Tree-Based Management of Revoked Certificates in Vehicular Ad-hoc Networks

Pino Caballero-Gil, Francisco Martín-Fernández, and Cándido Caballero-Gil

***Abstract*—A dynamic authenticated data structure based on k-ary trees is here proposed to improve the performance of certificate revocation in vehicular ad-hoc networks. Such a structure allows taking advantage of a duplex construction of the new standard SHA-3. In particular, efficient algorithms for search, insertion, deletion and restructuring the used k-ary trees are presented. This is a work in progress, and in the near future an implementation of a proof-of-concept prototype for smartphones will be available.**

***Index Terms*—k-ary tree, certificate revocation, VANET, hash function.**

I. Introduction

SECURITY is a crucial requirement in any communication network. In particular, being able to identify and exclude misbehaving nodes from the network is absolutely necessary to guarantee trustworthiness of network services. One of the basic solutions to accomplish this task in networks where communications are based on a Public Key Infrastructure (PKI) is the use of certificate revocation. Thus, a critical part in such networks is the management of revoked certificates. Related to this issue, in the bibliography we can find two different types of solutions. On the one hand, a decentralized proposal enables revocation without the intervention of any centralized infrastructure, based on trusting the criteria of network nodes. On the other hand, a centralized approach is based on the existence of a central Certificate Authority (CA), which is the only entity responsible for deciding on the validity of each node certificate, and all nodes trust it. This second approach is usually based on the distribution of the so-called Certificate Revocation Lists (CRLs), which can be seen as blacklists of revoked certificates.

Vehicular Ad-hoc NETworks (VANETs) are self-organizing networks built up from moving vehicles that communicate with each other mainly to prevent adverse circumstances on the roads, but also to achieve a more efficient traffic management. In particular, these networks are considered an emerging research area of mobile communications because they offer a wide variety of possible applications, ranging from road safety and transport efficiency, to commercial services, passenger comfort, and infotainment delivery. Furthermore, VANETs can be seen as an extension of mobile ad-hoc networks where there are not only mobile nodes, there called On-Board Units (OBUs), but also static nodes, which are the so-called Road-Side Units (RSUs) [**?**].

Research supported by Spanish MINECO and FEDER under projects TIN2011-25452 and IPT-2012-0585-370000, and FPI scholarships BES-2012-051817 and BES-2009-016774.

Department of Statistics, Operations Research and Computing. University of La Laguna. SPAIN Email: pcaballe@ull.es, francisco.martin.07@ull.edu.es, ccabgil@ull.es

IEEE 1609 is a family of standards based on the approved standard IEEE 802.11p for vehicular communications. Within such a family, 1609.2 deals with the issues related to security services for applications and management messages. In particular, the IEEE 1609.2 standard defines the use of PKIs, CAs and CRLs in VANETs, and implies that in order to revoke a vehicle, a CRL has to be issued by the CA to the RSUs, who are in charge of sending the information to the OBUs. Thus, an efficient management of certificate revocation is crucial for the robust and reliable operation of VANETs.

Once VANETs are implemented in practice on a large scale, their size will grow and the use of multiple temporary certificates or pseudonyms will become necessary to protect the privacy of the users. Thus, it is foreseeable that CRLs will grow up to become very large. Moreover, in this context it is also expected a phenomena known as implosion request, consisting of several nodes who synchronously want to download the CRL at the time of its updating, producing serious congestion and overload of the network, what could ultimately lead to a longer latency in the process of validating a certificate.

This proposal uses a k-ary tree as an Authenticated Data Structure (ADS), for the management of certificate revocation in VANETs. By using this ADS, the process of query on the validity of certificates will be more efficient because OBUs will send queries to RSUs, who will answer them on behalf of the CA. In this way, at the same time the CA will no longer be a bottleneck, and OBUs will not have to download the entire CRL. In particular, the used perfect k-ary trees are based on the application of a duplex construction of the Secure Hash Algorithm SHA-3 that was recently chosen as standard [**?**], because the combination of both structures allows improving efficiency of updating and querying of revoked certificates.

This paper is organized as follows. Section 2 focuses on the necessary preliminaries while Section 3 provides a brief explanation of our proposal based on the combination of perfect k-ary trees and a duplex construction of the Secure Hash Algorithm SHA-3. Finally, Section 4 discusses conclusions and possible future research lines.

