Radboud University

Sound Hashing Modes of Arbitrary Functions, Permutations, and Block Ciphers (SoK)

Joan Daemen[1] Bart Mennink[1] Gilles Van Assche[2]

Fast Software Encryption

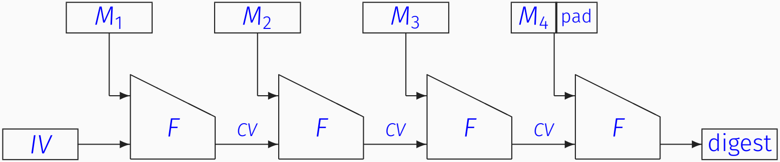
Paris, March 2019

[1]Radboud University
[2]STMicroelectronics

Hash function example 1: SHA-256

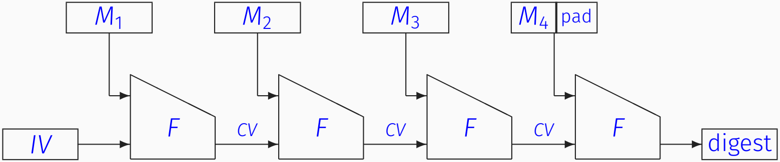
Hash function example 1: SHA-256

Hash function h from compression function F with **Merkle-Damgård**:

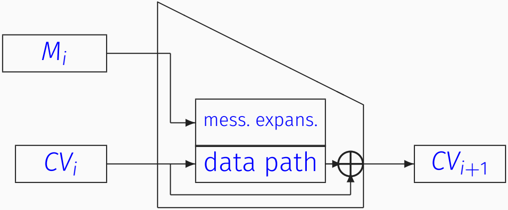


Hash function example 1: SHA-256

Hash function h from compression function F with **Merkle-Damgård**:

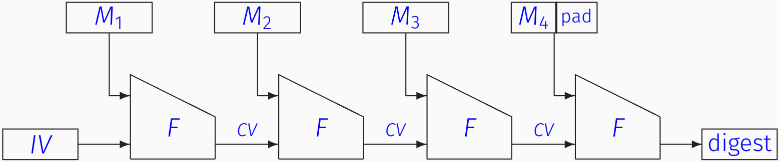


Compression function F from block cipher B with **Davies-Meyer**:

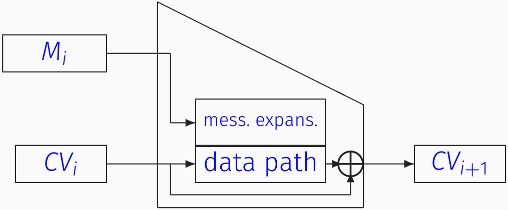


Hash function example 1: SHA-256

Hash function h from compression function F with **Merkle-Damgård**:



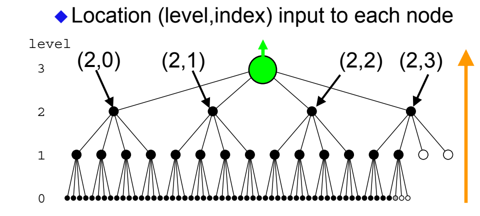
Compression function F from block cipher B with **Davies-Meyer**:



Underlying primitive: block cipher with 256-bit block and 512-bit *key*

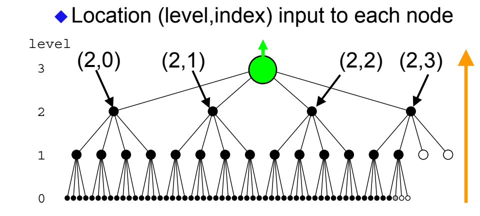
Example 2: MD6 [Rivest et al. 2008]

Hash function h from CF F with dedicated tree hash mode:

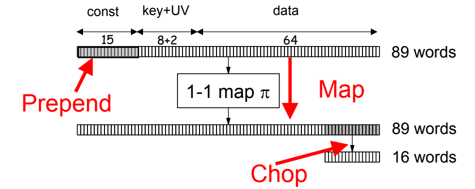


Example 2: MD6 [Rivest et al. 2008]

Hash function *h* from CF *F* with dedicated tree hash mode:

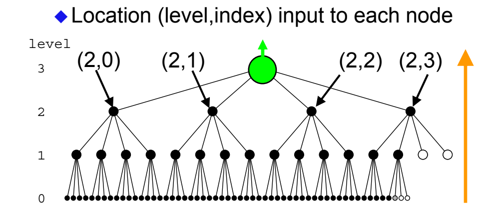


CF *F* from permutation *P* with dedicated construction:

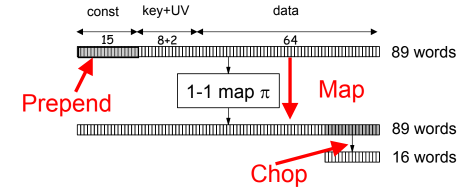


Example 2: MD6 [Rivest et al. 2008]

Hash function h from CF F with dedicated tree hash mode:



CF F from permutation P with dedicated construction:

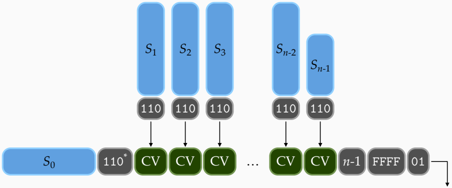


Underlying primitive: 5696-bit permutation

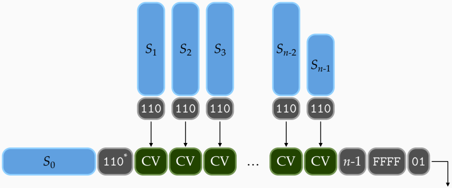
Example 3: KangarooTwelve [Keccak Team 2016]

Example 3: KangarooTwelve [Keccak Team 2016]

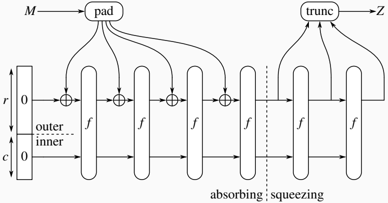
Parallel XOF from XOF with Sakura-encoded [KT 2014] tree hash mode:



