Hash-Based Signatures

CASED

Johannes Buchmann, Andreas Hülsung
Supported by DFG and DAAD

Part VIII:
Authentication Path Generation

Tree Traversal Algorithms

CASED

How to Compute Authentication Path Nodes?

A CERTIFIED DIGITAL SIGNATURE

Ralph C. Merkle
Xerox PARC
3333 Coyote Hill Road,
Palo Alto, Ca. 94304
merkle@xerox.com
(Subtitle: That Antique Paper from 1979)

Merkle Tree Traversal in Log Space and Time

Michael Szydlo

RSA Laboratories, Bedford, MA 01730. mszydlo@rsasecurity.com

Fractal Merkle Tree Representation and Traversal

Markus Jakobsson[1], Tom Leighton[2,3], Silvio Micali[3], and Michael Szydlo[1]

Merkle tree traversal revisited

Johannes Buchmann, Erik Dahmen, and Michael Schneider

Technische Universität Darmstadt
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
{buchmann,dahmen,mischnei}@cdc.informatik.tu-darmstadt.de

Optimal Trade-Off for Merkle Tree Traversal

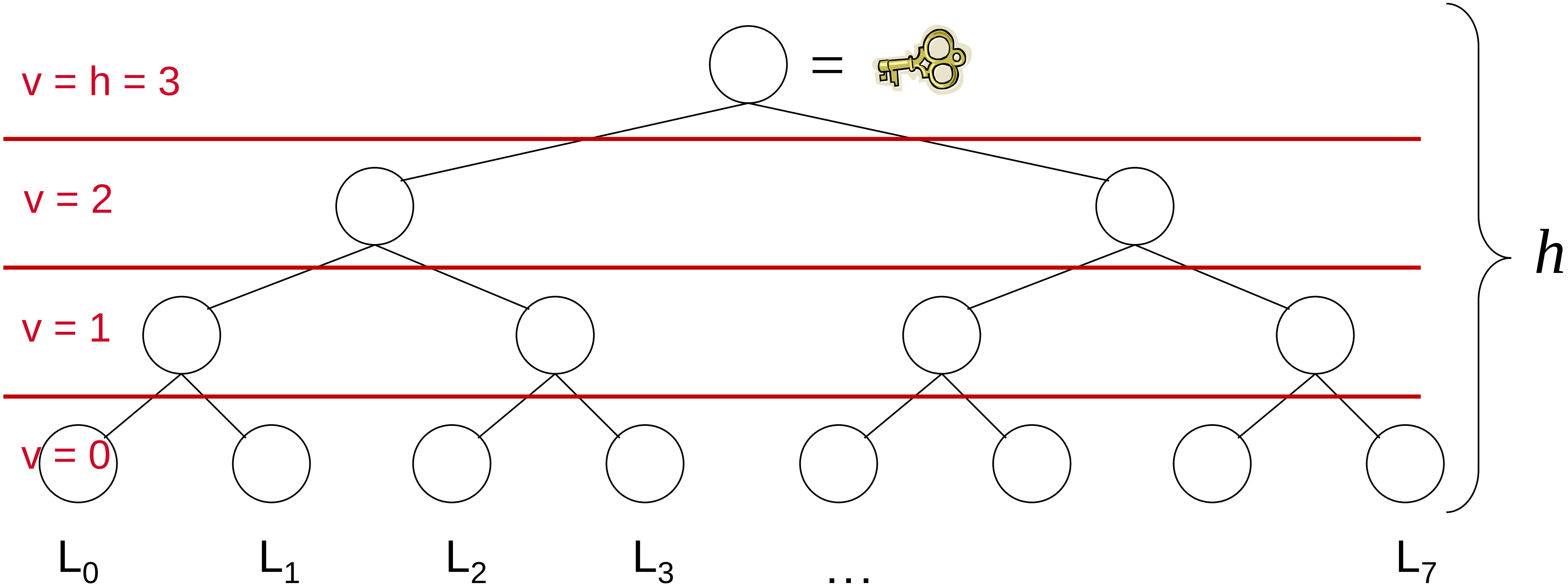
Piotr Berman[1,⋆], Marek Karpinski[2,⋆⋆], and Yakov Nekrich[2,⋆⋆⋆]

TreeHash

(Mer89)

TreeHash

- TreeHash(v,i): Computes node on level v with leftmost descendant L_i
- Public Key Generation: Run TreeHash(h,0)



