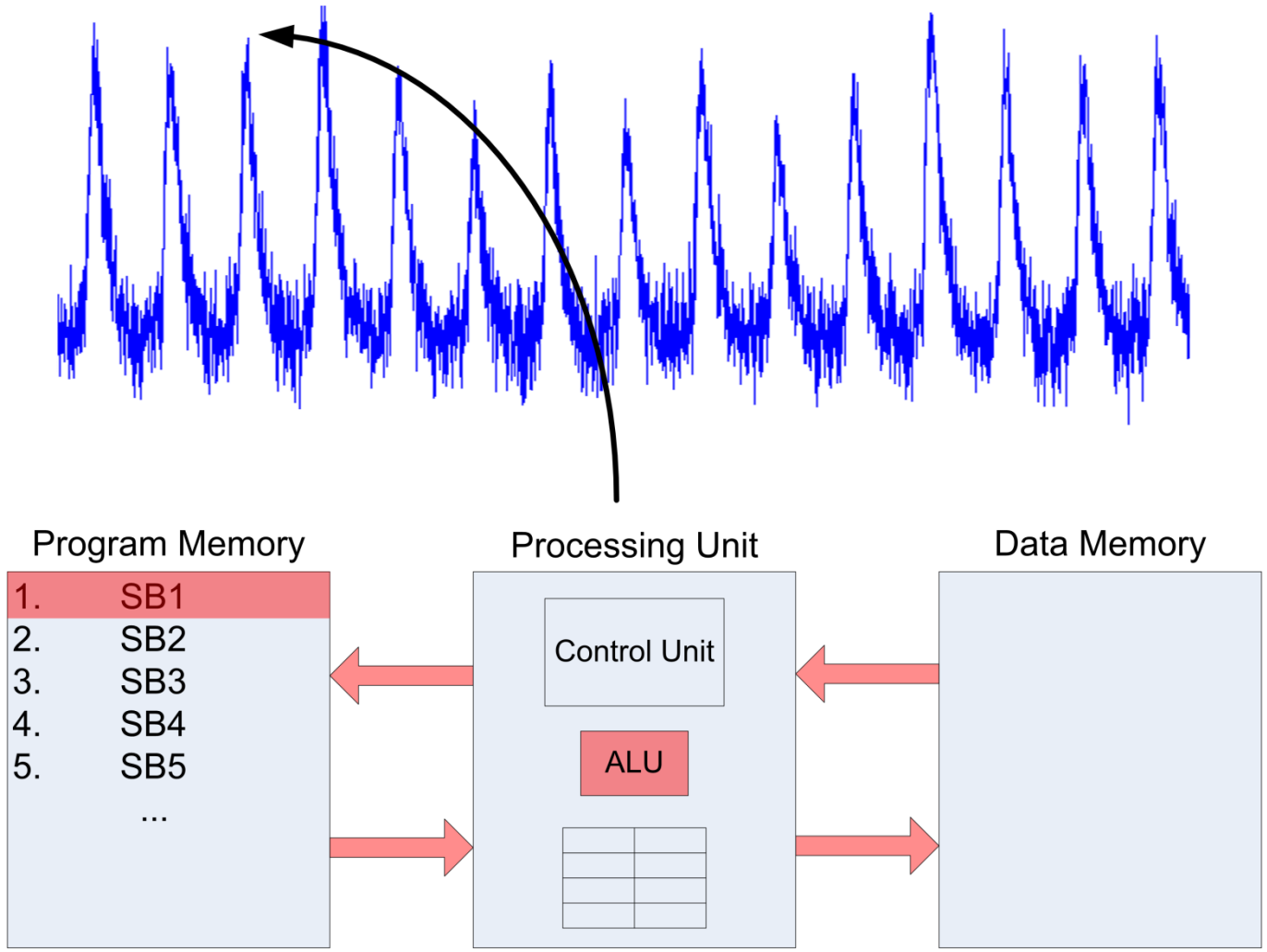
- Deterministic execution of the AES S-boxes





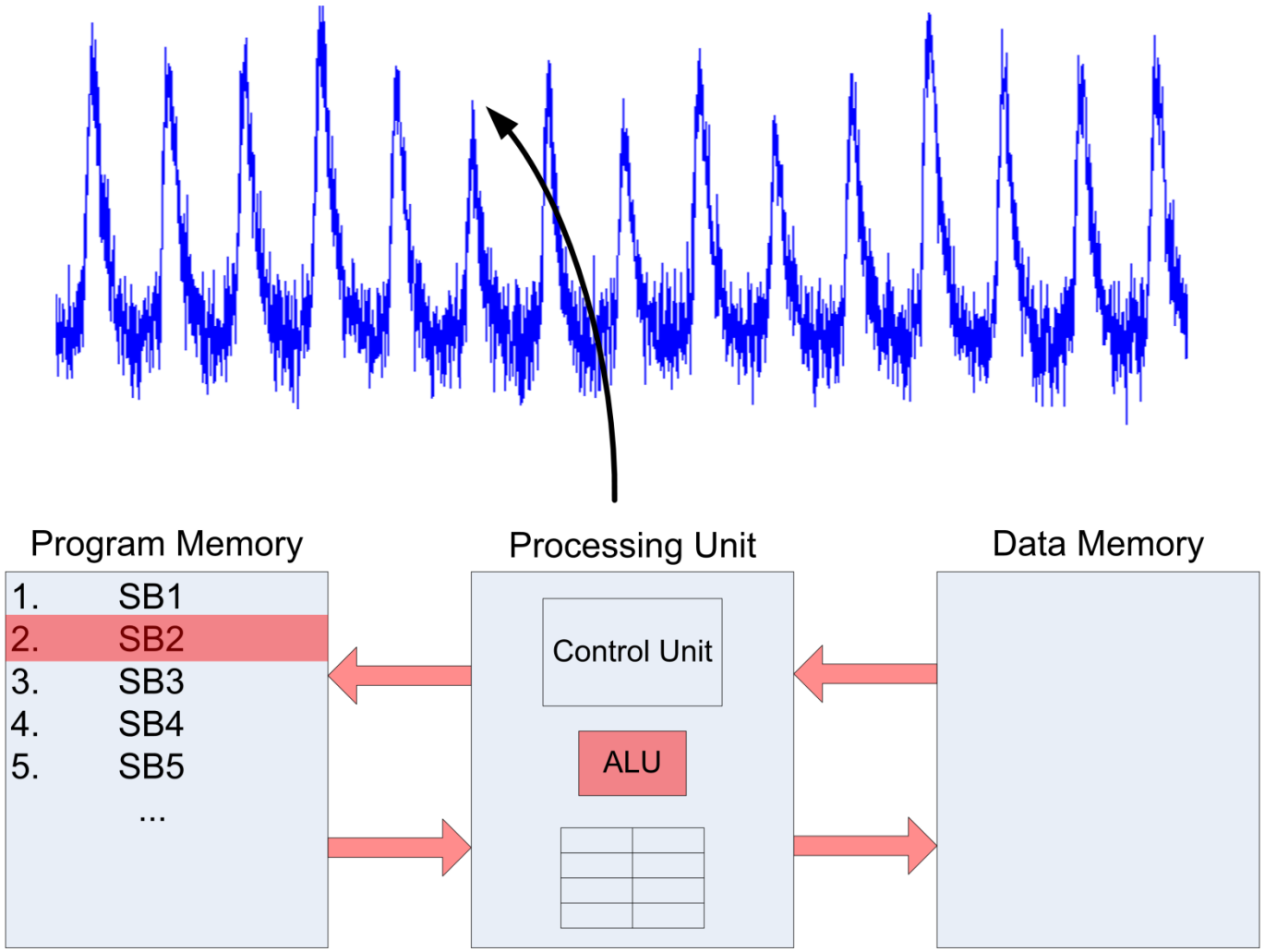
Intuition (I)

- Deterministic execution of the AES S-boxes



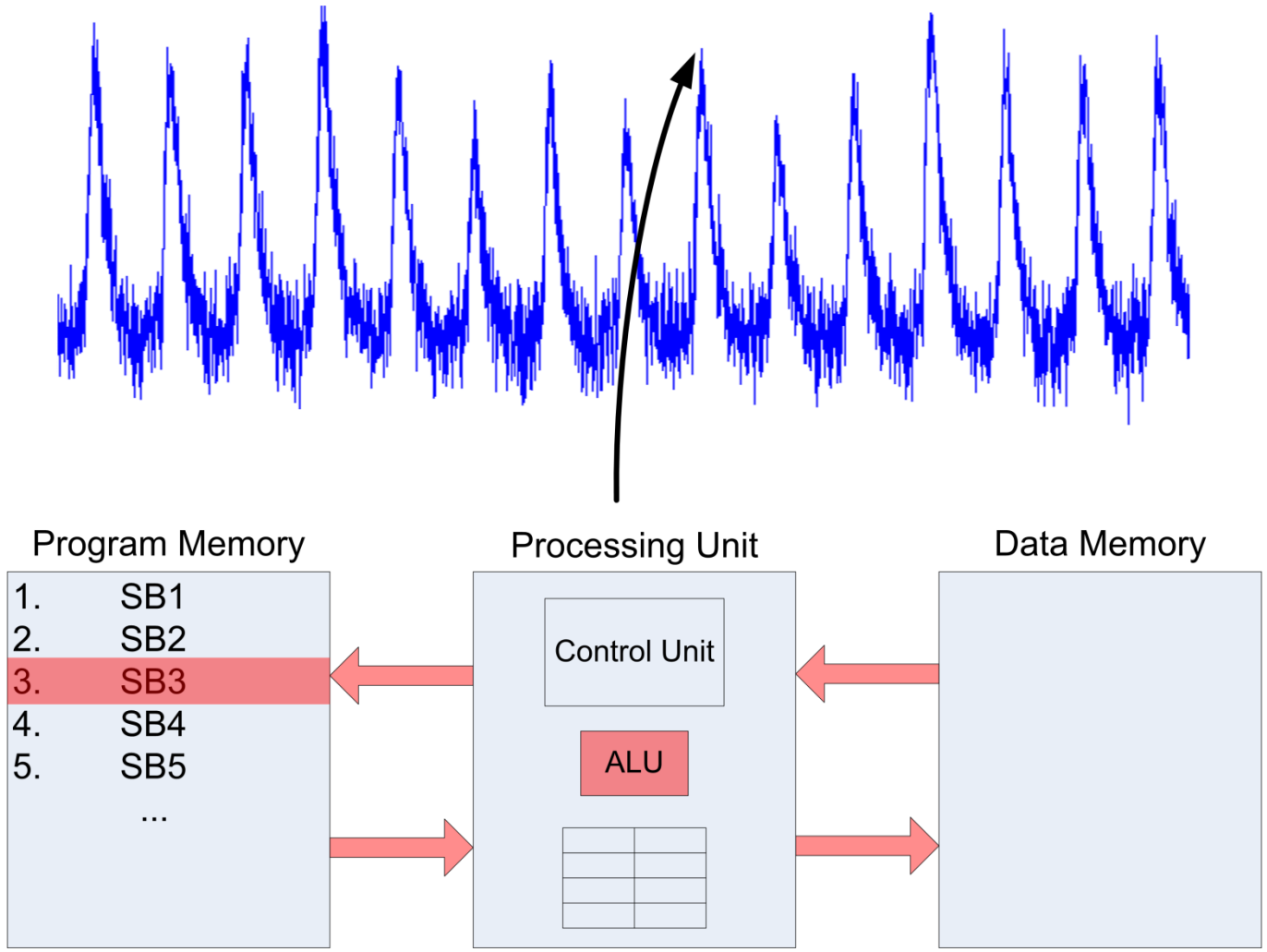
Intuition (I)

