Leakage Resilient Value Comparison With Application to Message Authentication

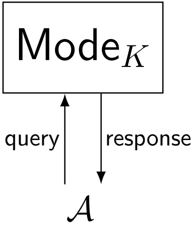
Christoph Dobraunig[1,2], Bart Mennink[3]

[1]: Lamarr Security Research (Austria)
[2]: Graz University of Technology (Austria)
[3]: Radboud University (The Netherlands)

ESCADA

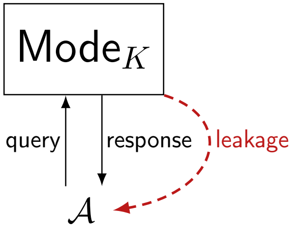
EUROCRYPT 2021
October 2021

Black-Box Security and Side-Channel Attacks



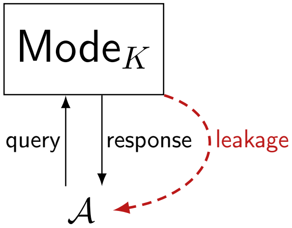
- Cryptographic modes are usually analyzed in black-box setting

Black-Box Security and Side-Channel Attacks



- Cryptographic modes are usually analyzed in black-box setting
- However, evaluations may leak secret information

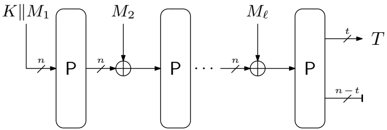
Black-Box Security and Side-Channel Attacks



- Cryptographic modes are usually analyzed in black-box setting
- However, evaluations may leak secret information
- Two main types of countermeasures:
  - Protection at implementation-level: masking or hiding
  - Protection at mode-level: leakage resilience

Example: Message Authentication (1/2)

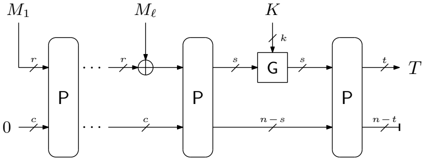
FKS: Full-state Keyed Sponge (Simplified) [BDPV12,GPT15,MRV15]



- Very efficient
- No mode-level protection against side-channel attacks
- Requires implementation-level protection

Example: Message Authentication (2/2)

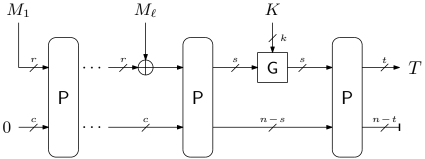
SuKS: Suffix Keyed Sponge [BDPV11,DEM+17,DM19]



- Processes key at the end
- Minimizes number of evaluations of secret states
- Leakage resilient if G and P leak up to λ bits of secrecy (per evaluation)

Example: Message Authentication (2/2)

SuKS: Suffix Keyed Sponge [BDPV11,DEM+17,DM19]

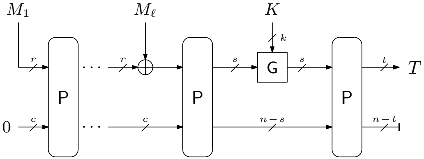


- Processes key at the end
- Minimizes number of evaluations of secret states
- Leakage resilient if G and P leak up to λ bits of secrecy (per evaluation)

How does SuKS verify tags?

Closer Look at SuKS

SuKS: Suffix Keyed Sponge [BDPV11,DEM+17,DM19]

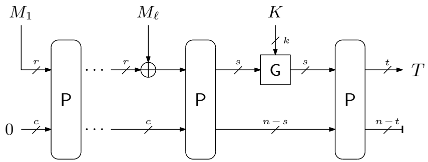


Tag Verification

- Given message/tag tuple $(M, T^\star)$:
  - Compute $T = \text{SuKS}(K, M)$
  - If $T^\star = T$ return 1, otherwise return 0

Closer Look at SuKS

SuKS: Suffix Keyed Sponge [BDPV11,DEM+17,DM19]



Tag Verification

- Given message/tag tuple $(M, T^\star)$:
  - Compute $T = \text{SuKS}(K, M)$
  - If $T^\star = T$ return 1, otherwise return 0
- Verification might leak information about T!

