Single-Trace Attacks on Keccak

Matthias J. Kannwischer[1], Peter Pessl[2], Robert Primas[3]

[1]Radboud University, Nijmegen
[2]Graz University of Technology (now with Infineon Technologies)
[3]Graz University of Technology

Side-Channel Attacks on Hash Functions?

- Plain hashing has no secrets, but there are keyed uses
  - HMAC? Classic DPA setting, threat is obvious...
- Keccak (SHA3/SHAKE) found ample new uses involving secrets
  - ...especially in post-quantum cryptography

Side-Channel Attacks on Hash Functions?

- Keccak uses in PQC include
  - derivation of a shared secret in a KEM
  - expansion of a secret seed in KEMs and signatures
  - hash-based signatures
- Above: side-channel attacker is limited to a single execution
  - at most averaging, but still no DPA

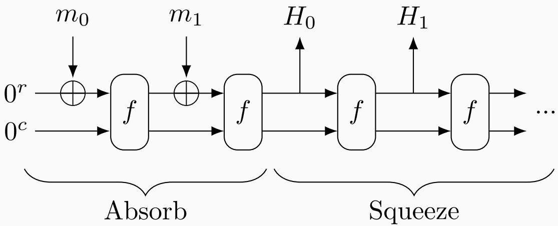
Are attacks even possible? Are countermeasures still needed?

Our Contribution

- Practical single-trace attack on Keccak (software) implementations
- Soft-analytical side-channel attack (SASCA)
    1. Template matching: retrieve probabilities of intermediates
    2. Belief propagation: combine all probabilities to infer most likely key
    - thus far: mainly applied to AES, but Keccak structurally very different
- Attack outcome
    - key-recovery in a large array of settings, countermeasures cannot be omitted
    - factors influencing the success rate:
      key size, bit width of device, structure of input

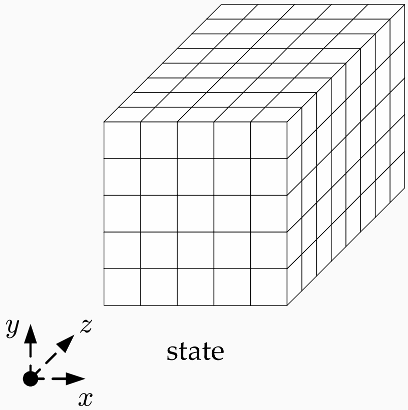
Keccak

- Sponge construction, 1600-bit state



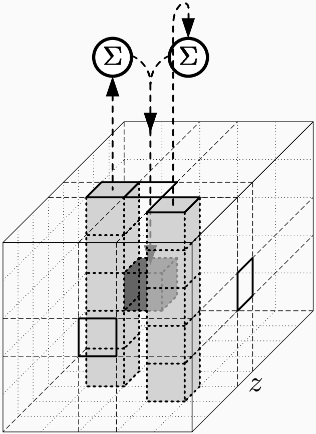
Keccak

- Sponge construction, 1600-bit state
- Keccak-f permutation



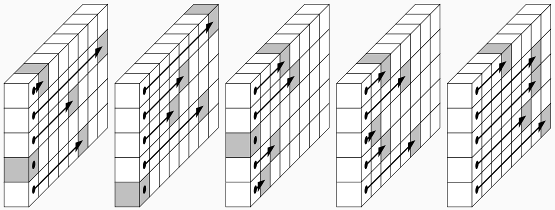
Keccak

- Sponge construction, 1600-bit state
- Keccak-f permutation
  - θ - add column parities



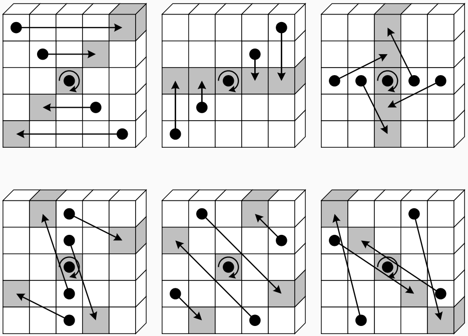
Keccak

- Sponge construction, 1600-bit state
- Keccak-f permutation
  - θ - add column parities
  - ρ - rotate lanes