- Deterministic execution of the AES S-boxes



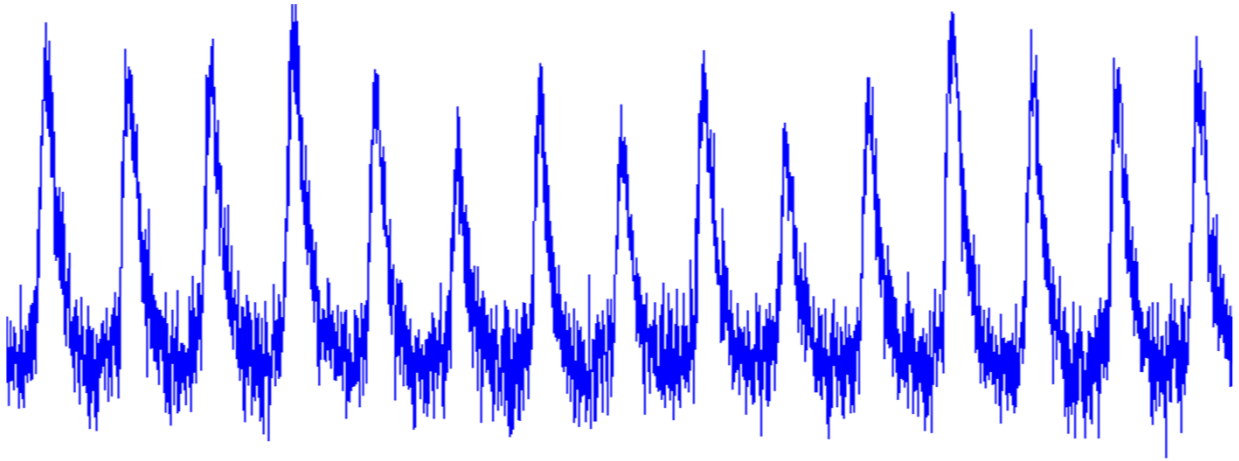
Intuition (I)

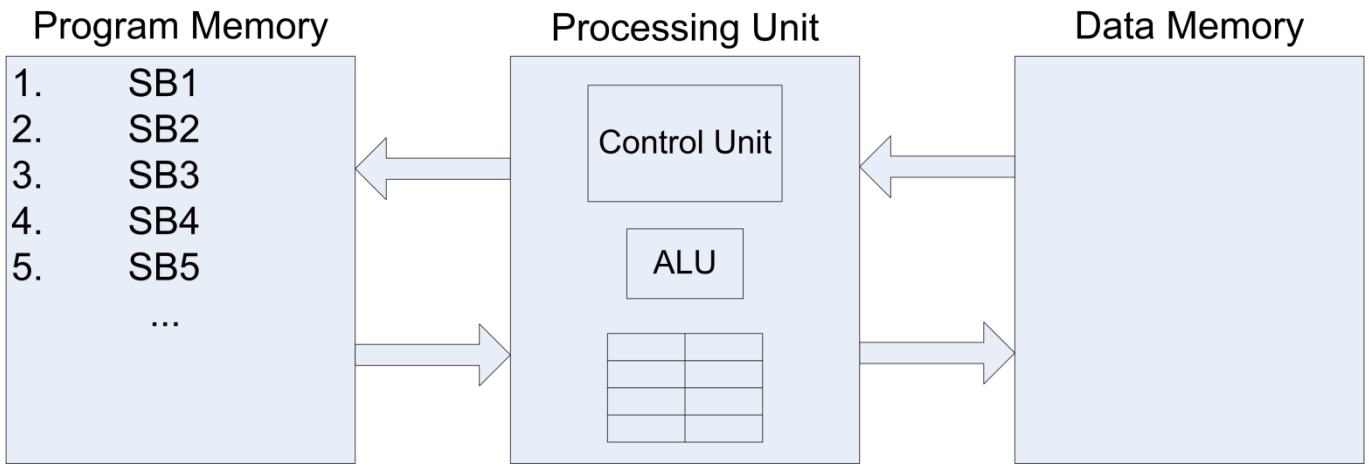
- Deterministic execution of the AES S-boxes



Intuition (II)

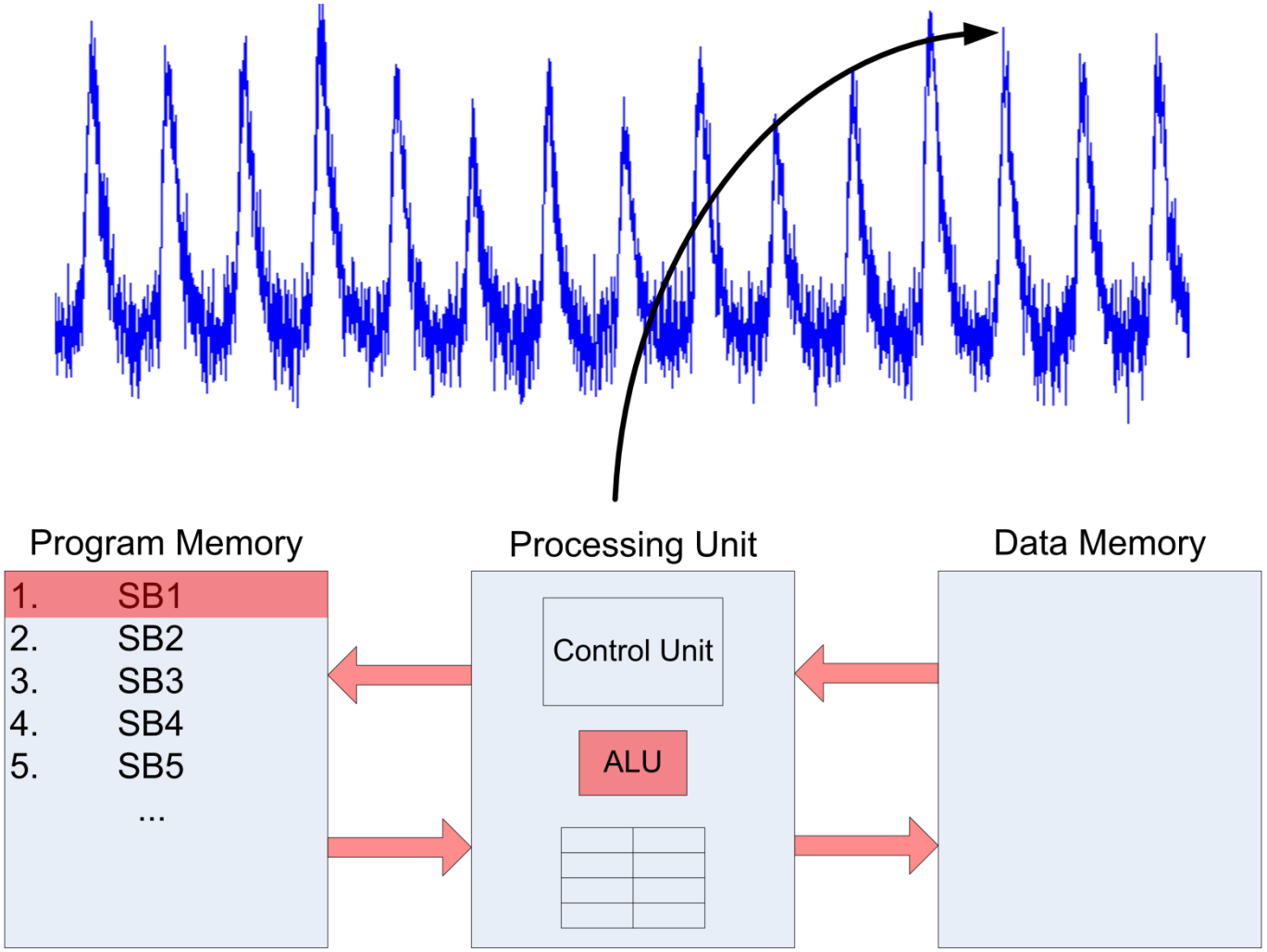
- Randomized execution of the AES S-boxes