Parallel XOF from XOF with Sakura-encoded [KT 2014] tree hash mode:

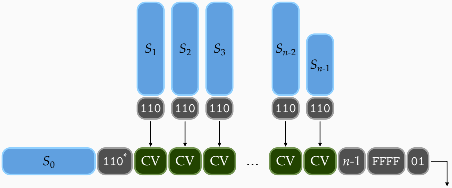


XOF from permutation with **sponge** [KT 2008]:

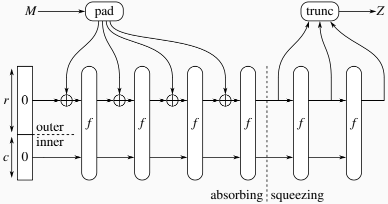


Example 3: KangarooTwelve [Keccak Team 2016]

Parallel XOF from XOF with Sakura-encoded [KT 2014] tree hash mode:



XOF from permutation with **sponge** [KT 2008]:



Underlying primitive: 1600-bit permutation KECCAK-p[12]

Basis for security of hash functions

- We cannot prove a hash function h is secure

Basis for security of hash functions

- We cannot prove a hash function h is secure
- Trust in security based on public scrutiny and cryptanalysis

Basis for security of hash functions

- We cannot prove a hash function h is secure
- Trust in security based on public scrutiny and cryptanalysis
- But we can prove security of idealized version $\mathcal{H}$ of the function
  - … $\mathcal{H}$ is h *with underlying primitive replaced by random one*

Basis for security of hash functions

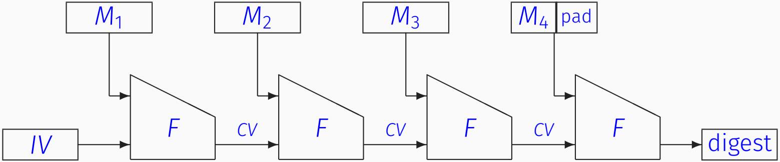
- We cannot prove a hash function h is secure
- Trust in security based on public scrutiny and cryptanalysis
- But we can prove security of idealized version $\mathcal{H}$ of the function
  - ... $\mathcal{H}$ is h *with underlying primitive replaced by random one*
- Ideal hash function: random oracle $\mathcal{RO}$
- Upper bound on advantage of distinguishing $\mathcal{H}$ from $\mathcal{RO}$
  - this bound says something about the mode only
  - better attacks must exploit specific properties of primitive

Basis for security of hash functions

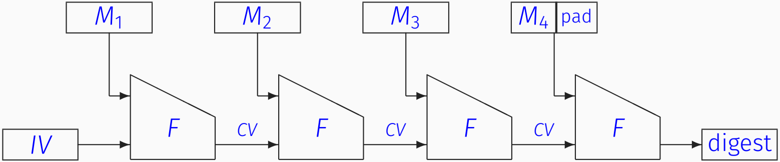
- We cannot prove a hash function h is secure
- Trust in security based on public scrutiny and cryptanalysis
- But we can prove security of idealized version $\mathcal{H}$ of the function
  - ... $\mathcal{H}$ is h *with underlying primitive replaced by random one*
- Ideal hash function: random oracle $\mathcal{RO}$
- Upper bound on advantage of distinguishing $\mathcal{H}$ from $\mathcal{RO}$
  - this bound says something about the mode only
  - better attacks must exploit specific properties of primitive
- In other words, they bound the success probability of generic attacks

What can happen if you don’t have a good bound?

What can happen if you don't have a good bound?

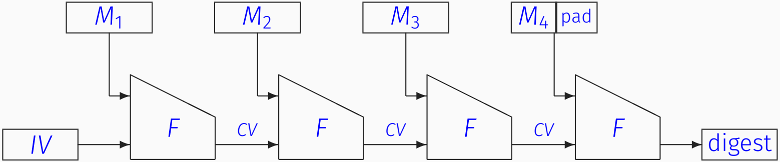


What can happen if you don't have a good bound?



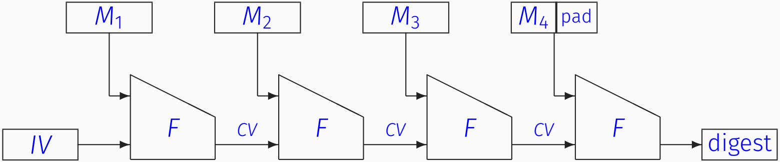
- Length extension property

What can happen if you don't have a good bound?



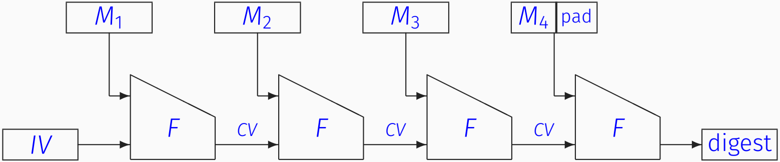
- Length extension property
  - MAC function $h(K|M)$ not secure against forgery

What can happen if you don't have a good bound?



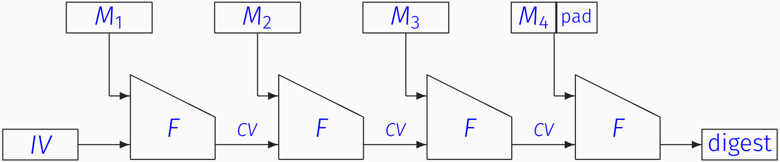
- Length extension property
  - MAC function $h(K|M)$ not secure against forgery
  - fixing requires adding expensive construction: HMAC

What can happen if you don't have a good bound?