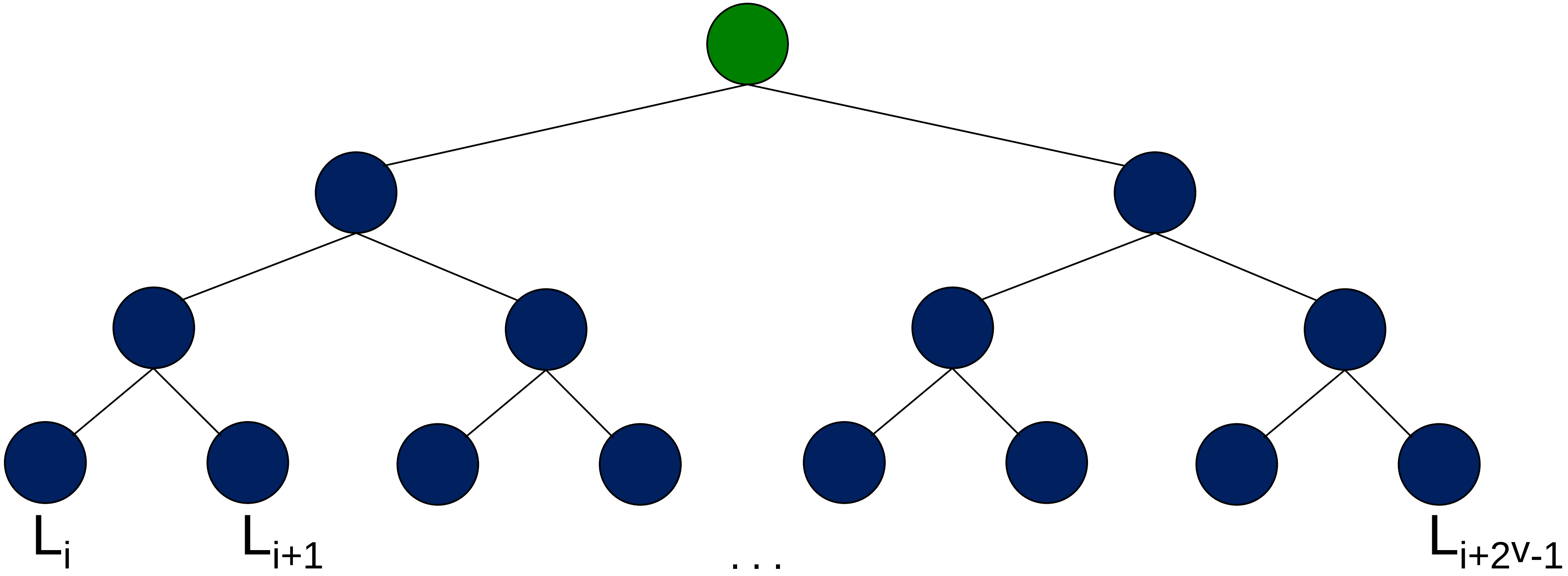
TreeHash

TreeHash(v,i)

```
Init Stack, N1, N2
For j = i to i+2^v-1 do
        N1 = LeafCalc(j)
        While N1.level() == Stack.top().level() do
                N2 = Stack.pop()
                N1 = ComputeParent( N2, N1 )
        Stack.push(N1)
Return Stack.pop()
```

TreeHash

TreeHash(v,i)



Efficiency?

Key generation: Every node has to be computed once.

cost = 2^h leaves + 2^h-1 nodes

=> optimal

Signature: One node on each level 0 <= v < h.

cost 2^h-1 leaves + 2^h-1-h nodes.

Many nodes are computed many times!

(e.g. those on level v=h-1 are computed 2^{h-1} times)

-> Not optimal if state allowed

The BDS Algorithm

Motivation
(for all Tree Traversal Algorithms)

CASED

No Storage:
Signature: Compute one node on each level $0 <= v < h$.
Costs: 2^h-1 leaf + 2^h-1-h node computations.