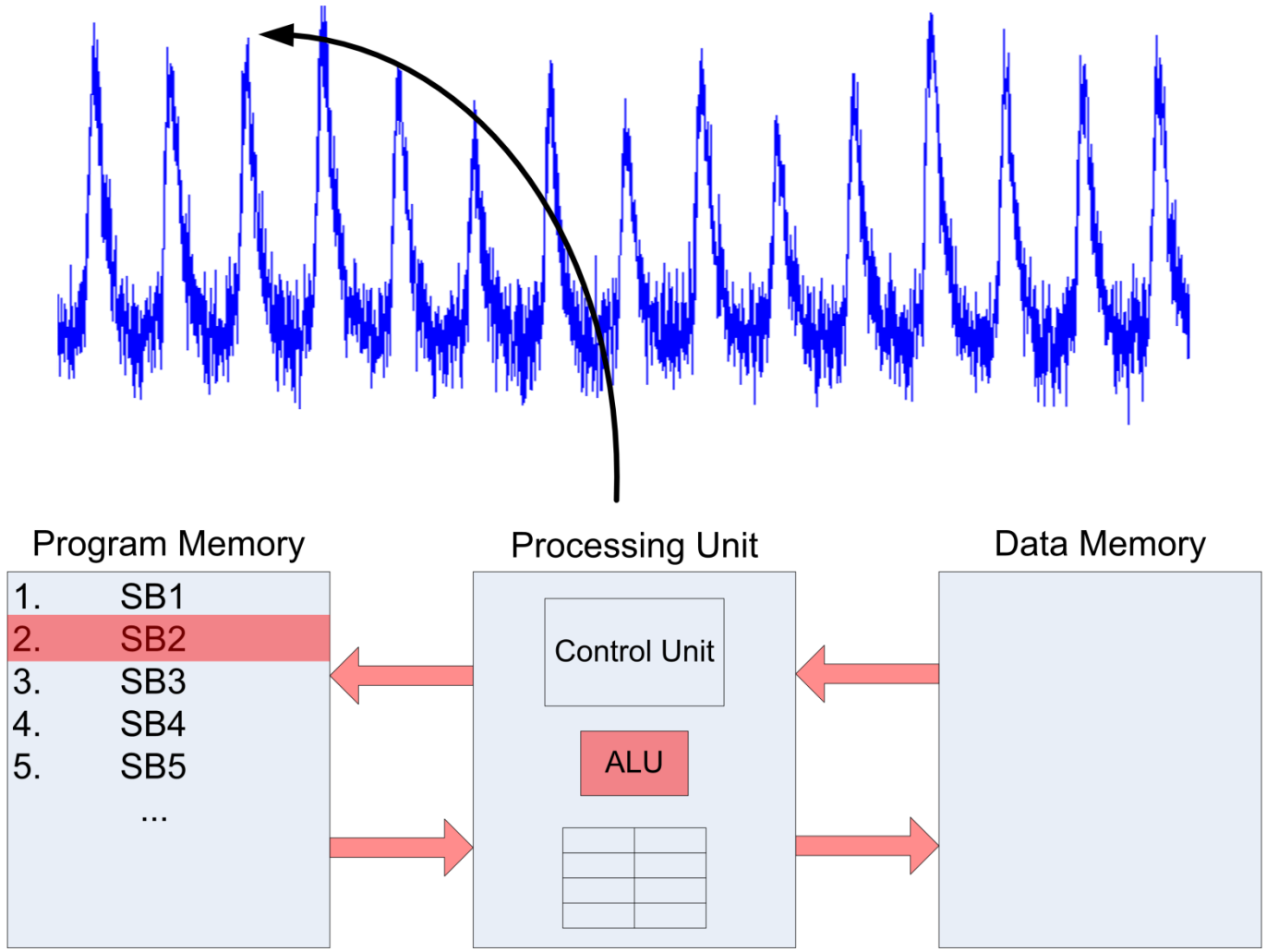
Intuition (II)

- Randomized execution of the AES S-boxes



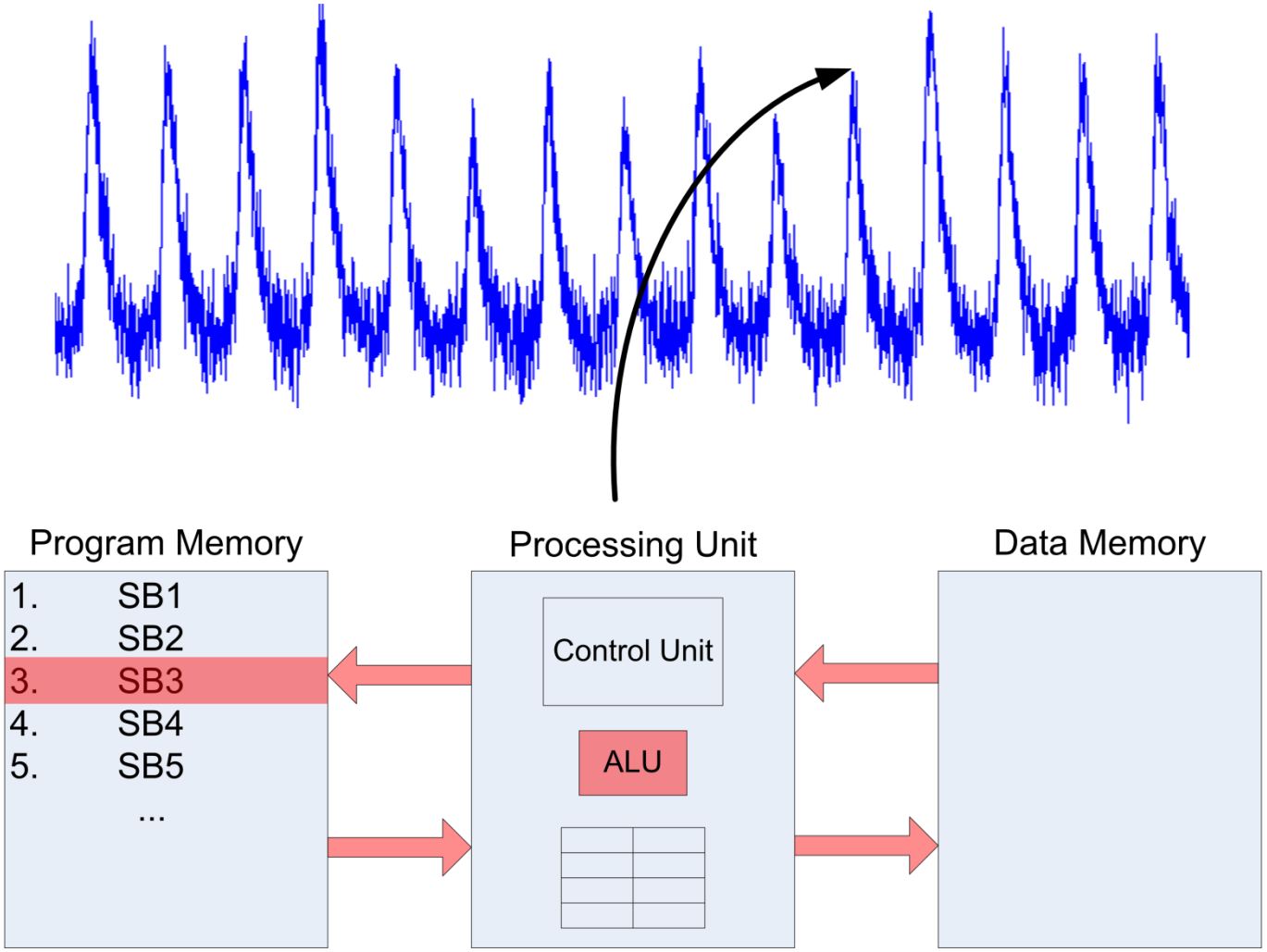
Intuition (II)

- Randomized execution of the AES S-boxes



Intuition (II)

- Randomized execution of the AES S-boxes



Expectation

- Points of interest spread over *t* cycles

Expectation

- Points of interest spread over *t* cycles

- Shuffling aims to amplify physical noise, by forcing the adversary to combine multiple points

Expectation

- Points of interest spread over *t* cycles
- Shuffling aims to amplify physical noise, by forcing the adversary to combine multiple points
- Easy solution to combine points is to “integrate”
  - Correlation between sensitive variable and actual leakage decreases by factor $\sqrt{t}$

Expectation

- Points of interest spread over t cycles
- Shuffling aims to amplify physical noise, by forcing the adversary to combine multiple points
- Easy solution to combine points is to "integrate"
  - Correlation between sensitive variable and actual leakage decreases by factor $\sqrt{t}$

$\Rightarrow$ Attack data complexity increases accordingly

Talk outline

1. Implementation: how to shuffle efficiently?
   (Permutation generation - see paper)
   a. Double indexing
   b. (new) Randomized execution path
   c. (new) Randomized program memory

2. Security evaluation: “better than integrating”
   a. Information theoretic analysis
   b. Actual chip measurements

Talk outline

1. Implementation: how to shuffle efficiently?
   (Permutation generation - see paper)
   a. Double indexing
   b. (new) Randomized execution path
   c. (new) Randomized program memory

2. Security evaluation: “better than integrating”
   a. Information theoretic analysis
   b. Actual chip measurements

Double indexing (S-box example)

Shuffled AES :
Program Memory

1. Load shuffle index from memory
2. Load state byte from memory
3. SB(state(index))
4. Store state byte in memory
5. Load shuffle index from memory
6. Load state byte from memory
7. SB(state(index))
8. Store state byte in memory

...

Data Memory

Permutation

1. 3
2. 15
3. 6

...

State

1. byte0
2. byte1
3. byte2
4. byte3

...

Double indexing (S-box example)

Shuffled AES :
Program Memory

1.	Load shuffle index from memory
2.	Load state byte from memory
3.	SB(state(index))
4.	Store state byte in memory
5.	Load shuffle index from memory
6.	Load state byte from memory
7.	SB(state(index))
8.	Store state byte in memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Double indexing (S-box example)

Shuffled AES :
Program Memory

1.	Load shuffle index from memory
2.	Load state byte from memory
3.	SB(state(index))
4.	Store state byte in memory
5.	Load shuffle index from memory
6.	Load state byte from memory
7.	SB(state(index))
8.	Store state byte in memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Double indexing (S-box example)

Shuffled AES :
Program Memory

```
1.    Load shuffle index from memory
2.    Load state byte from memory
3.    SB(state(index))
4.    Store state byte in memory
5.    Load shuffle index from memory
6.    Load state byte from memory
7.    SB(state(index))
8.    Store state byte in memory
        ...
```

Data Memory

Permutation

```
1.    3
2.    15
3.    6
      ...
```

State

```
1.    byte0
2.    byte1
3.    byte2
4.    byte3
      ...
```

Double indexing (S-box example)

Shuffled AES :
Program Memory

1.	Load shuffle index from memory
2.	Load state byte from memory
3.	SB(state(index))
4.	Store state byte in memory
5.	Load shuffle index from memory
6.	Load state byte from memory
7.	SB(state(index))
8.	Store state byte in memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Double indexing (S-box example)

Shuffled AES :
Program Memory

1.	Load shuffle index from memory
2.	Load state byte from memory
3.	SB(state(index))
4.	Store state byte in memory
5.	Load shuffle index from memory
6.	Load state byte from memory
7.	SB(state(index))
8.	Store state byte in memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Double indexing (S-box example)

Shuffled AES :
Program Memory

1.	Load shuffle index from memory
2.	Load state byte from memory
3.	SB(state(index))
4.	Store state byte in memory
5.	Load shuffle index from memory
6.	Load state byte from memory
7.	SB(state(index))
8.	Store state byte in memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

- Two RAM accesses per R/W access to operands
  $\Rightarrow$ Very large cycle counts

Talk outline

1. Implementation: how to shuffle efficiently?
   (Permutation generation - see paper)
   a. Double indexing
   b. (new) Randomized execution path
   c. (new) Randomized program memory

2. Security evaluation: “better than integrating”
   a. Information theoretic analysis
   b. Actual chip measurements

Randomized execution path

Shuffled AES :
Program Memory

1.	Generate shuffled address array
2.	Find next instruction
3.	Load state byte from memory
4.	SB0
5.	Find next instruction
6.	Load state byte from memory
7.	SB1
	...

Data Memory

Permutation

1.	1
2.	15
3.	0
	...