II. Preliminaries

In order to improve efficiency of communication and computation in the management of revoked public-key certificates in VANETs, some authors have proposed the use of particular ADSs such as Merkle trees [**?**] and skip lists [**?**] [**?**]. However, to the best of our knowledge no previous work has described in detail the use of k-ary trees in general as hash trees for revoked certificate management.

In general, a hash tree is a tree structure whose nodes contain digests that can be used to verify larger pieces of

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

data. The leaves in a hash tree are hashes of data blocks while nodes further up in the tree are the hashes of their respective children so that the root of the tree is the digest representing the whole structure. Most implemented hash trees require the use of a cryptographic hash function in order to prevent collisions.

Most hash tree implementations are binary, but this work proposes the use of a more general structure known as k-ary tree, which is a rooted tree in which each node has no more than k children. Specifically we propose the use of a perfect k-ary tree in which all leaf nodes are at the same depth. Thus, one of the major drawbacks of ordered tree structures, which is the necessary restructuring when there are changes in the tree, only occurs when the perfect k-ary tree requires a new level of depth, because otherwise the nodes simply are inserted from left to right to complete each level of depth. In this way, our proposal is based on a dynamic tree-based data structure that varies depending on the number of revoked certificates.

The authenticity of the used hash tree structure is guaranteed thanks to the CA signature of the root. When an RSU has to respond to an OBU about a query on a certificate, it proceeds in the following way. If it finds the digest of the certificate among the leaves of the tree because it is a revoked certificate, then the RSU sends to the OBU the route from the root to the corresponding leaf, along with all the siblings of the nodes on this path. After checking all the digests corresponding to the received path and the CA signature of the root, the OBU gets convinced of the validity of the evidence on the revoked certificate received from the RSU.

III. Our Proposal

The proposed model is based on the following notation:

- h: Cryptographic hash function used to define the hash tree.
- D (≥ 1): Depth of the hash tree.
- d ($< D$): Depth of an internal node in the hash tree.
- s: Number of revoked certificates.
- RC_j ($j = 1, 2, ..., s$): Serial number of the $j - th$ Revoked Certificate.
- N_{ij} ($i = D - d$ and $j = 0, 1...$): Internal Node of the hash tree obtained by hashing the concatenation of all the digests contained in its children.
- N_{0j} ($j = 0, 1...$): Leaf node of the hash tree containing $h(RC_j)$, ordered according to revocation.
- k: Maximum number of children for each internal node in the hash tree.
- f: Basic cryptographic hash function of SHA-3, called Keccak.
- n: Bit size of the digest of h, which is here assumed to be the lowest possible size of SHA-3 digest, 224.
- b: Bit size of the input to f, which is here assumed to be one of the possible values of Keccak, 800.
- r: Bit size of input blocks after padding for h, which is here assumed to be 352.
- c: Difference between b and r, which is here assumed to be as in SHA-3, $2n$, that is 448.
- l: Bit size of output blocks for building the digest of h, which is here assumed to be lower than r.

Regarding the cryptographic hash function h used in the hash tree (see Figure **??**), our proposal is based on the use of a new version of the Secure Hash Algorithm SHA-3. The padding of the input is a minimum $10*1$ pattern that consists of a 1 bit, zero or more 0 bits (maximum $r - 1$) and a final 1 bit, and the basic cryptographic hash function f called Keccak [**?**] contains 24 rounds of a basic transformation that involves 5 steps called theta, rho, pi, chi and iota, and the input is represented by a 5×5 matrix of 64-bit lanes, but our proposal is based on 32-bit lanes.

Another proposed variation of SHA-3 is the combination of a duplex version of the sponge structure of SHA-3 [**?**] and a hash k-ary tree. On the one hand, like the sponge construction of SHA-3, our proposal based on a duplex construction also uses Keccak as fixed-length transformation f, the same padding rule based on the $10*1$ pattern, and data bit rate r. On the other hand, unlike a sponge function, the duplex construction output corresponding to an input string might be obtained through the concatenation of the outputs resulting from successive input blocks (see Figure **??**). Thus, the use of the duplex construction in our proposed hash tree allows the insertion of a new revoked certificate as new leaf of the tree by running a new iteration of the duplex construction only on the new revoked certificate. In particular, the RSU can take advantage of all the digests corresponding to the sibling nodes of the new node, which were computed in previous iterations, by simply discarding the same minimum number of the last bits of each one of those digests so that the total size of the resulting digest of all the children remains the same, n.