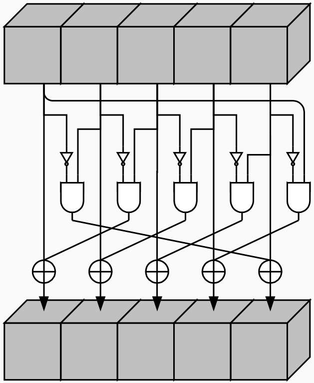
Keccak

- Sponge construction, 1600-bit state
- Keccak-f permutation
  - θ - add column parities
  - ρ - rotate lanes
  - π - reorder lanes



Keccak

- Sponge construction, 1600-bit state
- Keccak-f permutation
  - θ - add column parities
  - ρ - rotate lanes
  - π - reorder lanes
  - χ - SBox

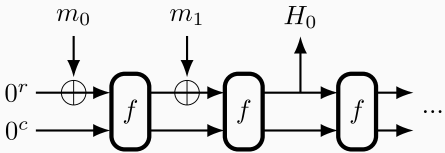


Keccak

- Sponge construction, 1600-bit state
- Keccak-f permutation
    - θ - add column parities
    - ρ - rotate lanes
    - π - reorder lanes
    - χ - SBox
    - ι - add round constant

Attack Setting

- Unprotected software implementation on a μC
- (Part of) the input is secret
  - and used only once
- Power measurements of a single execution
  - no differential SCA
  - have to use (some sort of) templates



Template Attacks on Hash Functions

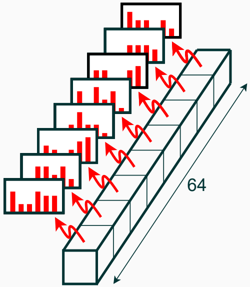
- Typical restrictions of template attacks
  - need templating device with known key
  - poor portability of templates between devices
- Same for Keccak?
  - often multiple calls inside a PK scheme, some with fully known data
  - message hash during signing, re-encryption in decapsulation, ...

Profiling directly on target device!

no separate profiling device needed, no portability problems

Step 1: Template Matching

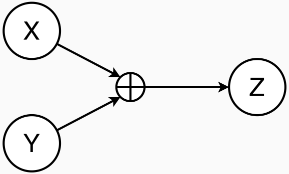
- Templating target: all loads/stores
  - HW leakage along lanes
  - assign probability vector to each part



- Now: combine all side channel info to find most likely key
  - efficient method: Soft Analytical Side-Channel Attacks (SASCA) [Veyrat-Charvillon et al., ASIACRYPT 2014]

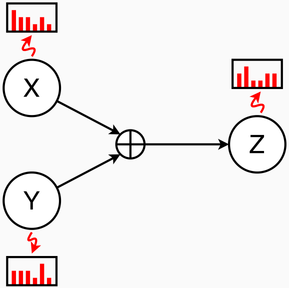
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$



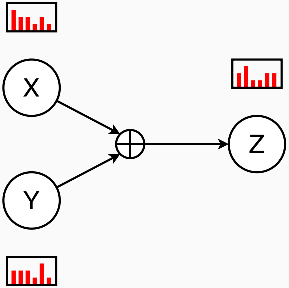
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$
2. incorporate leakage information in graph



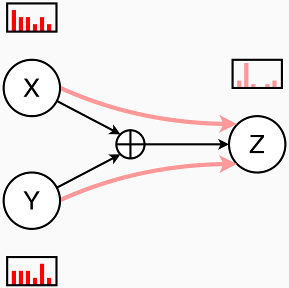
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$
2. incorporate leakage information in graph
3. run *Belief Propagation*
   - goal: find marginals of variables
   - message passing principle
   - simplest version: enumerate inputs
   - important: avoid circular reasoning



