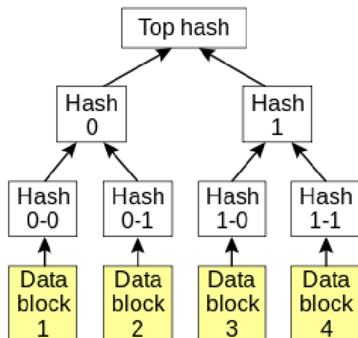
6: Hash Trees and Hash-Based Signatures
Hash Trees (Merkle, 1979)



(binary hash tree of depth 2)

similar to the iterated Merkle-Damgård (MD) structure but unlike MD, hash trees support parallel execution

Hash Trees in Detail (1)

Assume a compression function C. We can treat $C(0,\cdot)$, $C(1,\cdot)$ and $C(2,\cdot)$ like three different functions. This is "domain separation" and simplifies the analysis.

The tree hash function H^T first hashes all the message blocks M_i to $X_i = C(0, M_i)$ and then calls the tree function T:

$$H^T(M_1, \ldots, M_L) = T(C(0, M_1), \ldots, C(0, M_L), |M|).$$

Observe that the message length length $|M|$ is the last input for T.

Hash Trees in Detail (2)

The tree hash function T is defined recursively.

If there is only one X_1 for T, the top hash is computed:

$$T(X_1, \ell) = C(2, M_1, \ell).$$

Similarly to MD, we include the length ℓ into the final call to C.

If the number of input blocks is even, then hash all the pairs X_{2j-1}, X_{2j} to a new X_j and then call T recursively:

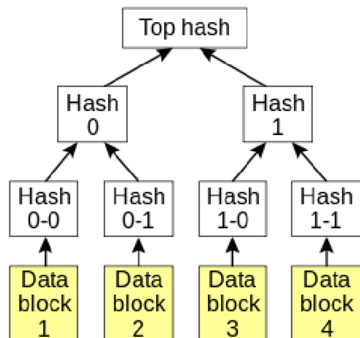
$$T(X_1, X_2, \ldots X_{2i}, \ell) = T(C(1, X_1, X_2), \ldots, C(1, X_{2i-1}, X_{2i}), \ell).$$

If the number of input blocks is odd and greater one, then do the same as for an even number of input blocks, except that the final input block X_{2i+1} is not hashed:

$$T(X_1, X_2, \ldots X_{2i+1}, \ell) = T(C(1, X_1, X_2), \ldots, C(1, X_{2i-1}, X_{2i}), X_{2i+1}, \ell).$$

Note that the final X_{2i+1} is renamed to the final X_{i+1}.

Hash Trees are as secure as Merkle-Damgård

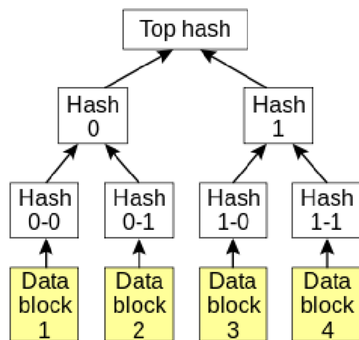


(binary hash tree of depth 2)

Theorem 10

If the compression function C is collision resistant (or preimage resistant, or 2nd preimage resistant), then so is the tree hash function H^T.

The Benefits of Hash Trees



(binary hash tree of depth 2)

- easily parallelizable
- one can verify a hash without having to know all the data (how?)
- one can efficiently compute a new hash if only a small part of the data has been changed (how?)
- we can use the compression function of any "conventional" hash function
- with very similar security

? But why do all "conventional" hash functions apply the compression function iteratively, i.e., perform sequential hashing, without the option to parallelize the operation?

One-Time Signatures

- Consider a fixed command **C** (“drop the bomb” or “sell all shares from Deutsche Bank” or ...) that Alice may send to Bob, some time.
- Bob needs some “authenticator” to be sure Alice requires him to perform **C** not Eve.
- Furthermore, if Alice later accuses him of having acted without her permission, he will use the authenticator to prove that he has been authorized by Alice.
- How can we do that, using a hash function?

Hash-Based Authenticators

- Alice chooses a secret $\mathbf{X_M}$ and computes a public $\mathbf{Y_C} = H(\mathbf{X_C})$.
- The deal with Bob: When challenged to prove that **C** has been authorized by Alice, he has to show any $\mathbf{X}^*$ with $\mathbf{Y_C} = H(\mathbf{X}^*)$.
- Alice's security depends on neither Bob nor anyone else (but her) being able to provide such a value $\mathbf{X}^*$. What is the requirement for H, then?