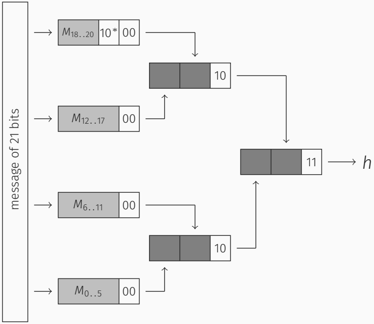
- Length extension property
  - MAC function $h(K|M)$ not secure against forgery
  - fixing requires adding expensive construction: HMAC
- Attacks with less complexity than expected
  - 2nd pre-image for long messages
  - multi-collisions
  - herding attack, …

What can happen if you don't have a good bound?

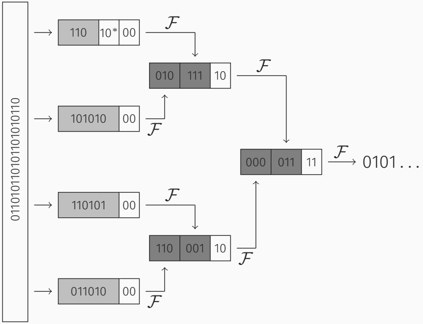


- Length extension property
  - MAC function $h(K|M)$ not secure against forgery
  - fixing requires adding expensive construction: HMAC
- Attacks with less complexity than expected
  - 2nd pre-image for long messages
  - multi-collisions
  - herding attack, ...
- Affect all old-style hash standards: MD5, SHA-1 and all SHA-2

Hashing, scope of this SoK paper

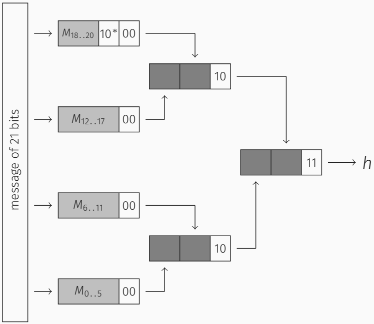


template generation
$Z \leftarrow \mathcal{T}(|M|, \text{params})$

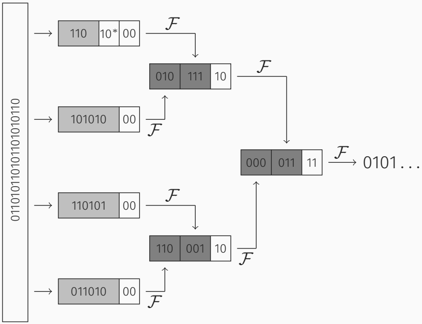


template execution $H \leftarrow \mathcal{F}(S_{\text{final}})$
with $S \leftarrow \mathcal{Y}[\mathcal{F}](Z, M)$

Hashing, scope of this SoK paper



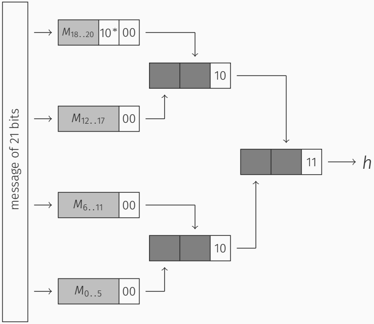
template generation
$Z \leftarrow \mathcal{T}(|M|, \text{params})$



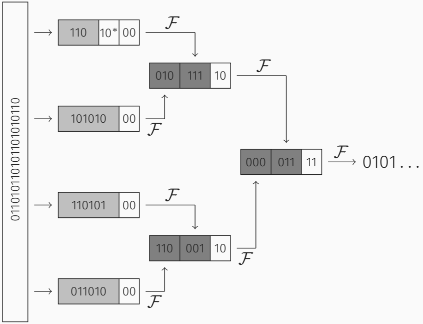
template execution $H \leftarrow \mathcal{F}(S_{\text{final}})$
with $S \leftarrow \mathcal{Y}[\mathcal{F}](Z, M)$

- Modes $\mathcal{T}$ for any tree topology, including sequential hashing

Hashing, scope of this SoK paper



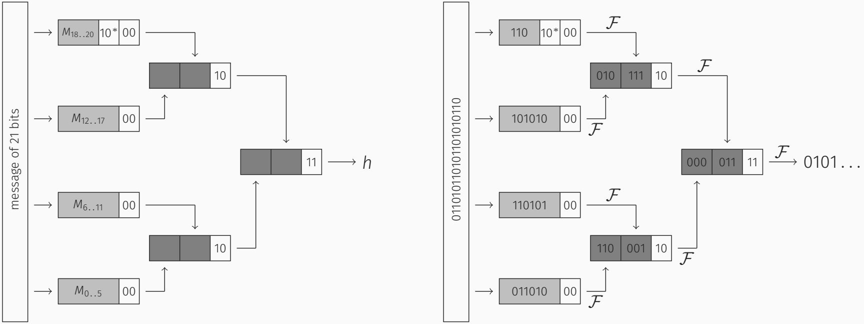
template generation
$Z \leftarrow \mathcal{T}(|M|, \text{params})$



template execution $H \leftarrow \mathcal{F}(S_{\text{final}})$
with $S \leftarrow \mathcal{Y}[\mathcal{F}](Z, M)$

- Modes $\mathcal{T}$ for any tree topology, including sequential hashing
- Three types of underlying function $\mathcal{F}$:

Hashing, scope of this SoK paper

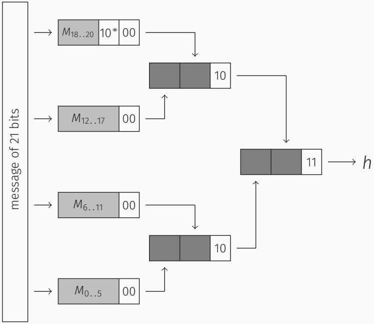


template generation
$Z \leftarrow \mathcal{T}(|M|, \text{params})$

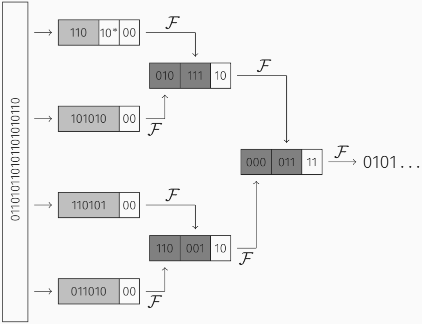
template execution $H \leftarrow \mathcal{F}(S_{\text{final}})$
with $S \leftarrow \mathcal{Y}[\mathcal{F}](Z, M)$