Address array

1.	6
2.	45
3.	3
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Generate shuffled address array
2.	Find next instruction
3.	Load state byte from memory
4.	SB0
5.	Find next instruction
6.	Load state byte from memory
7.	SB1
	...

Data Memory

Permutation

1.	1
2.	15
3.	0
	...

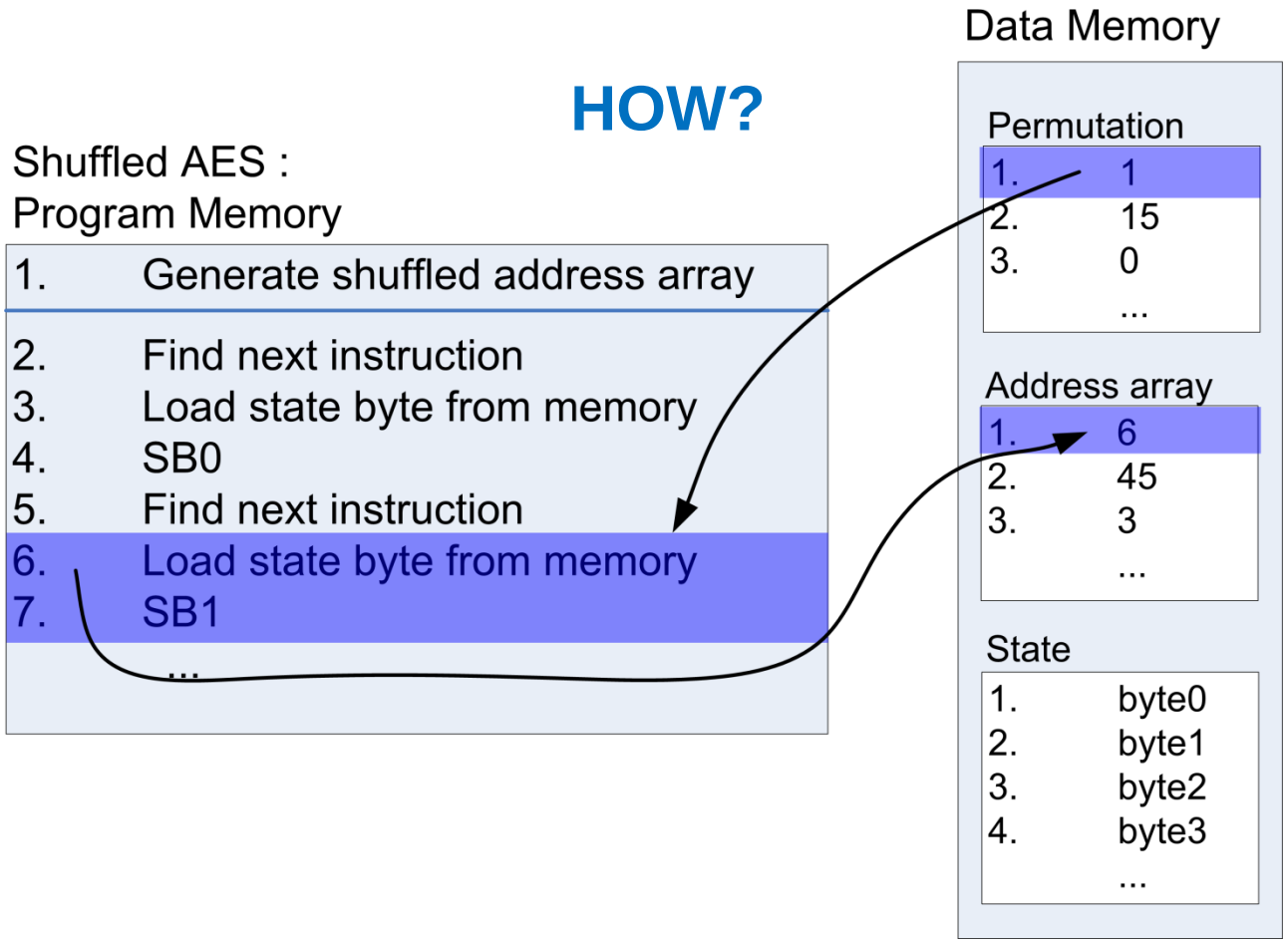
Address array

1.	6
2.	45
3.	3
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path



Randomized execution path

Shuffled AES :
Program Memory

1.	Generate shuffled address array
2.	Find next instruction
3.	Load state byte from memory
4.	SB0
5.	Find next instruction
6.	Load state byte from memory
7.	SB1
	...

Data Memory

Permutation

1.	1
2.	15
3.	0
	...

Address array

1.	6
2.	45
3.	3
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Generate shuffled address array
2.	Find next instruction
3.	Load state byte from memory
4.	SB0
5.	Find next instruction
6.	Load state byte from memory
7.	SB1
	...

Data Memory

Permutation

1.	1
2.	15
3.	0
	...

Address array

1.	6
2.	45
3.	3
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Generate shuffled address array
2.	Find next instruction
3.	Load state byte from memory
4.	SB0
5.	Find next instruction
6.	Load state byte from memory
7.	SB1
	...

Data Memory

Permutation

1.	1
2.	15
3.	0
	...

Address array

1.	6
2.	45
3.	3
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Generate shuffled address array
2.	Find next instruction
3.	Load state byte from memory
4.	SB0
5.	Find next instruction
6.	Load state byte from memory
7.	SB1
	...

Data Memory

Permutation

1.	1
2.	15
3.	0
	...

Address array

1.	6
2.	45
3.	3
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

- Better exploitation of registers
  $\Rightarrow$ Significantly reduces the cycle count

Talk outline

1. Implementation: how to shuffle efficiently?
   (Permutation generation - see paper)
   a. Double indexing
   b. (new) Randomized execution path
   c. (new) Randomized program memory

2. Security evaluation: “better than integrating”
   a. Information theoretic analysis
   b. Actual chip measurements

Randomized execution path

Shuffled AES :
Program Memory

1.	Shuffle program memory
2.	Load state byte from memory
3.	SB0
4.	Load state byte from memory
5.	SB1
6.	Load state byte from memory
7.	SB2
8.	Load state byte from memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Shuffle program memory
2.	Load state byte from memory
3.	SB0
4.	Load state byte from memory
5.	SB1
6.	Load state byte from memory
7.	SB2
8.	Load state byte from memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Shuffle program memory
2.	Load state byte from memory
3.	SB3
4.	Load state byte from memory
5.	SB15
6.	Load state byte from memory
7.	SB6
8.	Load state byte from memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Shuffle program memory
2.	Load state byte from memory
3.	SB3
4.	Load state byte from memory
5.	SB15
6.	Load state byte from memory
7.	SB6
8.	Load state byte from memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Shuffle program memory
2.	Load state byte from memory
3.	SB3
4.	Load state byte from memory
5.	SB15
6.	Load state byte from memory
7.	SB6
8.	Load state byte from memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

Randomized execution path

Shuffled AES :
Program Memory

1.	Shuffle program memory
2.	Load state byte from memory
3.	SB3
4.	Load state byte from memory
5.	SB15
6.	Load state byte from memory
7.	SB6
8.	Load state byte from memory
	...

Data Memory

Permutation

1.	3
2.	15
3.	6
	...

State

1.	byte0
2.	byte1
3.	byte2
4.	byte3
	...

- Large precomputations (technology dependent)
  - Somehow similar to a one-time program
- Minimum “online” cycle count (~ unprotected)

Evaluations on Atmel AtMega644p

- Includes shuffling of other AES operations (among 16 for each of them – see paper for the details)

Implementation	Clock Cycles
Unprotected AES	2739 (3546 with KS)
Double Indexing shuffling	30202 (46395 with KS)
Randomized Path shuffling	6934 (14834 with KS)
Randomized Memory shuffling	3299*

** Excludes precomputations (approx. 18 milliseconds)*

Talk outline

1. Implementation: how to shuffle efficiently?
   (Permutation generation - see paper)
   a. Double indexing
   b. (new) Randomized execution path
   c. (new) Randomized program memory

2. Security evaluation: “better than integrating”
   a. Information theoretic analysis
   b. Actual chip measurements

Better than integrating attacks?

- Security evaluations of masking schemes usually exploit the leakage of all the shares

Better than integrating attacks?

- Security evaluations of masking schemes usually exploit the leakage of all the shares
- Integrating attacks ignore “permutation leakages”
  - (Which appear within the double indexing and randomized execution path methods)

Better than integrating attacks?

- Security evaluations of masking schemes usually exploit the leakage of all the shares
- Integrating attacks ignore “permutation leakages”
  - (Which appear within the double indexing and randomized execution path methods)

$\Rightarrow$ Natural next step: include them in analysis

Our model

- Standard (sensitive operation) leakages

$$L_0 \leftsquigarrow \mathsf{Sbox}(k_{\mathsf{P}(0)} \oplus X_{\mathsf{P}(0)}), \qquad L_1 \leftsquigarrow \mathsf{Sbox}(k_{\mathsf{P}(1)} \oplus X_{\mathsf{P}(1)}),$$
$$L_2 \leftsquigarrow \mathsf{Sbox}(k_{\mathsf{P}(2)} \oplus X_{\mathsf{P}(2)}), \qquad \ldots$$

Our model

- Standard (sensitive operation) leakages

$$L_0 \leftsquigarrow \mathsf{Sbox}(k_{\mathsf{P}(0)} \oplus X_{\mathsf{P}(0)}), \qquad L_1 \leftsquigarrow \mathsf{Sbox}(k_{\mathsf{P}(1)} \oplus X_{\mathsf{P}(1)}),$$
$$L_2 \leftsquigarrow \mathsf{Sbox}(k_{\mathsf{P}(2)} \oplus X_{\mathsf{P}(2)}), \qquad \ldots$$

- Let S*c* = P(*c*) be the part of the master key manipulated at cycle *c*, with P the shuffling perm.
- We additionally consider permutations leakages

$$L'_c \leftsquigarrow S_c$$

EC09 information theoretic analysis (I)

- Based on the leakage probability distribution

$$\Pr[\mathbf{L} = \mathbf{l} | K_s = k] = \sum_c \frac{\mathsf{f}(c, s, \mathbf{l}')}{\sum_{c'} \mathsf{f}(c', s, \mathbf{l}')} \Pr[L_c = l_c | K_s = k]$$

EC09 information theoretic analysis (I)

- Based on the leakage probability distribution

$$\Pr[\mathbf{L} = \mathbf{l} | K_s = k] = \sum_c \frac{\mathsf{f}(c, s, \mathbf{l}')}{\sum_{c'} \mathsf{f}(c', s, \mathbf{l}')} \Pr[L_c = l_c | K_s = k]$$