Lamport Signatures (1979)

- Let H be an n-bit hash function.
- Generate $2n$ secrets $\mathbf{X_{i,b}}$ and $2n$ public values $\mathbf{Y_{i,b}} = H(\mathbf{X_{i,b}})$ with $i \in \{1, \ldots, n\}$ and $b \in \{0, 1\}$.
- Use the n values $\mathbf{X_{i,b}}$ as signatures for "the i-th bit of $H(M)$ is b".
- Make sure that the $\mathbf{X_{i,1-b}}$ are never compromised.
- Specifically: The $(2n^2)$-bit secret key $\mathbf{X_{i,b}}$ (for $i \in \{1, \ldots, n\}$) must not be used to sign more than one single message.

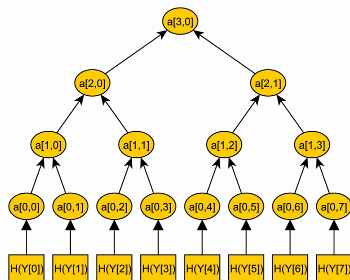
Theorem 11

If H is preimage and collision resistant, a secret key is used to sign one message only, these signatures are secure against existential forgery attacks under one chosen message.

- If a secret key is used twice to sign two different $H(M)$ and $H(M')$, how could you exploit this to forge another signature?

Merkle's Tree Signatures

Combines the Ideas of One-Time Signatures and Hash Trees



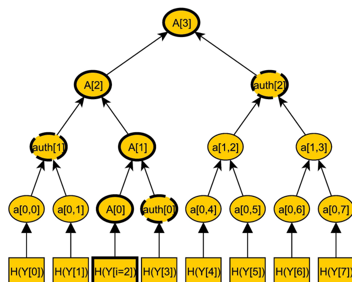
key generation:

1. generate a^d Lamport key pairs (sk_i, pk_i)
2. compute a tree-hash of the 2^d Lamport keys
3. the root public key PK is the result from the tree hash

Bauhaus-Universität Weimar

Merkle's Tree Signatures (2)

Combines the Ideas of One-Time Signatures and Hash Trees

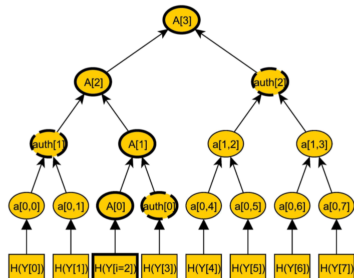


signature generation:

1. pick unused key pair (pk_i, sk_i)
2. sign msg using sk_i.
3. signature:
   - the Lamport signature
   - the public Lamport key pk_i
   - and *d* hash values to verify the path from pk_i to the root

Merkle's Tree Signatures (3)

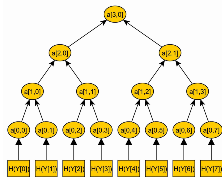
Combines the Ideas of One-Time Signatures and Hash Trees



signature verification:

- Given PK, msg, signature, pk_i, i:
  1. Do the hashes $H(X_{i,b}$ of the Lamport signature parts match the public Lamport key $Y_{i,b}$ from pk_i?
  2. Use pk_i and the d hash values to compute an “independent” root R of the Merkle tree!
  3. Does the “independent” root match the public root key, i.e., is R=PK?

Tree Signatures: Pros and Cons



- \+ simple
- \+ brings signatures to low-end systems (no need to implement big-integer-arithmetic or the like)
- \+ a single cryptographic assumption (unlike classical hash-then-sign signatures, based, e.g., on RSA)
- \+ post-quantum secure (unlike RSA, ElGamal, ect.)
- – limited number of messages to be signed
- – insecure if any leaf is used more than once
- – fast signing seems to require storing the entire tree
- – low-memory signing seems to require full tree recomputation

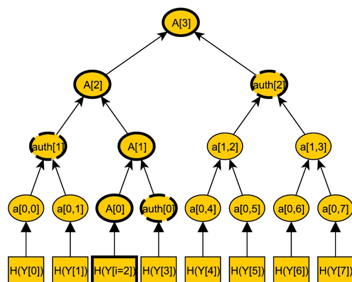
Efficient Implementations of Merkle's Tree Signatures

- a single symmetric key **K**
- the secrets $\mathbf{X_{i,j,k}} := H(\mathbf{K}||i||j||k)$ can be computed on demand

? but what about the internal hash values:

? store all intermediate hash values (too much memory)

? compute all the intermediate hash values on demand (too slow)



- Merkle's tree-traversal for depth-d-tree:
  - store $O(d^2)$ intermediate hash values
  - compute $O(d)$ fresh intermediate values for each new signature (on the average)
- 2004: Szydlo reduced the storage to $O(d)$