- Modes $\mathcal{T}$ for any tree topology, including sequential hashing
- Three types of underlying function $\mathcal{F}$:
  - arbitrary function: XOF, hash, or compression function

Hashing, scope of this SoK paper



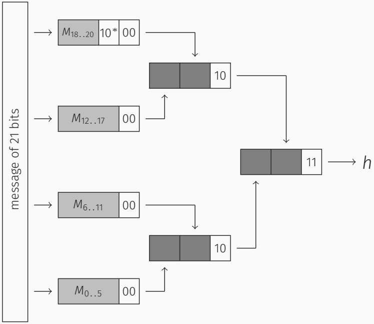
template generation
$Z \leftarrow \mathcal{T}(|M|, \text{params})$



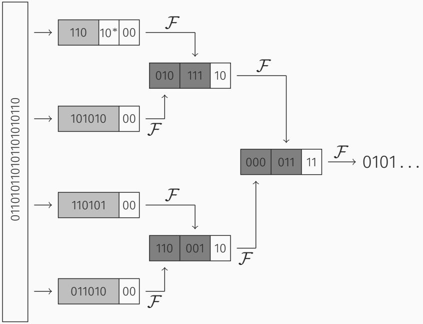
template execution $H \leftarrow \mathcal{F}(S_{\text{final}})$
with $S \leftarrow \mathcal{Y}[\mathcal{F}](Z, M)$

- Modes $\mathcal{T}$ for any tree topology, including sequential hashing
- Three types of underlying function $\mathcal{F}$:
  - arbitrary function: XOF, hash, or compression function
  - truncated permutation

Hashing, scope of this SoK paper



template generation
$Z \leftarrow \mathcal{T}(|M|, \text{params})$



template execution $H \leftarrow \mathcal{F}(S_{\text{final}})$
with $S \leftarrow \mathcal{Y}[\mathcal{F}](Z, M)$

- Modes $\mathcal{T}$ for any tree topology, including sequential hashing
- Three types of underlying function $\mathcal{F}$:
  - arbitrary function: XOF, hash, or compression function
  - truncated permutation
  - (truncated) block cipher

Conditions for sound hashing

Conditions for sound hashing

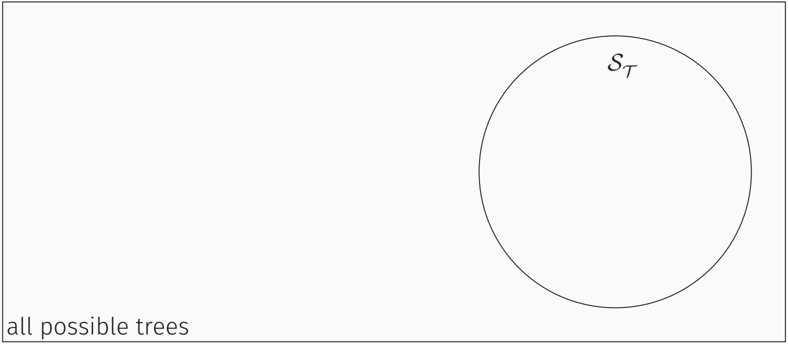
We prove it is hard to distinguish $\mathcal{H}$ from $\mathcal{RO}$ if $\mathcal{T}$ satisfies certain conditions:

Conditions for sound hashing

We prove it is hard to distinguish $\mathcal{H}$ from $\mathcal{RO}$ if $\mathcal{T}$ satisfies certain conditions:

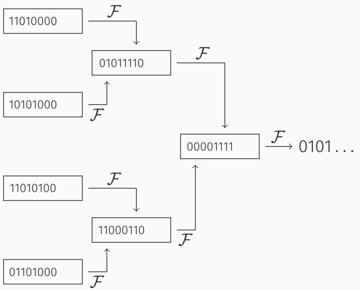
- For all cases:
  - message-decodability
  - subtree-freeness
  - radical-decodability
- For permutations and block ciphers:
  - leaf-anchoring

Trees and the set $\mathcal{S}_{\mathcal{T}}$

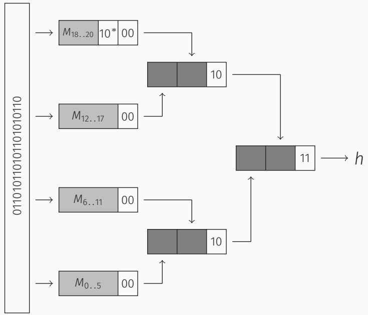


$\mathcal{S}_{\mathcal{T}}$: the set of all possible trees that can be generated by mode $\mathcal{T}$