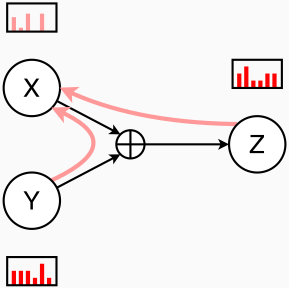
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$
2. incorporate leakage information in graph
3. run *Belief Propagation*
   - goal: find marginals of variables
   - message passing principle
   - simplest version: enumerate inputs
   - important: avoid circular reasoning



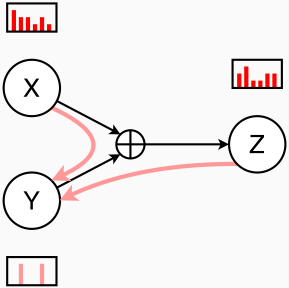
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$
2. incorporate leakage information in graph
3. run *Belief Propagation*
   - goal: find marginals of variables
   - message passing principle
   - simplest version: enumerate inputs
   - important: avoid circular reasoning



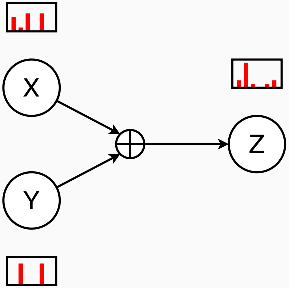
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$
2. incorporate leakage information in graph
3. run *Belief Propagation*
   - goal: find marginals of variables
   - message passing principle
   - simplest version: enumerate inputs
   - important: avoid circular reasoning



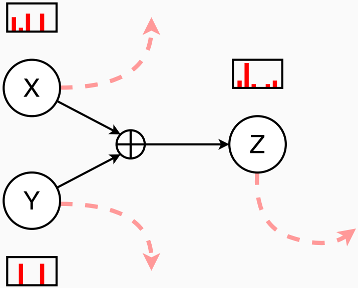
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$
2. incorporate leakage information in graph
3. run *Belief Propagation*
   - goal: find marginals of variables
   - message passing principle
   - simplest version: enumerate inputs
   - important: avoid circular reasoning



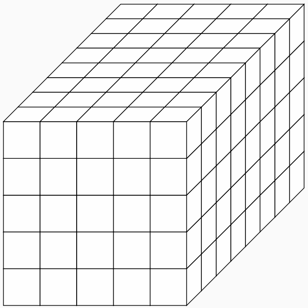
Step 2: SASCA / Belief Propagation

1. model implementation as a *factor graph*
   - variable nodes
   - factor nodes
   - example: $X \oplus Y = Z$
2. incorporate leakage information in graph
3. run *Belief Propagation*
   - goal: find marginals of variables
   - message passing principle
   - simplest version: enumerate inputs
   - important: avoid circular reasoning



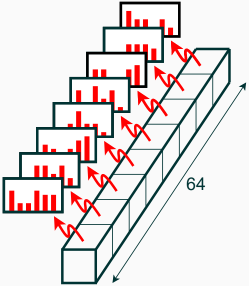
A First Factor Graph of Keccak

- Bitwise description
  - each bit after each step is a variable
- Terrible performance. . .
  - leakage on bytes/words, not bits
  - lots of information lost during propagation



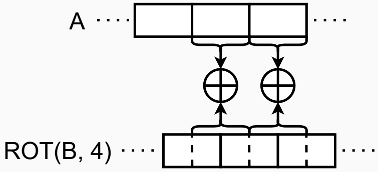
Solution: Clustering

- Cluster multiple bits in a single variable node
  - bits along a lane
  - ideally: no spreading of side-channel info
- Cluster size vs. resource usage
  - runtime and memory: exp. in cluster size
  - we support 8-bit and 16-bit clusters