- Different scenarios
  - Template attack without permutation leakage (UNI-TA)

$$\mathsf{f}(c, s, \mathbf{l}') = 1/16$$

EC09 information theoretic analysis (I)

- Based on the leakage probability distribution

$$\Pr[\mathbf{L} = \mathbf{l} | K_s = k] = \sum_c \frac{\mathrm{f}(c, s, \mathbf{l}')}{\sum_{c'} \mathrm{f}(c', s, \mathbf{l}')} \Pr[L_c = l_c | K_s = k]$$

- Different scenarios
  - Template attack without permutation leakage (UNI-TA)

$$\mathrm{f}(c, s, \mathbf{l}') = 1/16$$

  - Template attack with permutation leakage (DPLEAK-TA)

$$\mathrm{f}(c, s, \mathbf{l}') = \Pr[L'_c = l'_c | S_c = s]$$

EC09 information theoretic analysis (I)

- Based on the leakage probability distribution

$$\Pr[\mathbf{L} = \mathbf{l} | K_s = k] = \sum_c \frac{\mathrm{f}(c, s, \mathbf{l}')}{\sum_{c'} \mathrm{f}(c', s, \mathbf{l}')} \Pr[L_c = l_c | K_s = k]$$

- Different scenarios
  - Template attack without permutation leakage (UNI-TA)

$$\mathrm{f}(c, s, \mathbf{l}') = 1/16$$

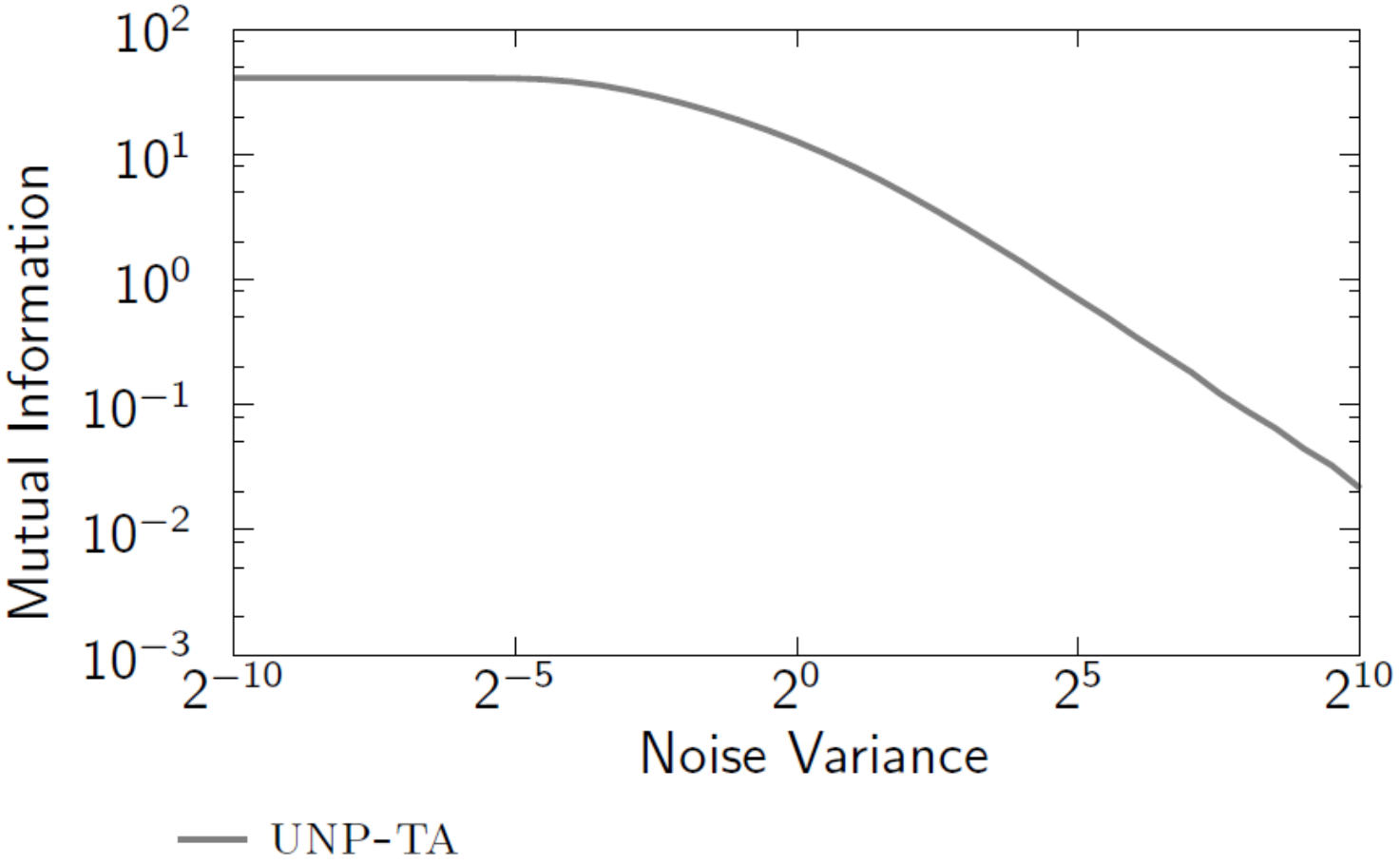
  - Template attack with permutation leakage (DPLEAK-TA)

$$\mathrm{f}(c, s, \mathbf{l}') = \Pr[L'_c = l'_c | S_c = s]$$

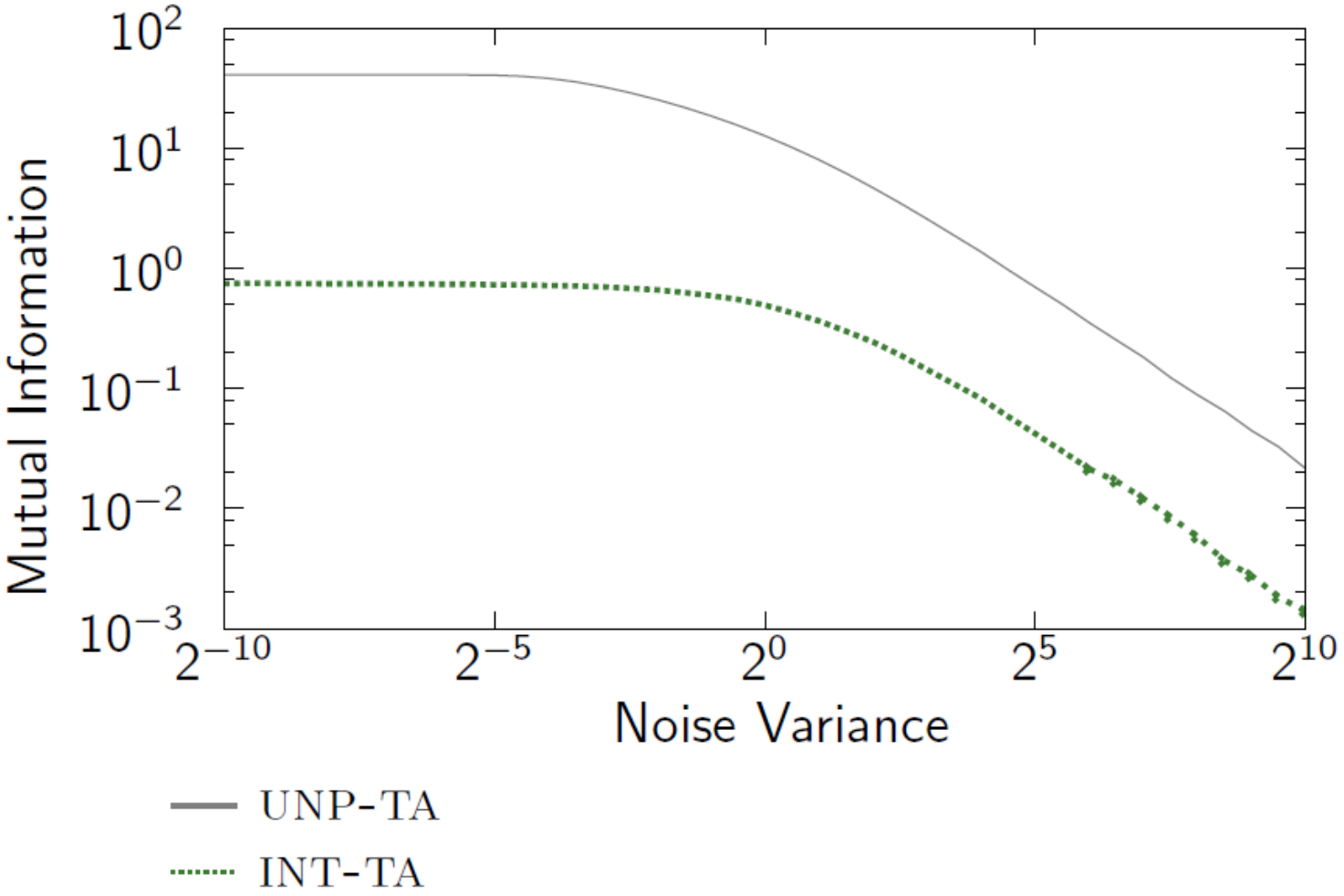
  - Random Start Index (RSIENUM-TA)

$$\mathrm{f}(c, s, \mathbf{l}') = \prod_{i=0}^{15} \Pr[L'_i = l'_i | S_i = (s - c + i) \bmod 16]$$