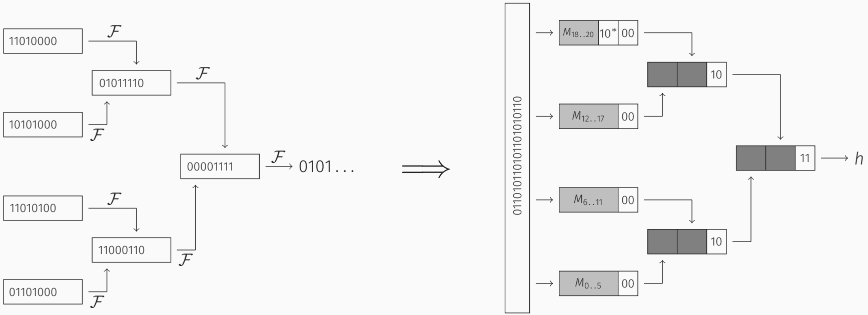
Condition 1: message decodability



$\Longrightarrow$

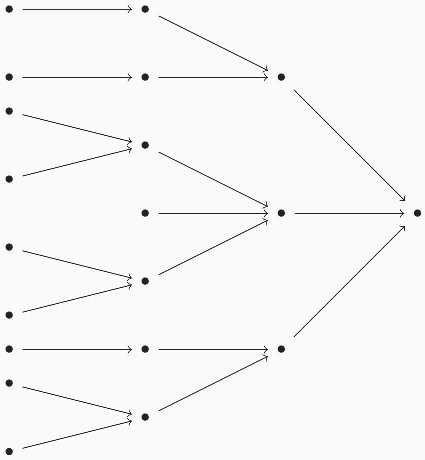


Condition 1: message decodability

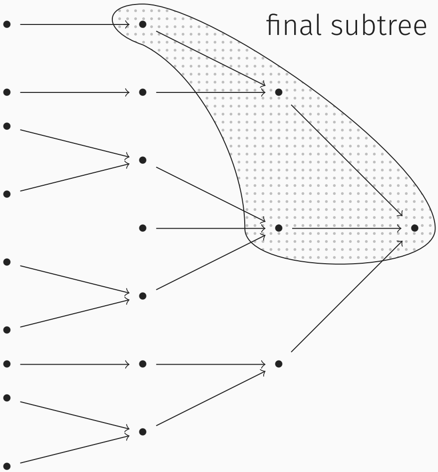


$\forall S \in \mathcal{S}_{\mathcal{T}}$ there exists an algorithm for decoding S to (M, Z)

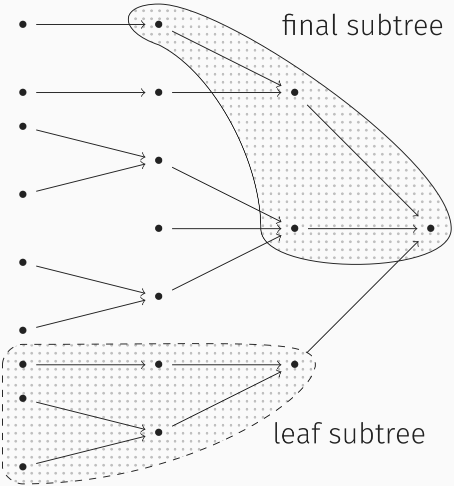
Condition 2: subtree-freeness



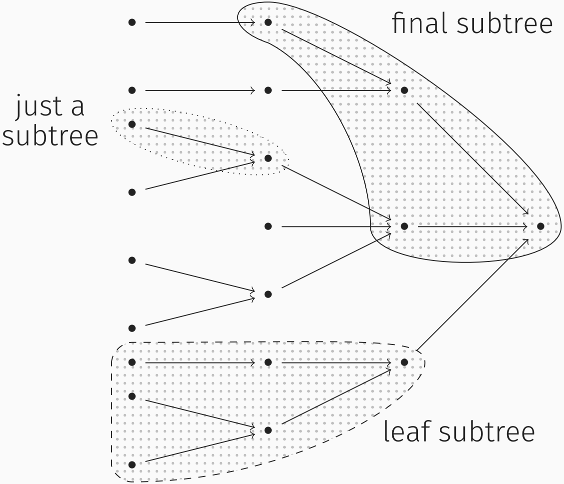
Condition 2: subtree-freeness



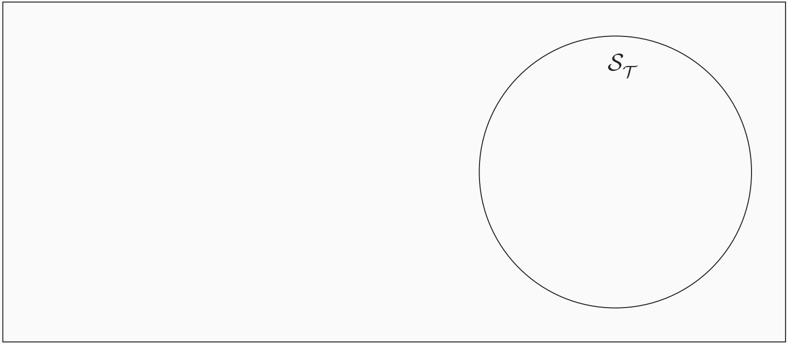
Condition 2: subtree-freeness



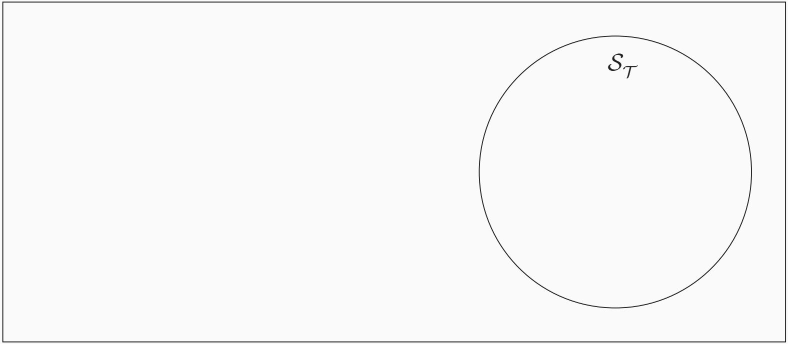
Condition 2: subtree-freeness



Condition 2: subtree-freeness

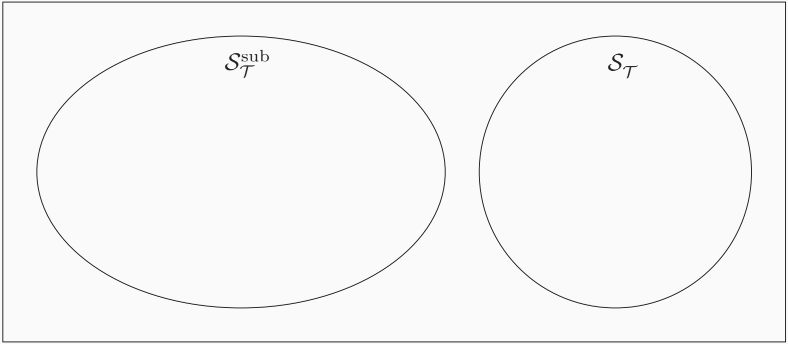


Condition 2: subtree-freeness



$\mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$: the set of all trees that are proper subtrees of a tree in $\mathcal{S}_{\mathcal{T}}$