While the maximum number of children of an internal node has not been reached, the RSU has to store not only all the digests of the tree structure but also the state resulting from the application of Keccak hash function f in the last iteration corresponding to such internal node, in order to use it as input in a next iteration.

Periodic delete operations of certificates that are in the tree and reach their expiration date, require rebuilding the part of the tree involving the path from those nodes to the root. Thus, in order to maximize our proposal, such tree rebuilding is proposed to be linked to the moment when all the sibling nodes of some internal node are expired because this avoids unnecessary reductions of the system efficiency by having to rebuild the tree very often.

The choice of adequate values for the parameters in our proposal must be done carefully, taking into account the relationships among them. In particular, the maximum bit-length of the tree identifier of each revoked certificate, the maximum tree size takes the following value:

$$n(1 + k + k^2 + k^3 + \cdots + k^D) = \frac{n(k^{D+1} - 1)}{k - 1}$$

Thus, since this quantity is upperbounded by the size of available memory in the RSU, and the maximum number of leaves of the k-ary tree k^D is lowerbounded by the number of revoked certificates s, then both conditions can be used to deduce the optimal value for k.

In our proposal, the used k-ary tree structure assigns a unique identifier to each revoked certificate to represent it in in order in each one of its leafs. Consequently, an auxiliary

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

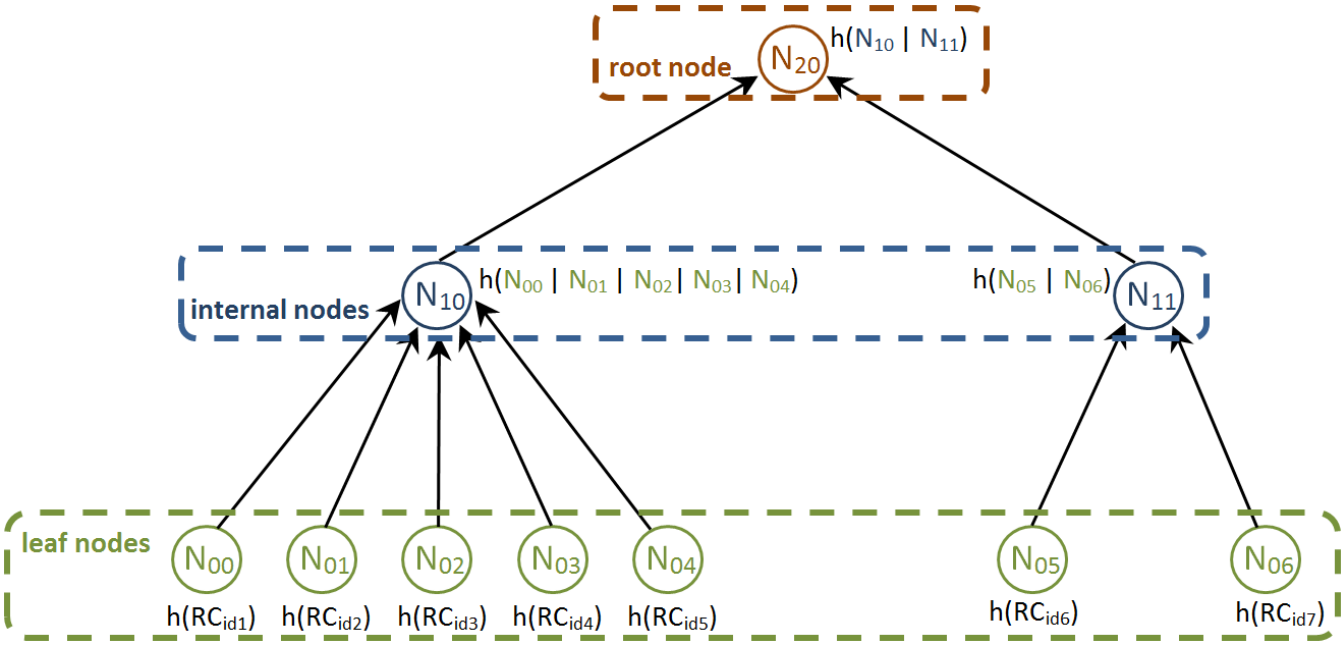

Fig. 1. Hash Tree Based on a Perfect 5-ary Tree

structure linking such identifiers with the corresponding certificate serial number is also stored in the RSU. Thus, when an OBU sends a request about a certificate, the RSU first gets from such structure the identifier generated by the k-ary tree structure using the certificate serial number, and then proceeds with the tree search.