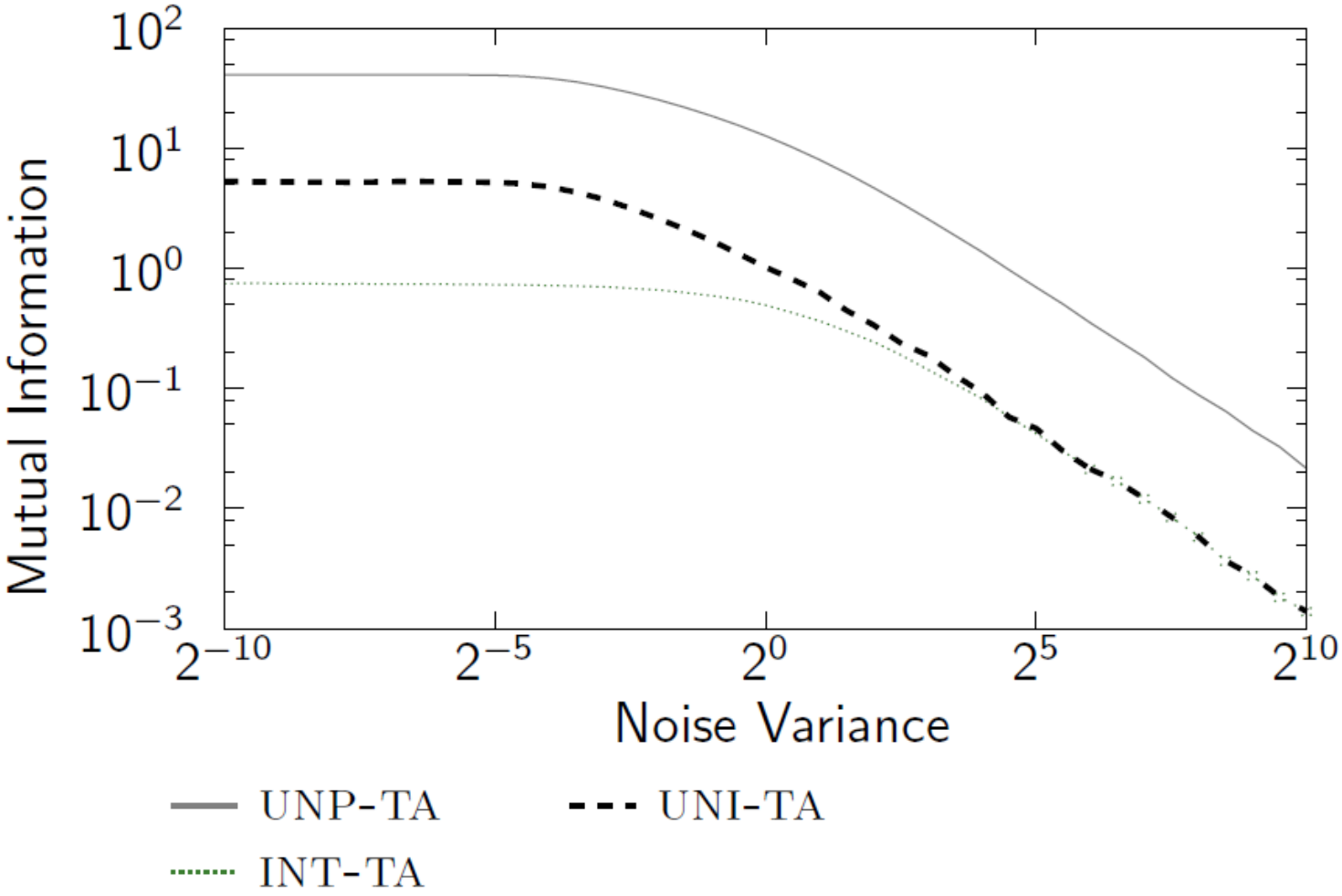
EC09 information theoretic analysis (II)



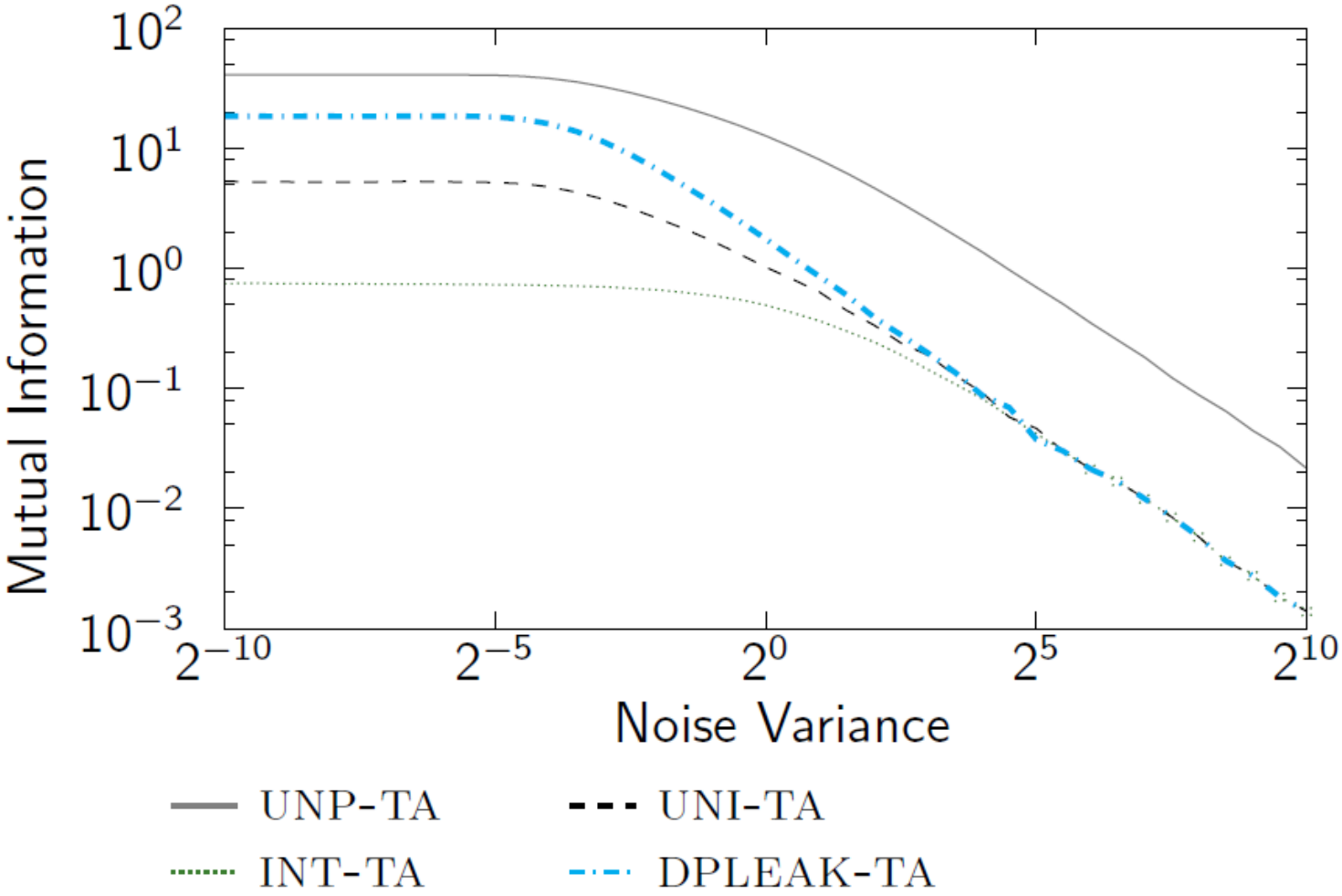
EC09 information theoretic analysis (II)



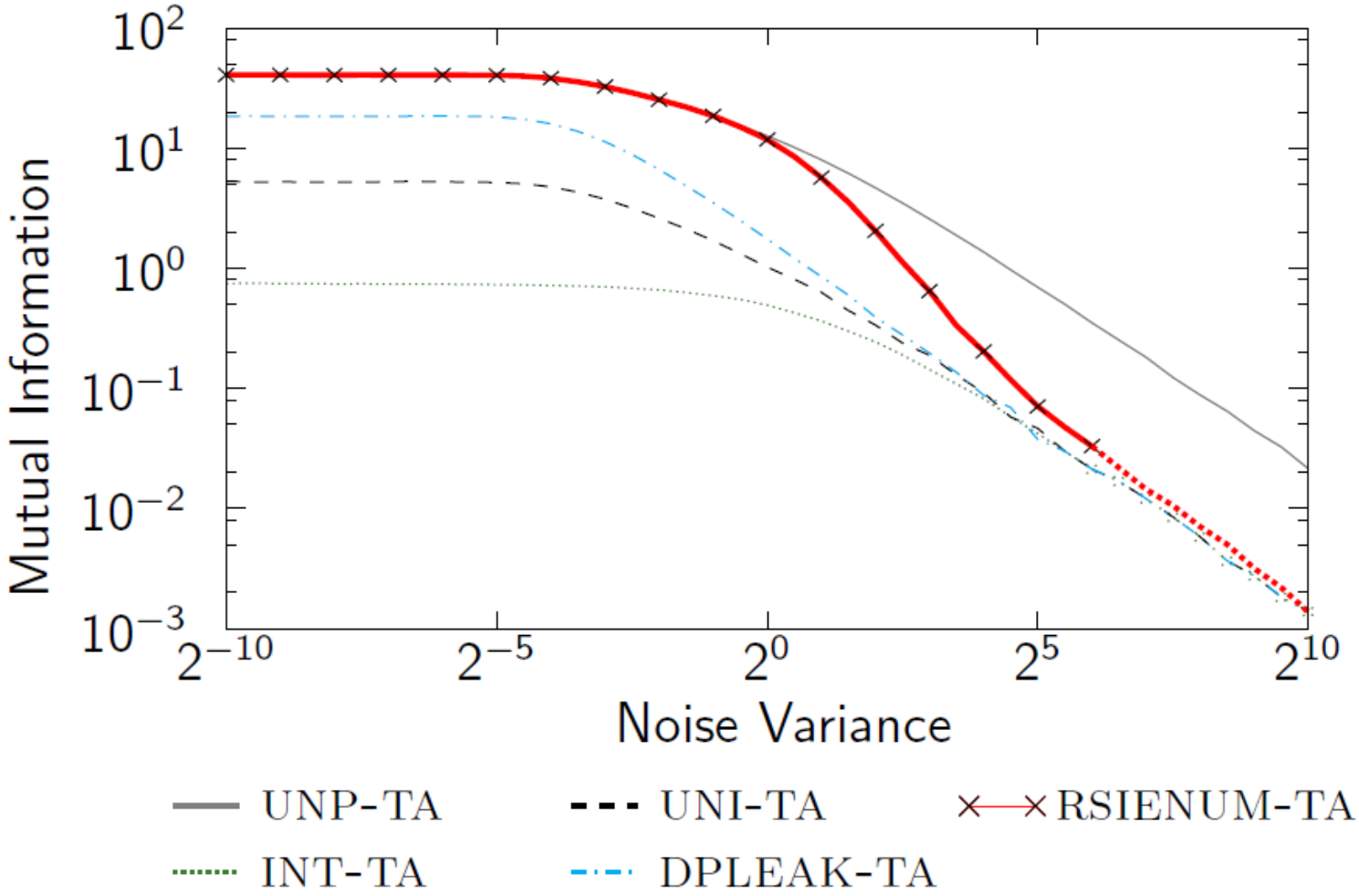
EC09 information theoretic analysis (II)



EC09 information theoretic analysis (II)



EC09 information theoretic analysis (II)