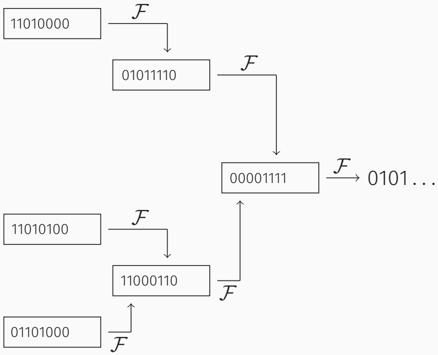
Condition 2: subtree-freeness



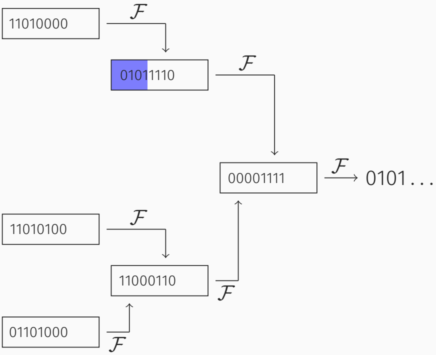
$\mathcal{S}_{\mathcal{T}}^{\mathrm{sub}}$: the set of all trees that are proper subtrees of a tree in $\mathcal{S}_{\mathcal{T}}$

Subtree-freeness: $\mathcal{S}_{\mathcal{T}} \cap \mathcal{S}_{\mathcal{T}}^{\mathrm{sub}} = \emptyset$