Leakage from Value Comparison

- Leakage resilience usually centers around MAC/AE design
- Tag verification often left out of scope
- Assumed to be protected at implementation level

Leakage from Value Comparison

- Leakage resilience usually centers around MAC/AE design
- Tag verification often left out of scope
- Assumed to be protected at implementation level

- But MAC design already uses protected primitive
- Why not re-use it for verification?

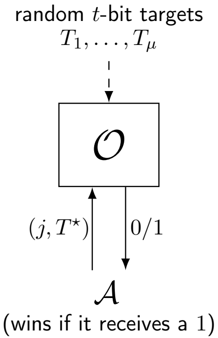
Leakage from Value Comparison

- Leakage resilience usually centers around MAC/AE design
- Tag verification often left out of scope
- Assumed to be protected at implementation level

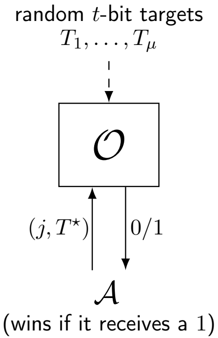
- But MAC design already uses protected primitive
- Why not re-use it for verification?

Formal analysis of leakage resilient value comparison

Modeling Value Comparison: Black-Box

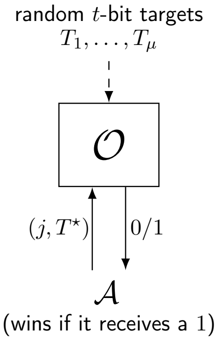


Modeling Value Comparison: Black-Box



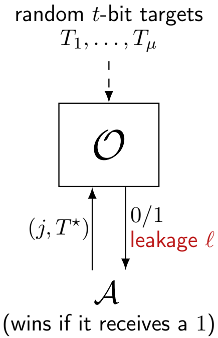
- Plain target verification works:
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[T_j \stackrel{?}{=} T^\star \right]\!\right]$

Modeling Value Comparison: Black-Box



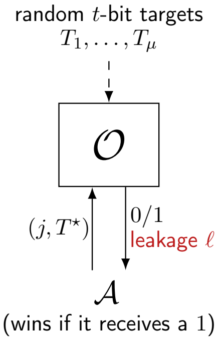
- Plain target verification works:
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[T_j \stackrel{?}{=} T^\star \right]\!\right]$
- Adversary making q queries
  wins with probability at most $q/2^t$

Modeling Value Comparison: Leaky Setting



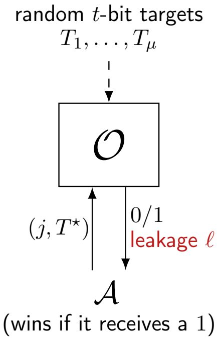
- Adversary gains leakage per oracle evaluation

Modeling Value Comparison: Leaky Setting



- Adversary gains leakage per oracle evaluation
- Plain target verification fails:
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[T_j \stackrel{?}{=} T^\star \right]\!\right]$
  - Oracle might leak λ bits of T_j per query
  - T_j is obtained after $\lceil t/\lambda \rceil$ queries

Modeling Value Comparison: Leaky Setting

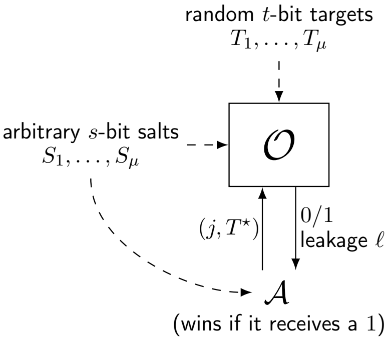


- Adversary gains leakage per oracle evaluation

- Plain target verification fails:
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[T_j \stackrel{?}{=} T^\star \right]\!\right]$
  - Oracle might leak λ bits of T_j per query
  - T_j is obtained after $\lceil t/\lambda \rceil$ queries

- A more sophisticated oracle $\mathcal{O}$ needed!