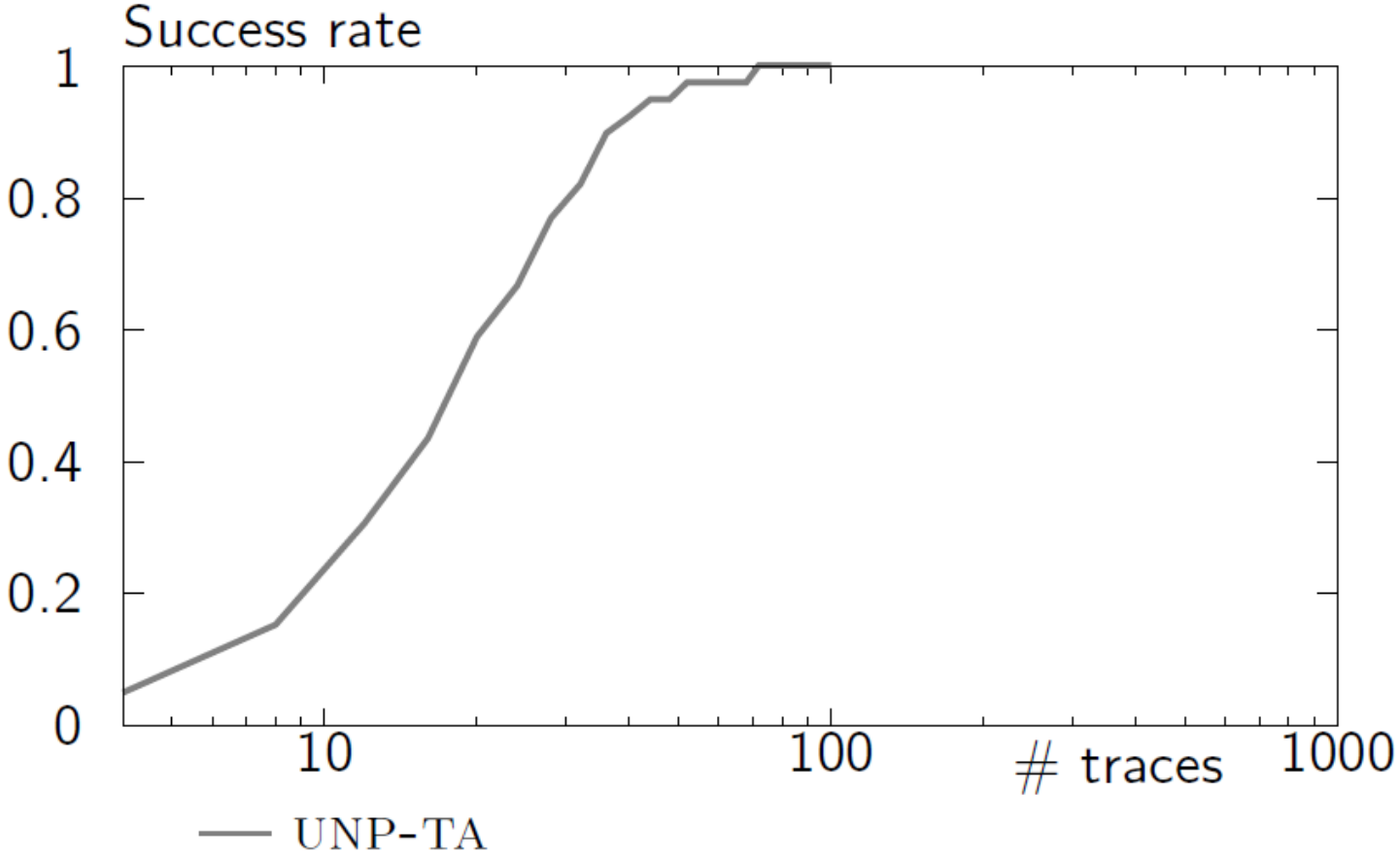


Talk outline

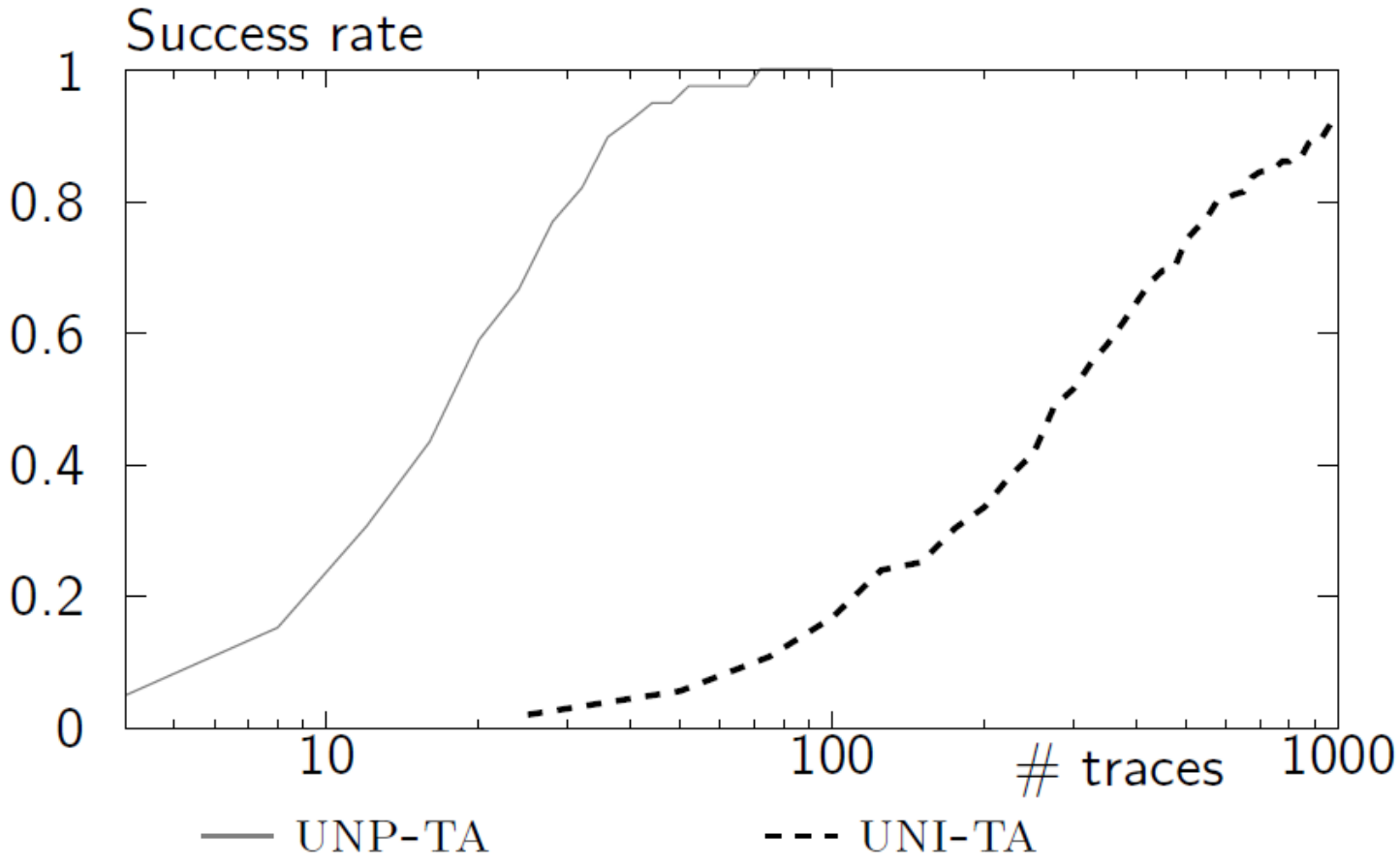
1. Implementation: how to shuffle efficiently?
   (Permutation generation - see paper)
   a. Double indexing
   b. (new) Randomized execution path
   c. (new) Randomized program memory

2. Security evaluation: “better than integrating”
   a. Information theoretic analysis
   b. Actual chip measurements

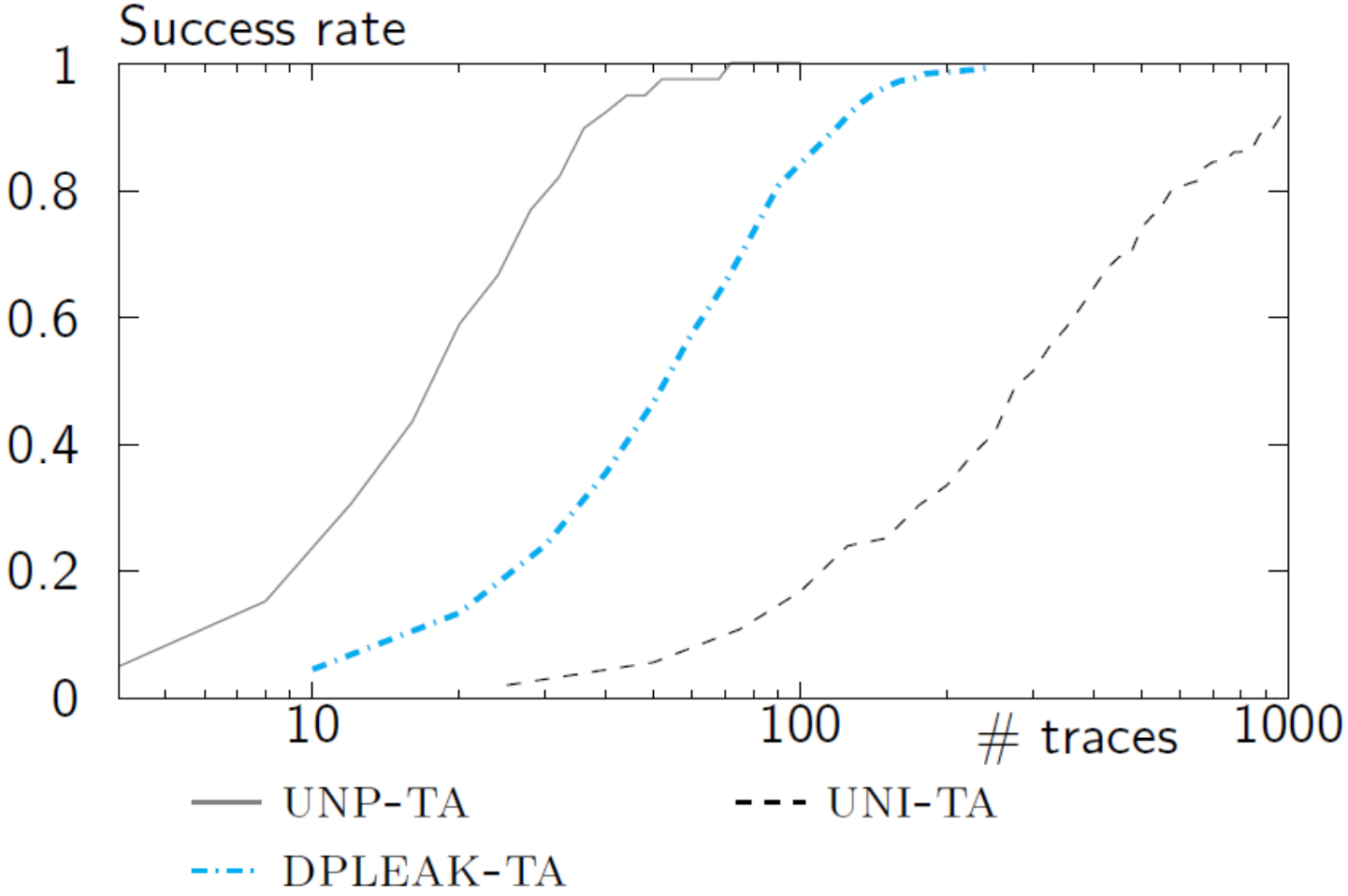
EC09 security analysis (Atmel AVR)