Condition 3: radical-decodability

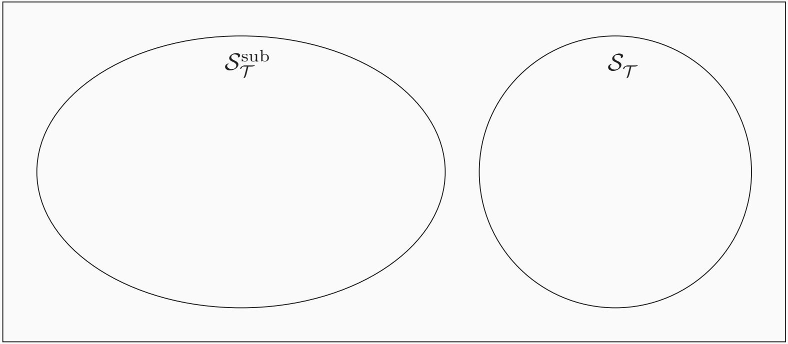


Condition 3: radical-decodability

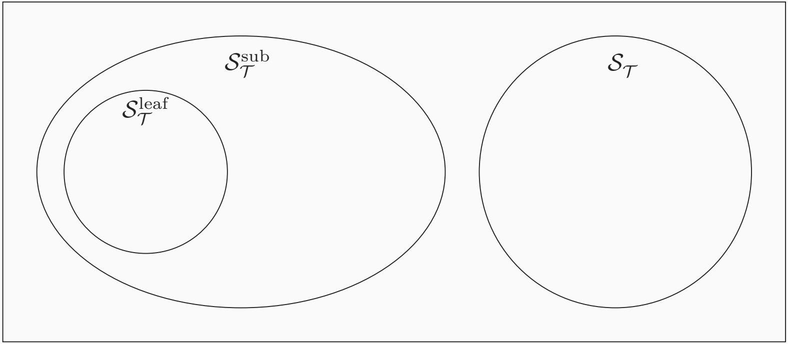


Radical: a CV that has no $\mathcal{F}$-pre-image

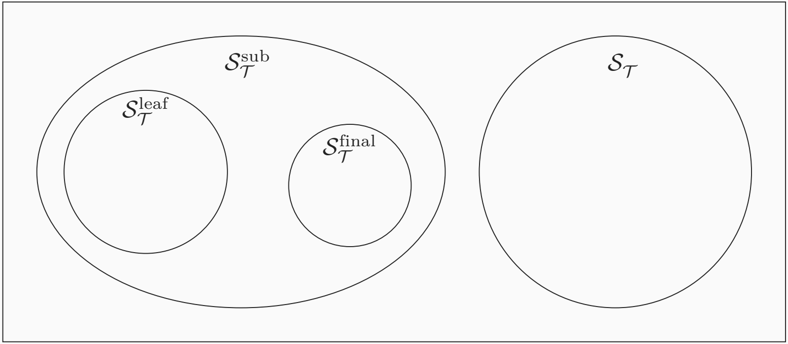
Condition 3: radical-decodability

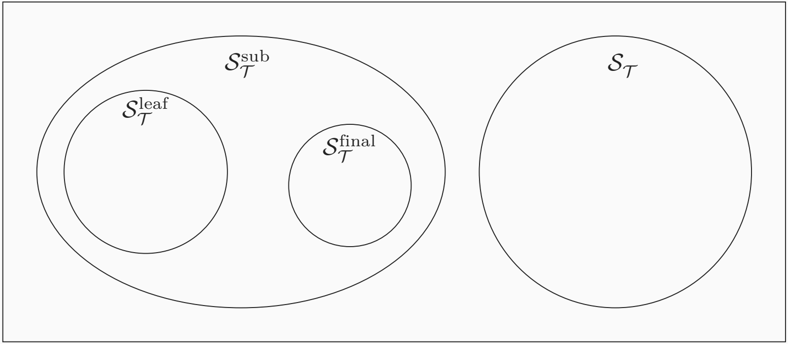


Condition 3: radical-decodability



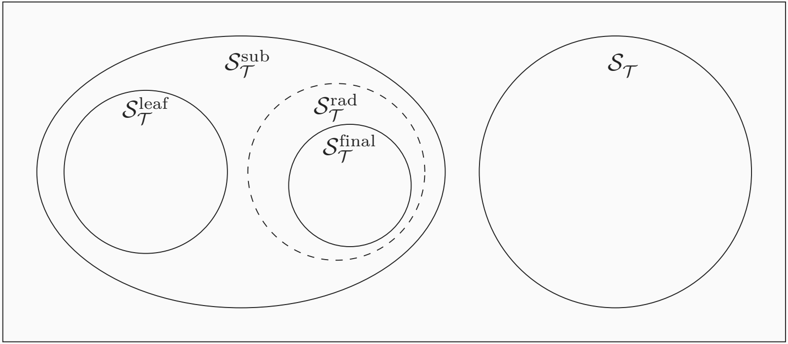
Condition 3: radical-decodability





Radical-decodability, simplified: for all final subtrees ($\mathcal{S}_{\mathcal{T}}^{\text{final}}$) one can **unambiguously identify** a *radical*

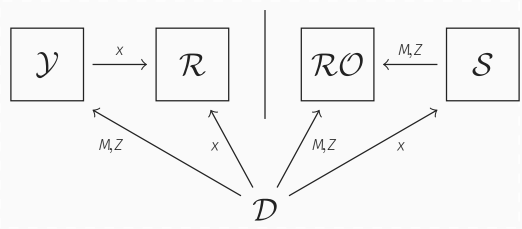
Condition 3: radical-decodability



Radical-decodability, simplified: for all final subtrees ($\mathcal{S}_{\mathcal{T}}^{\text{final}}$) one can **unambiguously identify** a *radical*

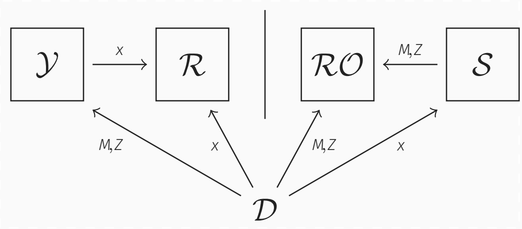
Radical-decodability, actually: this is true for all subtrees in some set $\mathcal{S}_{\mathcal{T}}^{\text{rad}}$ that includes $\mathcal{S}_{\mathcal{T}}^{\text{final}}$

Adversary model: differentiating from a random oracle



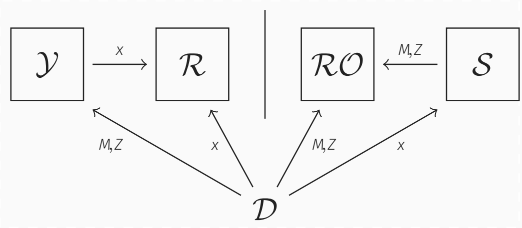
- Indifferentiability [Maurer et al. 2004]

Adversary model: differentiating from a random oracle



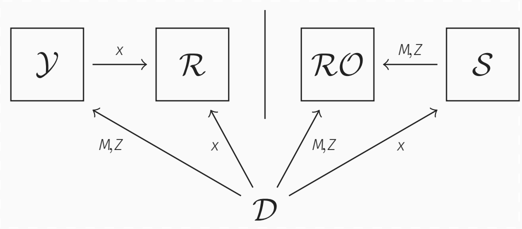
- Indifferentiability [Maurer et al. 2004] for hashing [Coron et al. 2005]

Adversary model: differentiating from a random oracle



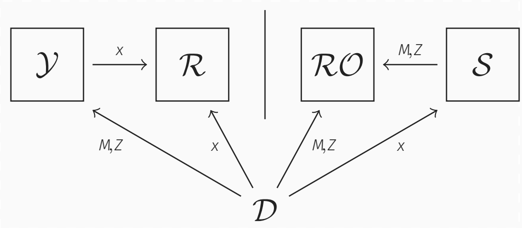
- Indifferentiability [Maurer et al. 2004] for hashing [Coron et al. 2005]
- For sponge: [KT 2008] $\text{adv} \leq \binom{N}{2} 2^{-c}$: birthday bound in capacity

Adversary model: differentiating from a random oracle



- Indifferentiability [Maurer et al. 2004] for hashing [Coron et al. 2005]
- For sponge: [KT 2008] $\text{adv} \leq \binom{N}{2} 2^{-c}$: birthday bound in capacity
- This paper: $\text{adv} \leq \binom{N}{2} 2^{-n}$: birthday bound in CV length

Adversary model: differentiating from a random oracle



- Indifferentiability [Maurer et al. 2004] for hashing [Coron et al. 2005]
- For sponge: [KT 2008] adv $\leq \binom{N}{2} 2^{-c}$: birthday bound in capacity
- This paper: adv $\leq \binom{N}{2} 2^{-n}$: birthday bound in CV length
- If mode satisfies our conditions

Condition 4: leaf-anchoring

- Problem with truncated permutation: **inverse queries**

Condition 4: leaf-anchoring

- Problem with truncated permutation: **inverse queries**
- Without additional condition this is easy to distinguish

Condition 4: leaf-anchoring

- Problem with truncated permutation: **inverse queries**
- Without additional condition this is easy to distinguish
- *Leaf anchoring*
    - n first bits of permutation input are *reserved*

Condition 4: leaf-anchoring

- Problem with truncated permutation: **inverse queries**
- Without additional condition this is easy to distinguish
- *Leaf anchoring*
  - n first bits of permutation input are *reserved*
  - constant IV in leaf nodes
  - CV in non-leaf nodes

Condition 4: leaf-anchoring

- Problem with truncated permutation: **inverse queries**
- Without additional condition this is easy to distinguish
- *Leaf anchoring*
  - n first bits of permutation input are *reserved*
  - constant IV in leaf nodes
  - CV in non-leaf nodes
- For block ciphers: anchoring must be in *data* input