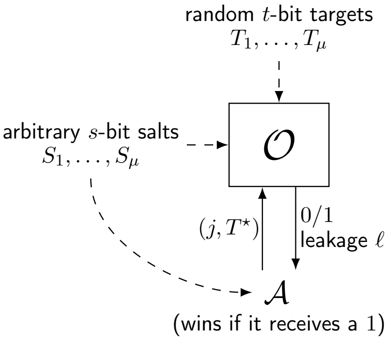
Modeling Value Comparison: General Model



General Model

- μ random target values $T_1, \ldots, T_\mu$
- μ salts $S_1, \ldots, S_\mu$
  - In principle unique
  - Randomization or omission possible
  - In applications, salts are often present

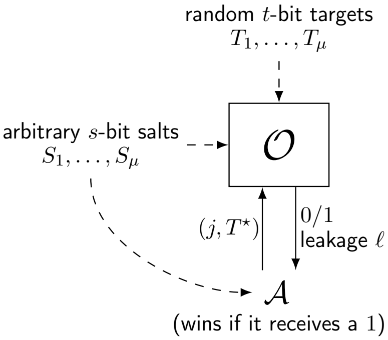
Modeling Value Comparison: General Model



General Model

- μ random target values $T_1, \ldots, T_\mu$
- μ salts $S_1, \ldots, S_\mu$
  - In principle unique
  - Randomization or omission possible
  - In applications, salts are often present
- $\mathcal{O}$ is some verification oracle

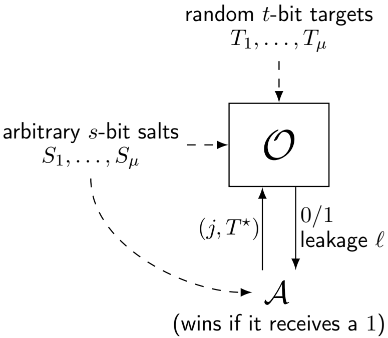
Modeling Value Comparison: General Model



General Model

- μ random target values $T_1, \ldots, T_\mu$
- μ salts $S_1, \ldots, S_\mu$
  - In principle unique
  - Randomization or omission possible
  - In applications, salts are often present
- $\mathcal{O}$ is some verification oracle
- Adversary $\mathcal{A}$ can make attempts $(j, T^\star) \mapsto 0/1$

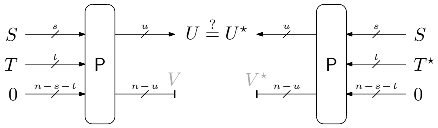
Modeling Value Comparison: General Model



General Model

- μ random target values $T_1, \ldots, T_\mu$
- μ salts $S_1, \ldots, S_\mu$
  - In principle unique
  - Randomization or omission possible
  - In applications, salts are often present
- $\mathcal{O}$ is some verification oracle
- Adversary $\mathcal{A}$ can make attempts $(j, T^\star) \mapsto 0/1$
- $\mathcal{A}$ also obtains leakage:
  - Evaluation of cryptographic primitive within $\mathcal{O}$ may leak λ bits (non-adaptively)
  - Each value comparison may leak λ bits (non-adaptively)

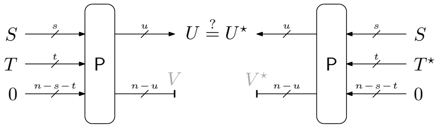
PVP: Permutation-Based Value Processing (1/2)