In order to allow the ordered insertion of revoked certificates, an auxiliary structure defined as a hash table is used as a quick and efficient way to return the order value required for the search in the tree. This structure is first generated by the CA and then sent to all RSUs so that they can properly perform searches of requested certificates through the following algorithm:

Search Algorithm
//Param: rcSearch, Id-Tree of Leaf Node to Search
//Return: retPath, Path to Reconstruct Root Node
function searchTree (int $rcSearch$)
01: **if** ($rcSearch > s$)
02: return [][]; // Not found
03: **endif**
04: int $depth$=0;
05: Node $retPath$[][] = new Node[D][k];
06: $retPath$[$depth$++][0]=$rNode$;
07: int $path$ = ($\lceil \frac{idNew}{k^{D-1}} \rceil - 1$) **mod** k;
08: INode $iNode$=$rNode$.gNode($path$);
09: $retPath$[$depth$++]=$iNode$.gMeAndBrothers();
10: **for** (int $it = (D-2); it > 0, it--$)
11: $path$ = ($\lceil \frac{idNew}{k^{it}} \rceil - 1$) **mod** k;
12: $iNode$=$iNode$.gNode($path$);
13: $retPath$[$depth$++]=$iNode$.gMeAndBrothers();
14: **endfor**
15: int $posLeaf$=($rcSearch - 1$) **mod** k;
16: LeafNode $leaf$ = $iNode$.gLeaf($posLeaf$);
17: $retPath$[$depth$]=$leaf$.gMeAndBrothers();
18: return $retPath$;
endfunction

The insertion operation in the tree implies having to update the tree not as often as in other proposals (see Figure **??**). The number of internal nodes to be updated will depend on whether the insertion implies a new level of the tree or not. Thus, the number of updates that are necessary to insert a leaf node without involving a new level of depth in the tree is $\lceil log_k s \rceil$.

On the other hand, if it is necessary to increase the depth of the tree due to the insertion of a new node, our proposal simply creates a new root node above so that the existent tree becomes a subtree, and a new internal node is inserted so that the new leaf node is at the same depth of the other leaf nodes. This implies that the number of updates necessary is defined by $\lceil log_k s \rceil + 1$.

The pseudocode of the insertion is shown below:

Insertion Algorithm
//Param: rcNew, Id-Tree of Leaf Node to Insert
function insertTree (int $rcNew$)
01: LeafNode $nLeaf$ = **new** LeafNode($rcNew$);
02: **if** ($D > 0$)
03: **if** ($rcNew > k^D$) *// New depth level*
04: INode $oldLevel$[k]=$rootNode$.gNLevel();
05: $rNode$.sNLevel(**new** INode[k]);
06: $rNode$.gNode(0).sNLevel($oLevel$);
07: D++;
08: **endif**
09: int $path$ = ($\lceil \frac{rcNew}{k^{D-1}} \rceil - 1$) **mod** k;
10: INode $iNode$=$rNode$.gNode($path$);
11: **for** (int $it = (D-2); it > 0, it--$)
12: **if** (!$iNode$.existNLevel())
13: $iNode$.sNLevel(**new** INode[k]);
14: **endif**
15: $path$ = ($\lceil \frac{rcNew}{k^{it}} \rceil - 1$) **mod** k;
16: $iNode$=$iNode$.gNode($path$);
17: **endfor**
18: **if** (!$iNode$.existLeavesNodes())
19: $iNode$.sLeaves(**new** LeafNode[k]);