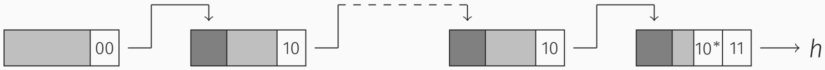
Condition 4: leaf-anchoring

- Problem with truncated permutation: **inverse queries**
- Without additional condition this is easy to distinguish
- *Leaf anchoring*
  - n first bits of permutation input are *reserved*
  - constant IV in leaf nodes
  - CV in non-leaf nodes
- For block ciphers: anchoring must be in *data* input
- Other countermeasures could be taken but this is the simplest

Condition 4: leaf-anchoring

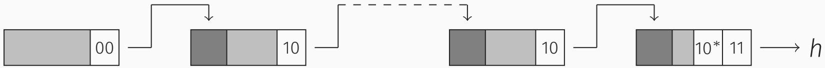
- Problem with truncated permutation: **inverse queries**
- Without additional condition this is easy to distinguish
- *Leaf anchoring*
  - n first bits of permutation input are *reserved*
  - constant IV in leaf nodes
  - CV in non-leaf nodes
- For block ciphers: anchoring must be in *data* input
- Other countermeasures could be taken but this is the simplest
- Adding a feedforward à la Davies-Meyer does **not** help

With a compression function:

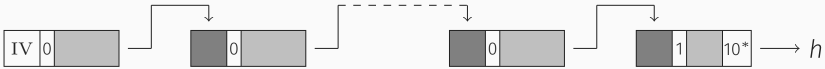


Minimum solutions for sequential hashing

With a compression function:



With a truncated permutation or block cipher:



Interesting implications of this work

- Tree hashing mode on top of a secure XOF

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF
    - e.g., KangarooTwelve on top of sponge

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF
  - e.g., KangarooTwelve on top of sponge
  - Sakura encoding [KT 2014] ensures subtree-freeness and radical decodability

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF
  - e.g., KangarooTwelve on top of sponge
  - Sakura encoding [KT 2014] ensures subtree-freeness and radical decodability
- Hashing based on permutations

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF
  - e.g., KangarooTwelve on top of sponge
  - Sakura encoding [KT 2014] ensures subtree-freeness and radical decodability
- Hashing based on permutations
  - Sponge is not covered: different type of animal

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF
  - e.g., KangarooTwelve on top of sponge
  - Sakura encoding [KT 2014] ensures subtree-freeness and radical decodability
- Hashing based on permutations
  - Sponge is not covered: different type of animal
  - MD6: n-bit IV in leaves and 1 framebit would have sufficed

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF
  - e.g., KangarooTwelve on top of sponge
  - Sakura encoding [KT 2014] ensures subtree-freeness and radical decodability
- Hashing based on permutations
  - Sponge is not covered: different type of animal
  - MD6: n-bit IV in leaves and 1 framebit would have sufficed
- Hashing based on block ciphers (e.g., MD5, SHA-1 and SHA-2)
  - Davies-Meyer feedforward is useless

Interesting implications of this work

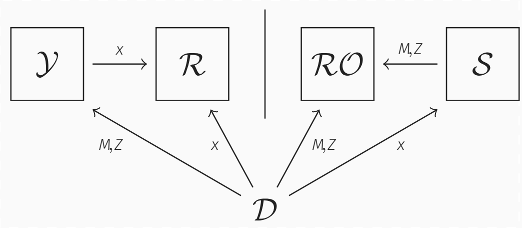
- Tree hashing mode on top of a secure XOF gives a secure XOF
  - e.g., KangarooTwelve on top of sponge
  - Sakura encoding [KT 2014] ensures subtree-freeness and radical decodability
- Hashing based on permutations
  - Sponge is not covered: different type of animal
  - MD6: n-bit IV in leaves and 1 framebit would have sufficed
- Hashing based on block ciphers (e.g., MD5, SHA-1 and SHA-2)
  - Davies-Meyer feedforward is useless
  - Merkle-Damgård strengthening is useless

Interesting implications of this work

- Tree hashing mode on top of a secure XOF gives a secure XOF
  - e.g., KangarooTwelve on top of sponge
  - Sakura encoding [KT 2014] ensures subtree-freeness and radical decodability
- Hashing based on permutations
  - Sponge is not covered: different type of animal
  - MD6: n-bit IV in leaves and 1 framebit would have sufficed
- Hashing based on block ciphers (e.g., MD5, SHA-1 and SHA-2)
  - Davies-Meyer feedforward is useless
  - Merkle-Damgård strengthening is useless
  - CV can be shorter than block length of cipher

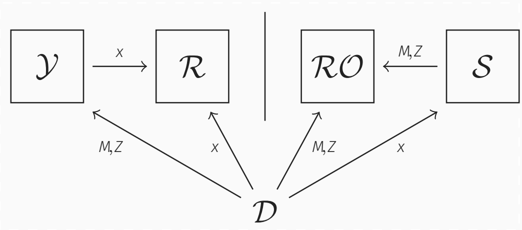
Thanks for your attention!

Intuition: why this works



- $(\mathcal{RO}, \mathcal{S})$ must act mode-consistent and it can:
  - Subtree-freeness $\rightarrow$ $\mathcal{A}$ can't learn CVs from (M, Z) queries
  - Radical-decodability $\rightarrow$ $\mathcal{S}$ can reconstruct any full tree S queried
  - Message-decodability $\rightarrow$ $\mathcal{S}$ can reconstruct M and Z from S
  - $\mathcal{S}$ then just queries $\mathcal{RO}$ with (M, Z) and forwards response to $\mathcal{A}$

Intuition: why this works



- $(\mathcal{RO}, \mathcal{S})$ must act mode-consistent and it can:
  - Subtree-freeness $\rightarrow$ $\mathcal{A}$ can't learn CVs from (M, Z) queries
  - Radical-decodability $\rightarrow$ $\mathcal{S}$ can reconstruct any full tree S queried
  - Message-decodability $\rightarrow$ $\mathcal{S}$ can reconstruct M and Z from S
  - $\mathcal{S}$ then just queries $\mathcal{RO}$ with (M, Z) and forwards response to $\mathcal{A}$
- Things break down when CVs collide

An example that is not radical-decodable