Example: XMSS with SHA2 and $h = 20$ ~25min

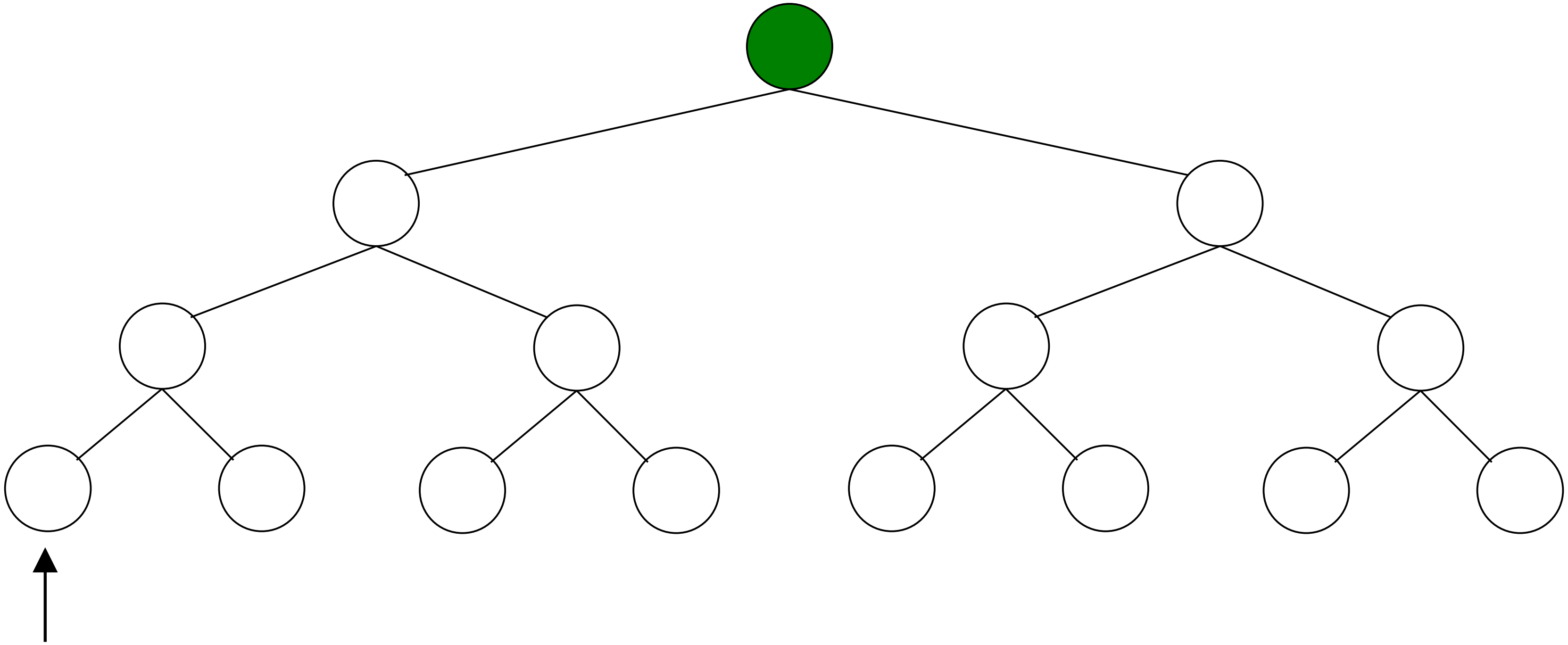
Store whole tree: $2^h n$ bits.

Example: $h=20$, $n=128$; storage: 2^{28}bits = 32MB

Idea: Look for time-memory trade-off!

Use a State

Authentication Paths



Observation 1

CASED

Same node in authentication path is recomputed many times!

Node on level v is recomputed for 2^v successive paths.

Idea: Keep authentication path in state.

-> Only have to update "new" nodes.

Result

Storage: h nodes

Time: ~ h leaf + h node computations (average)