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

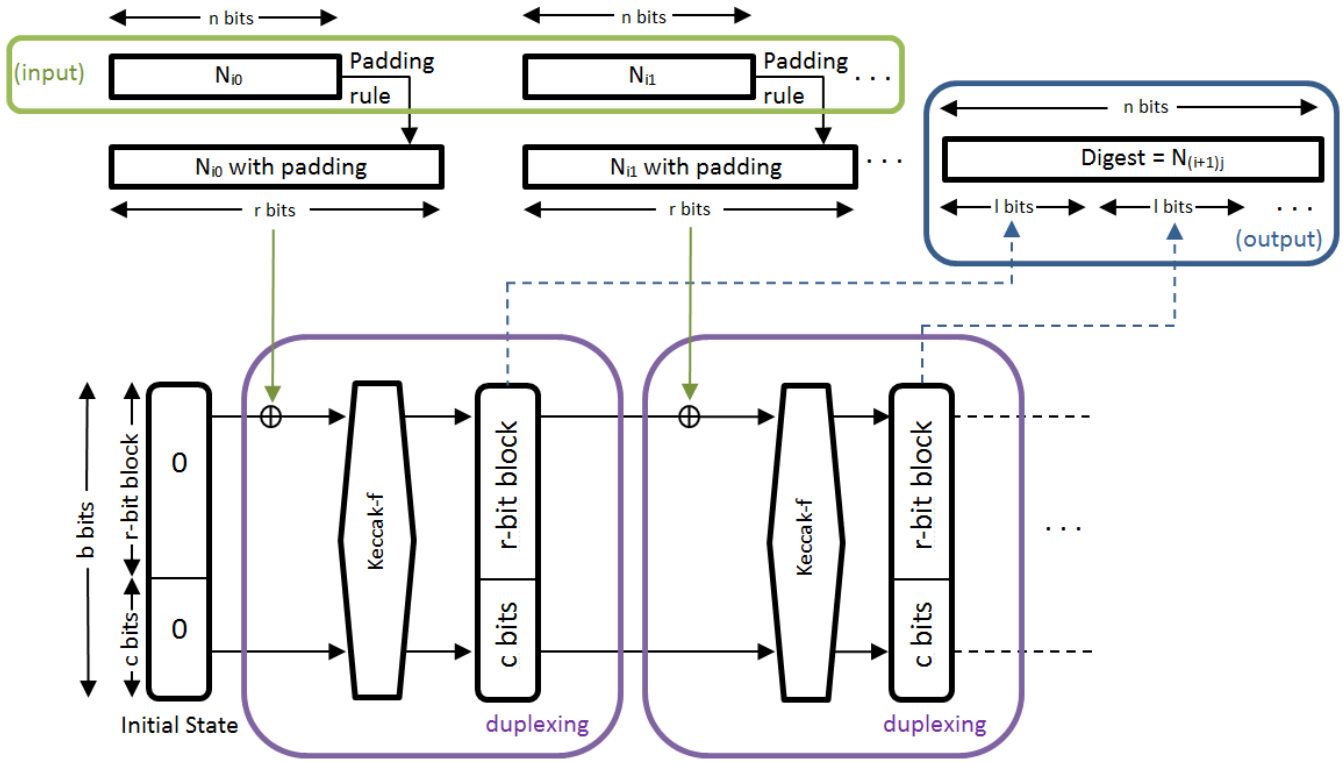

Fig. 2. Proposed Duplex Construction

20: **endif**
21: int $posLeaf$ = $(rcNew - 1)$ **mod** k;
22: $iNode$.sLeaf($posLeaf$, $nLeaf$);
23: **endif**
24: **else** *// First Leaf Node of the k-ary tree*
25: $rNode$.sNLevel(**new** INode[k]);
26: Inode $iNode$ = $rNode$.gNode(0);
27: $iNode$.sLeaves(**new** LeafNode[k]);
28: $iNode$.setLeaf(0, $nLeaf$);
29: $D = 2$;
30: **endelse**
31: *// Recalculate All Affected Parent Nodes*
32: Node $upNode$ = $nLeaf$;
33: **while** (!$upNode$.equals(rNode))
34: $upNode$=$upNode$.gParent();
35: $upNode$.update();
36: **endwhile**
endfunction

When necessary, leaf nodes are deleted from the network if all its siblings have to be also deleted (see Figure **??**). In that case, the full subtree of all the siblings is deleted according to the following algorithm:

Deletion Algorithm
//Param: rcDelete, Id-Tree of Leaf Node to Delete
function deleteTree (int $rcDelete$)
01: **if** ($rcDelete > s$)
02: return; // Not found
03: **endif**
04: $path$ = $(\lceil \frac{rcNew}{k^{D-1}} \rceil - 1)$ **mod** k;
05: INode $iNode$=$rNode$.gNode($path$);
06: **for** (int $it = (D-2); it > 0, it--$)
07: $path$ = $(\lceil \frac{rcNew}{k^{it}} \rceil - 1)$ **mod** k;
08: $iNode$=$iNode$.gNode($path$);
09: **endfor**
10: int $posLeaf$ = $(rcNew - 1)$ **mod** k;
11: LeafNode $leaf$ = $iNode$.gLeaf($posLeaf$);
12: $leaf$.sDisable();
13: **if** ($iNode$.allLeavesDisable())
14: *// It really eliminates that branch*
15: INode $upNode$ = $iNode$.gParent();
16: $upNode$.removeNode($iNode$);
17: recTreeFrom($upNode$, $rcDelete$);
18: **endif**
endfunction

The computational cost of the delete operation is due to the necessary reconstructions. A key aspect to consider is that when the tree is restructured, the structure that maps the certificate number with the identifier in the k-ary tree is also updated. The number of necessary updates to be performed in the internal nodes is:

$$(\lceil log_k s \rceil - 1) + \sum_{j=1}^{k} (\left\lceil \frac{s}{k^j} \right\rceil - \left\lceil \frac{RC_{AnyLeafNodeRemove}}{k^j} \right\rceil)$$

If removing a subtree leads to th deletion of a higher subtree containing such subtree, the number of updates is reduced by as many superior subtrees as eliminated. Besides, if removing a subtree means reducing the depth of the tree, the number of updates is also reduced by 1 single update, corresponding to the root node.

Restructuring Algorithm
//Param: iNode, Branch Belonging to the rcDelete
// rcDelete, Id-Tree of Leaf Node to Delete
function recTreeFrom (Node $iNode$, int $rcDelete$)
01: int d=$iNode$.gDepth();