- Let P be an n-bit permutation
- Consider value comparison

$$\mathcal{O} : (j, T^\star) \mapsto \left[\!\!\left[\text{left}_u(\mathsf{P}(S_j \parallel T_j \parallel 0^*)) \stackrel{?}{=} \text{left}_u(\mathsf{P}(S_j \parallel T^\star \parallel 0^*)) \right]\!\!\right]$$

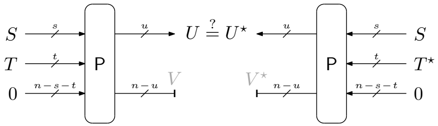
PVP: Permutation-Based Value Processing (1/2)



- Let P be an n-bit permutation
- Consider value comparison

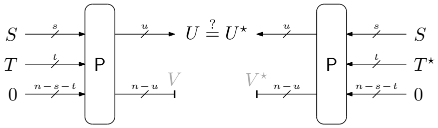
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[\text{left}_u(\mathsf{P}(S_j \parallel T_j \parallel 0^*)) \stackrel{?}{=} \text{left}_u(\mathsf{P}(S_j \parallel T^\star \parallel 0^*)) \right]\!\right]$
- PVP gives leakage resilient value comparison

PVP: Permutation-Based Value Processing (2/2)



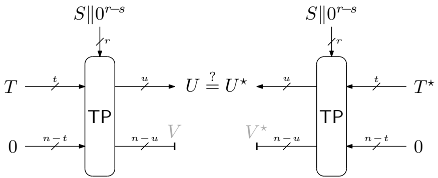
- If P is a public permutation (e.g., Keccak-f):
  - We require $t, u \ll n$, but typically n is large enough
  - Similar to earlier suggestion of designers of ISAP [DEM+19]

PVP: Permutation-Based Value Processing (2/2)



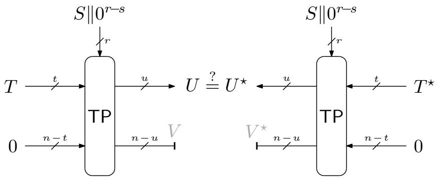
- If P is a public permutation (e.g., Keccak-f):
  - We require $t, u \ll n$, but typically n is large enough
  - Similar to earlier suggestion of designers of ISAP [DEM+19]
- If P is a secret permutation (e.g., AES_K):
  - No limitation on t, u
  - Better security bound but one needs protected AES_K

TPVP: Tweakable Permutation-Based Value Processing



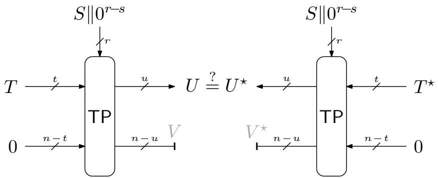
- Let TP be an n-bit tweakable permutation with r-bit tweaks
- Consider value comparison
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[\text{left}_u(\mathsf{TP}(S_j \parallel 0^*, T_j \parallel 0^*)) \stackrel{?}{=} \text{left}_u(\mathsf{TP}(S_j \parallel 0^*, T^\star \parallel 0^*)) \right]\!\right]$

TPVP: Tweakable Permutation-Based Value Processing



- Let TP be an n-bit tweakable permutation with r-bit tweaks
- Consider value comparison
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[\text{left}_u(\text{TP}(S_j \parallel 0^*, T_j \parallel 0^*)) \stackrel{?}{=} \text{left}_u(\text{TP}(S_j \parallel 0^*, T^\star \parallel 0^*))\right]\!\right]$
- TPVP gives leakage resilient value comparison

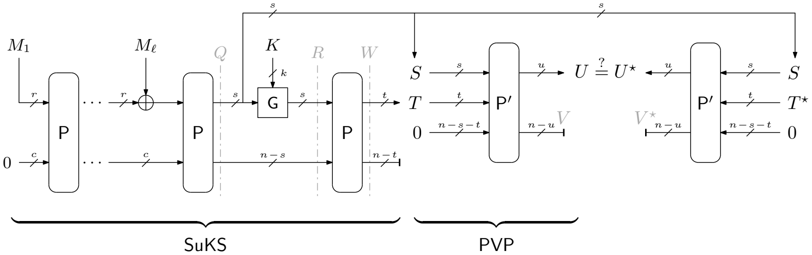
TPVP: Tweakable Permutation-Based Value Processing