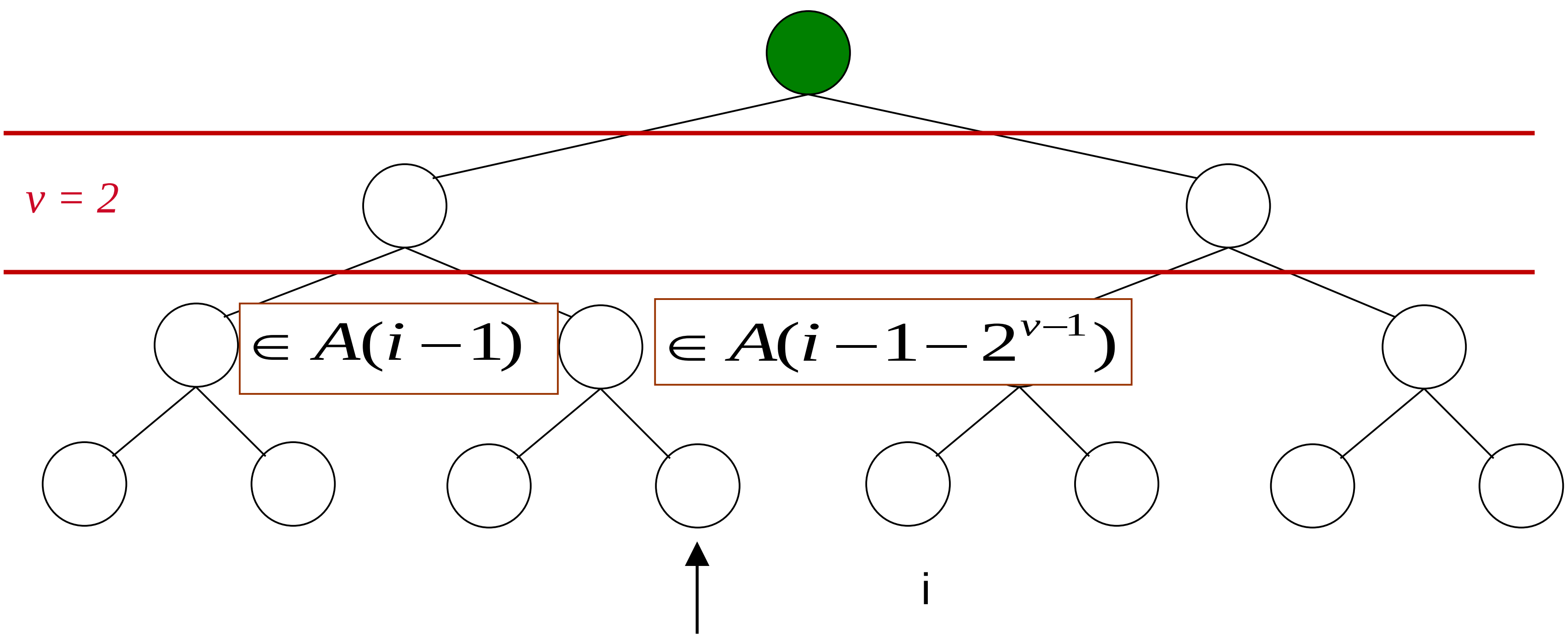
But: Worst case still 2^h-1 leaf + 2^h-1-h node computations!

-> Keep in mind. To be solved.

Observation 2

When new left node in authentication path is needed, its children have been part of previous authentication paths.

Computing Left Nodes



Result

Storing $\left\lceil \frac{h}{2} \right\rceil$ nodes

all left nodes can be computed with one node computation / node

Observation 3

Right child nodes on high levels are most costly.

Computing node on level v requires
2^v leaf and 2^v-1 node computations.

Idea: Store right nodes on top k levels during key generation.

Result
Storage: 2^k-2 n bit nodes
Time: ~ h-k leaf + h-k node computations (average)

Still: Worst case $2^{h-k}-1$ leaf + $2^{h-k}-1-(h-k)$ node computations!

Distribute Computation

Intuition

CASED

Observation:

- For every second signature only one leaf computation
- Average runtime: ~ h-k leaf + h-k node computations

Idea: Distribute computation to achieve average runtime in worst case.

Focus on distributing computation of leaves

TreeHash with Updates

CASED

TreeHash.init(v,i)
1: Init Stack, N1, N2, j=i, j_max = i+2^v-1
2: Exit

TreeHash.update()
1: If j <= j_max
2: N1 = LeafCalc(j)
3: While N1.level() == Stack.top().level() do
5: N2 = Stack.pop()
6: N1 = ComputeParent(N2, N1)
7: Stack.push(N1)
8: Set j = j+1
9: Exit

One leaf per update

Distribute Computation

Concept

- Run one TreeHash instance per level $0 <= v < h-k$
- Start computation of next right node on level v when current node becomes part of authentication path.
- Use scheduling strategy to guarantee that nodes are finished in time.
- Distribute $(h-k)/2$ updates per signature among all running TreeHash instances

Distribute Computation

CASED

Worst Case Runtime

Before:
$2^{h-k}-1$ leaf and $2^{h-k}-1-(h-k)$ node computations.

With distributed computation:
$(h-k)/2 + 1$ leaf and $3(h-k-1)/2 + 1$ node computations.

Add. Storage

Single stack of size h-k nodes for all TreeHash instances.
\+ One node per TreeHash instance.
= 2(h-k) nodes

BDS Performance

CASED

Storage:

$$3h + \left\lfloor \frac{h}{2} \right\rfloor - 3k - 2 + 2^k \; n \text{ bit nodes}$$

$+2h-2k$ n bit seeds for forward secure XMSS.

Runtime:

$(h-k)/2+1$ leaf and

$3(h-k-1)/2+1$ node computations.

$+(h-k)$ calls to FSPRG for forward secure XMSS in the worst case.