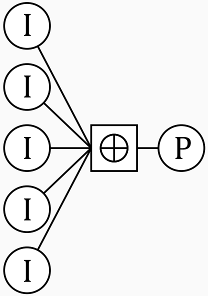
Clustering: Misalignment

- Problem: misalignment of clusters
  - previous SASCA on AES: operations on bytes
  - Keccak operations not aligned
- Example: $A \oplus \mathrm{ROT}(B, 4)$
- Need to split clusters
  - requires extraction of marginals



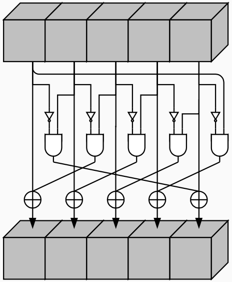
Clustering: Handling θ

- Computation of column parity
  - 5-input $\oplus$ node (efficient propagation)
  - enumeration of all possible values: 2^{40} (8-bit cluster)
  - solution: fast convolution of distributions using Walsh-Hadamard transform



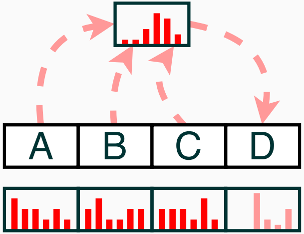
Clustering: Further Considerations

- Handling χ
  - break up clusters to deal with invertability



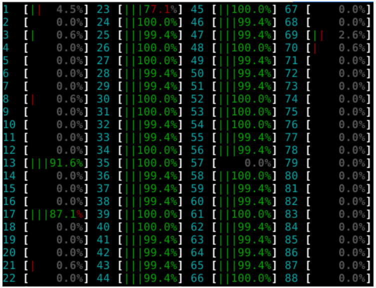
Clustering: Further Considerations

- Handling χ
  - break up clusters to deal with invertability
- Handling 32-bit leakage
  - found efficient method to combine leakage
  - convolution instead of enumeration



Attack Runtime

- Open-source Python implementation of BP on Keccak
  `https://github.com/keccaksasca/keccaksasca`
- Restriction to first two rounds of Keccak-f
- Runtime per BP iteration (updating all nodes once)
  - 8-bit clusters: $\sim$ seconds on single core
  - 16-bit clusters: $\sim$ 1 minute using 44 cores
  - 8-bit clusters sufficient in most cases
- BP: iterative algorithm, repeat until convergence.
  - typically < 10 iterations

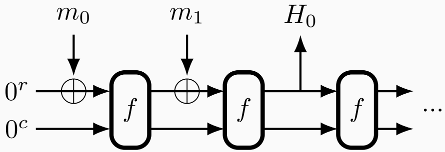


Attack Evaluation

- Goal: recover secret input of Keccak-f
- Evaluation tool: leakage simulations
  - noisy HW-leakage of loads/stores (at typical locations)
  - for 8, 16, and 32-bit implementations
  - vary noise σ, retrieve success rate
- Analyze impact of key size
  - evaluate 128 and 256-bit keys

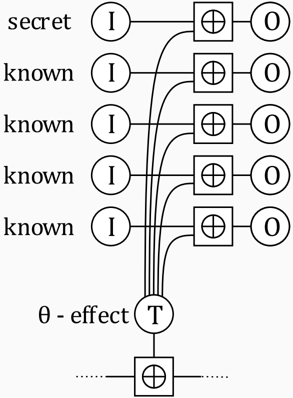
On the Impact of the Input State

- Keccak-f input: part secret, part known
- Content of public part impacts success rate!
- *All-zero public input*
  - state = secret || 0000...
  - example: SHAKE(128-bit seed)
- *Random public input*
  - state = secret || rand
  - example: H(msg || key)
- Attacks with *Random public input* work much better!



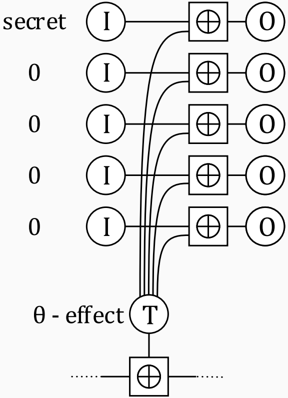
But why though?