- Let TP be an n-bit tweakable permutation with r-bit tweaks
- Consider value comparison
  $\mathcal{O} : (j, T^\star) \mapsto \left[\!\left[\text{left}_u(\mathsf{TP}(S_j \parallel 0^*, T_j \parallel 0^*)) \stackrel{?}{=} \text{left}_u(\mathsf{TP}(S_j \parallel 0^*, T^\star \parallel 0^*))\right]\!\right]$
- TPVP gives leakage resilient value comparison

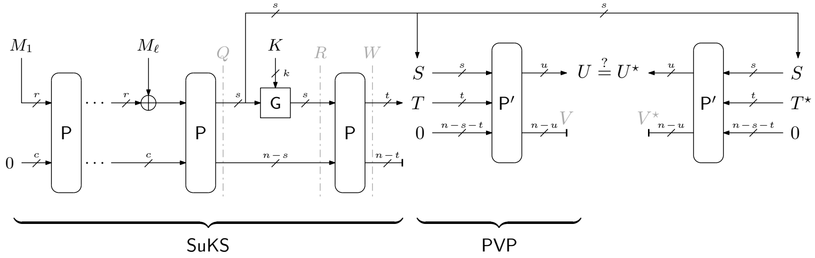
- Same conditions on t, u apply
- TPVP with secret permutation was used in Spook [BBB+19]

SuKS-then-PVP (StP)



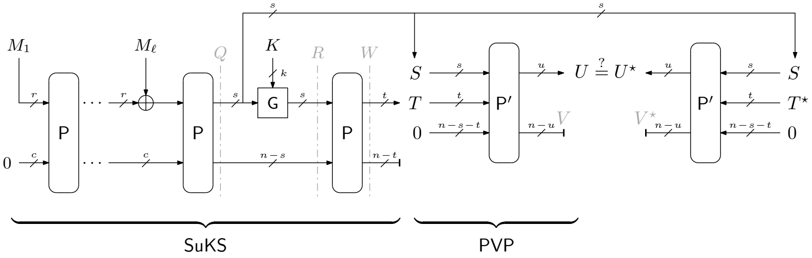
- Natural combination of SuKS and PVP
- Salt taken from keyless computation of SuKS
  - Sufficiently random
  - Non-secret to adversary

SuKS-then-PVP (StP)



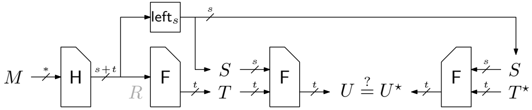
- Natural combination of SuKS and PVP
- Salt taken from keyless computation of SuKS
  - Sufficiently random
  - Non-secret to adversary
- Leakage resilience of StP follows from that of SuKS and of PVP

SuKS-then-PVP (StP)



- Natural combination of SuKS and PVP
- Salt taken from keyless computation of SuKS
  - Sufficiently random
  - Non-secret to adversary
- Leakage resilience of StP follows from that of SuKS and of PVP
- Disadvantage of composition: independent primitives P and P′ needed

Hash-then-Function-then-Function (HaFuFu)



- H is hash function and F is secret random function
- HaFuFu: uses same F for MAC and for verification
- Salt taken from keyless computation of H
- Leakage resilience of HaFuFu: as before, but dedicated proof needed

Conclusion

Value Comparison

- Prominent role in tag verification
- Further applications in fault countermeasures
- Can be done efficiently by re-using existing resources
- Processed value comparison leads to slightly larger success probability

More in Paper

- Exact leakage resilience analysis
- Security assumptions
- Relaxation of salt

Thank you for your attention!