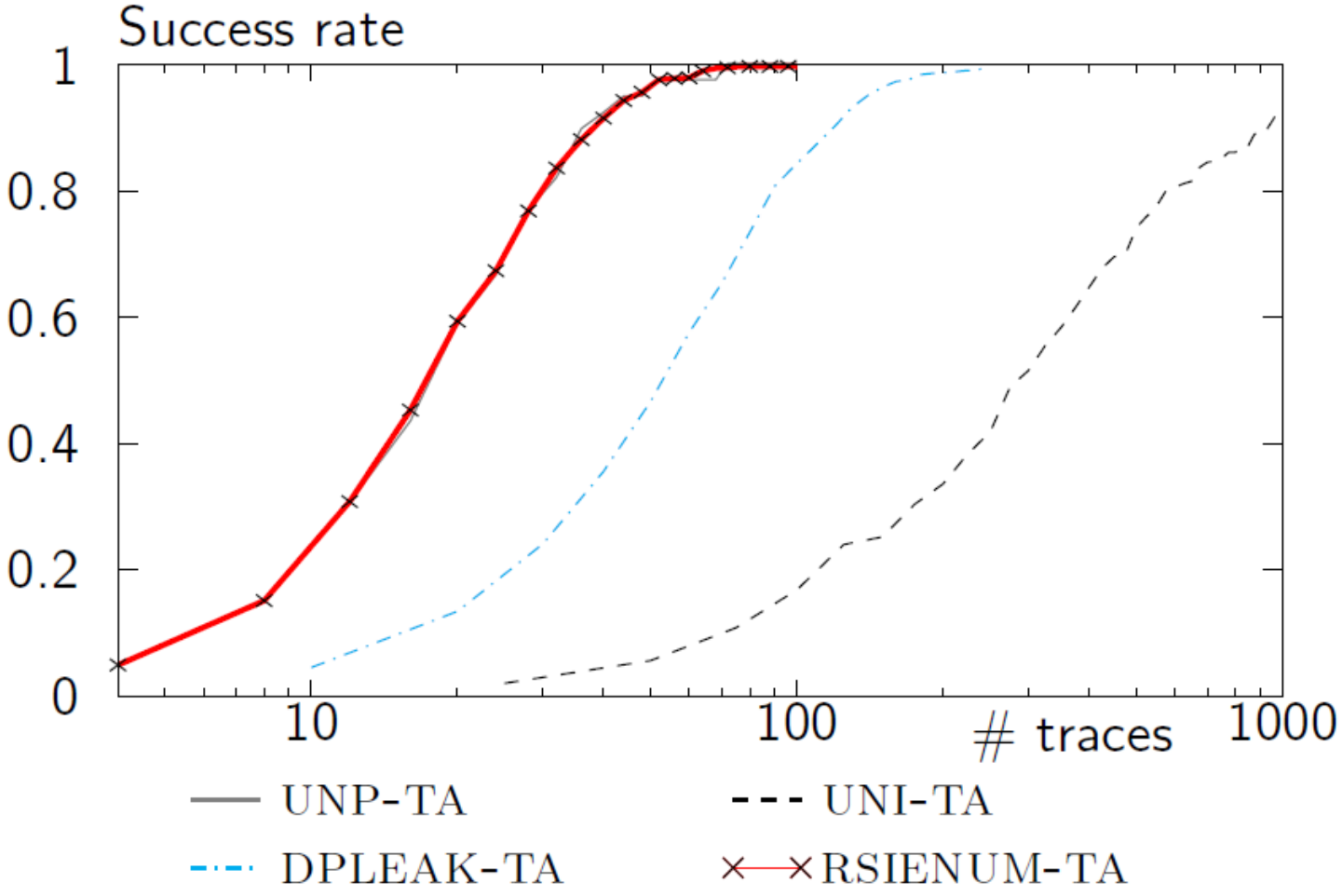
EC09 security analysis (Atmel AVR)



EC09 security analysis (Atmel AVR)



EC09 security analysis (Atmel AVR)



Indirect leakages!

- **Fact:** for similar amounts of noise, actual attacks more efficient than simulated ones

Indirect leakages!

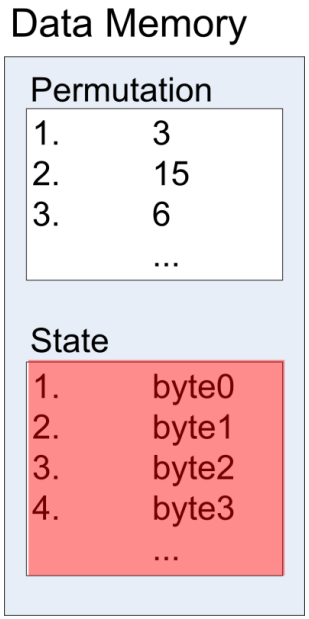
- **Fact:** for similar amounts of noise, actual attacks more efficient than simulated ones
  - Surprising since the opposite is usually observed (because simulated attacks have *perfect leakage models* while actual attacks use *estimated ones*)

Indirect leakages!

- **Fact:** for similar amounts of noise, actual attacks more efficient than simulated ones
  - Surprising since the opposite is usually observed (because simulated attacks have *perfect leakage models* while actual attacks use *estimated ones*)
- Why?

Indirect leakages!

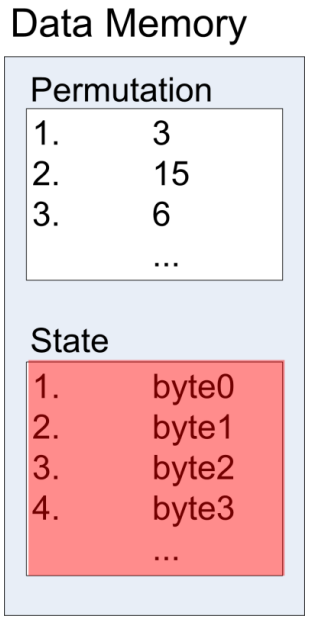
- **Fact:** for similar amounts of noise, actual attacks more efficient than simulated ones
  - Surprising since the opposite is usually observed (because simulated attacks have *perfect leakage models* while actual attacks use *estimated ones*)
- Why? Recall the processor structure:



Indirect leakages!

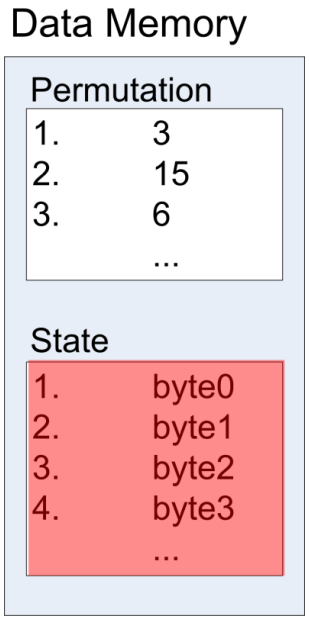
- **Fact:** for similar amounts of noise, actual attacks more efficient than simulated ones
  - Surprising since the opposite is usually observed (because simulated attacks have *perfect leakage models* while actual attacks use *estimated ones*)

- Why? Recall the processor structure:
  - Even if operation order is shuffled, the state bytes are still manipulated by the same hardware resources!



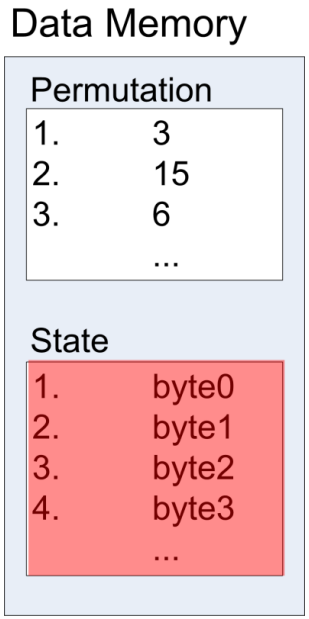
Indirect leakages!

- **Fact:** for similar amounts of noise, actual attacks more efficient than simulated ones
  - Surprising since the opposite is usually observed (because simulated attacks have *perfect leakage models* while actual attacks use *estimated ones*)
- Why? Recall the processor structure:
  - Even if operation order is shuffled, the state bytes are still manipulated by the same hardware resources!
    - As these resources have slightly different leakage models, this leads to additional (indirect) leakage!



Indirect leakages!

- **Fact:** for similar amounts of noise, actual attacks more efficient than simulated ones
  - Surprising since the opposite is usually observed (because simulated attacks have *perfect leakage models* while actual attacks use *estimated ones*)
- Why? Recall the processor structure:
  - Even if operation order is shuffled, the state bytes are still manipulated by the same hardware resources!
    - As these resources have slightly different leakage models, this leads to additional (indirect) leakage!
    - *Even for randomized program memory !!*



Conclusions

- Shuffling can be implemented in different ways
  - Tradeoff between “online” and “offline” cycles

Conclusions

- Shuffling can be implemented in different ways
  - Tradeoff between “online” and “offline” cycles
- Cautionary note: shuffling can be used as a noise amplifier – never as a noise generator!

Conclusions

- Shuffling can be implemented in different ways
  - Tradeoff between “online” and “offline” cycles
- Cautionary note: shuffling can be used as a noise amplifier – never as a noise generator!
- Computation matters: enumerable permutations lead to easier attacks => RSI should be avoided
  - (and all the operations in the block cipher should be permuted over at least 16!)

Conclusions

- Shuffling can be implemented in different ways
  - Tradeoff between “online” and “offline” cycles
- Cautionary note: shuffling can be used as a noise amplifier – never as a noise generator!
- Computation matters: enumerable permutations lead to easier attacks => RSI should be avoided
  - (and all the operations in the block cipher should be permuted over at least 16!)
- Indirect leakages can appear!
  - Ideally, not only the order of operations should be shuffled, but also the resources used

THANKS

http://perso.uclouvain.be/fstandae/