- Reason: $\oplus$ of θ-effect T
- Observation: knowing T allows key recovery



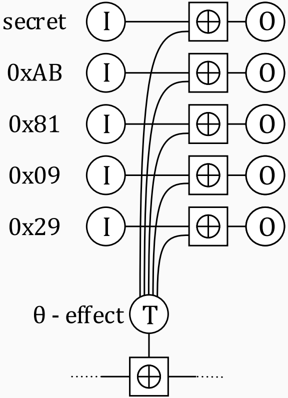
But why though?

- Reason: $\oplus$ of θ-effect T
- Observation: knowing T allows key recovery
- *All-zero public input*
  - T added 4 times to 0
  - same operation 4 times, averaging

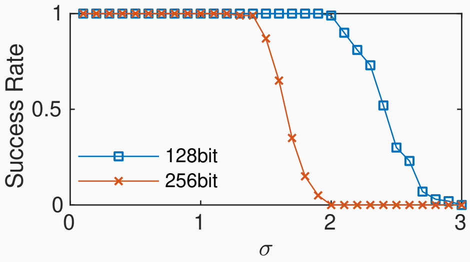


But why though?

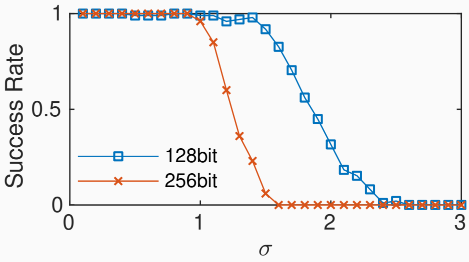
- Reason: $\oplus$ of θ-effect T
- Observation: knowing T allows key recovery
- *All-zero public input*
  - T added 4 times to 0
  - same operation 4 times, averaging
- *Random public input*
  - T added to 4 different values
  - similar to a DPA using 4 traces



8-bit HW leakage, real $\sigma \approx 0.5$ (XMEGA128D4)



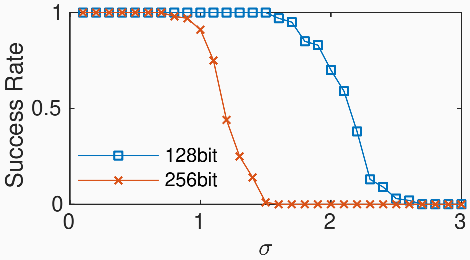
Random public input



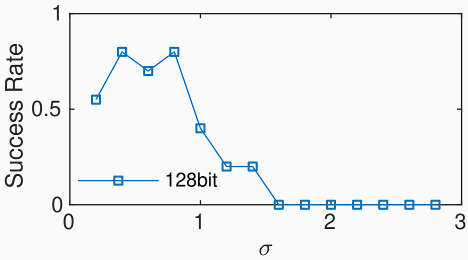
All-zero public input

Results: 16-bit Device

16-bit HW leakage, real $\sigma \approx$?



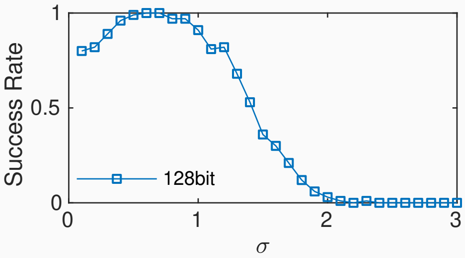
Random public input



All-zero public input

Results: 32-bit Device

32-bit HW leakage, real $\sigma \approx 0.4$ - 3 (STM32F303)



Random public input

Conclusion

Single-trace attacks are a considerable threat . . .

- especially for 8/16-bit implementations, situation less clear for 32-bit devices

But . . .

- we used a simple leakage model (simulations with univariate HW templates)
- more sophisticated attacker will fare better (remember: on-device profiling)

Must always include (basic) countermeasures . . .

- hiding (shuffling, dummy operations, etc.) effective
- masking also an option, but some restrictions

https://github.com/keccaksasca/keccaksasca

Thank you!