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

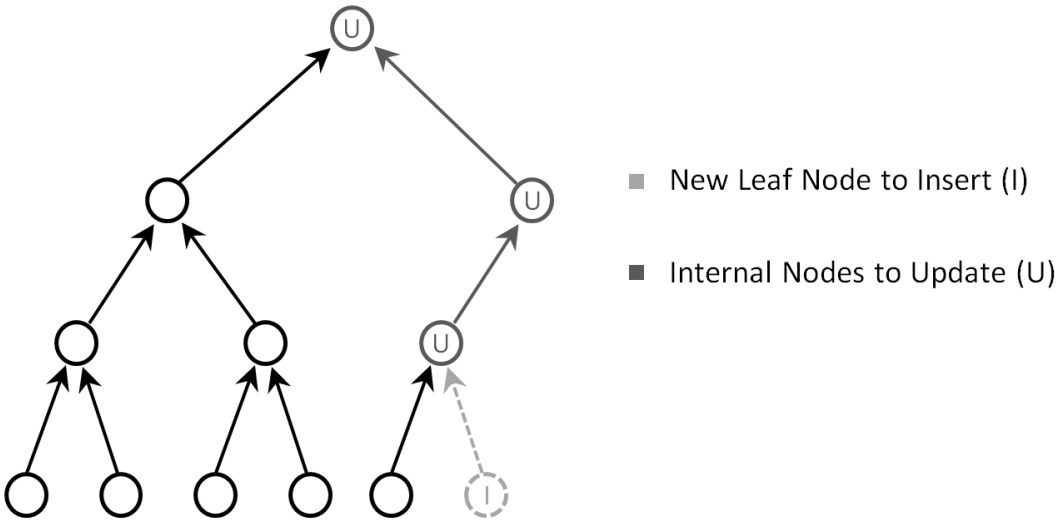

Fig. 3. Example of Insertion

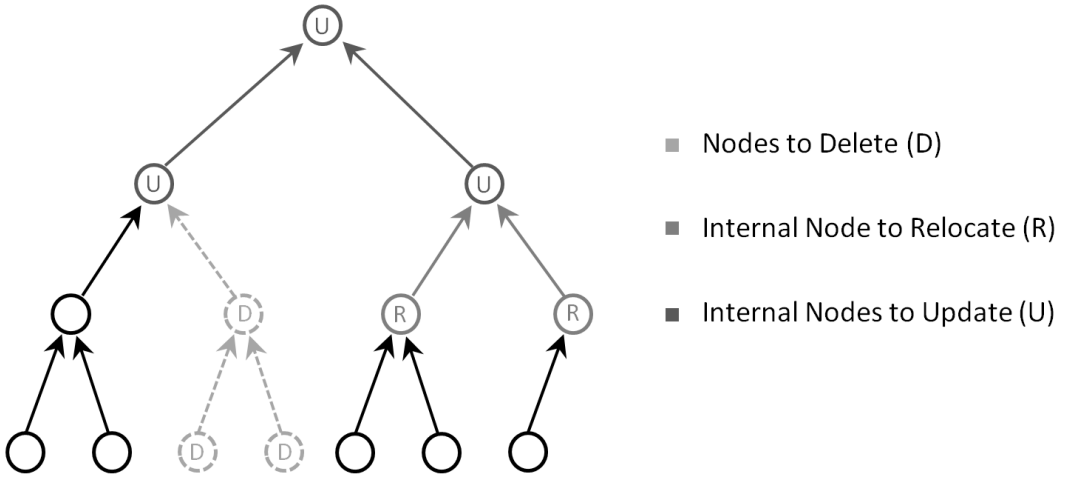

Fig. 4. Example of Deletion

```
02: int me = (⌈rcDelete / k^(D−d)⌉);
03: int total = (⌈s / k^(D−d)⌉);
04: Node nNode, auxNode;
05: for (int it = (me + 1); it <= total; it++)
06:   // Find Node in 'it' position in 'd' depth
07:   nNode = findNode(it, d);
08:   // Restructure intern branch of iNode
09:   // using branchs of nNode, if it is necessary
10:   moveBranch(iNode, nNode);
11:   iNode.update();
12:   // Upgrade Nodes Parents not change more
13:   // from iNode
14:   updateUpIfIsPossible(iNode, it);
15:   iNode = nNode;
16: endfor
17: if (!iNode.hasNodes())
18:    while(
         (!iNode.gParent().equals(rNode)) &&
         (iNode.gParent().nNodes() != 1)
         )
19:      auxNode = iNode;
20:      iNode = iNode.gParent();
21:      iNode.removeNode(auxNode);
22:   endwhile
23:   if (rNode.equals(iNode))
24:      auxNode = iNode;
25:      rNode = rNode.gNode(0);
26:      iNode.delete();
27:      rNode.update();
28:      return;
29:   endif
30:   else
31:      iNode.gParent().removeNode(idNode);
32:   endelse
32: endif
33: else
34:   moveBranch(iNode);
35: endelse
36: iNode.update();
37: while (!iNode.equals(rNode))
38:   iNode = iNode.gParent();
39:   iNode.update();
40: endwhile
endfunction
```

IV. Conclusions

One of the most important security issues in VANETs is the problem of certificate revocation management, so an

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

efficient verification of public-key certificates by OBUS is crucial to ensure the safe operation of the network. However, as VANETs grow, certificate revocation lists will also grow, making it impossible their issuance. This paper proposes a more efficient alternative to CRL distribution, which uses an authenticated data structure based on dynamic k-ary tree. In addition, the proposed mechanism applies the basic hash function of the new SHA-3 standard called Keccak combined with a duplex construction. Thanks to the structure of the used k-ary tree, the duplex construction allows taking advantage of the digests of previous revoked certificates for calculating the hash of every new revoked certificate, so that its inclusion in the tree can be performed by a single iteration of the hash function. Both the analysis of optimal values for the parameters, a comparison with previous proposals, and the implementation of the proposal both on VANET devices and on Android and iOS smartphones are part of work in progress.

REFERENCES

[1] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, *Keccak sponge function family main document version 2.1*, Updated submission to NIST (Round 2), 2010.

[2] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, *Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications*, Selected Areas in Cryptography, pp. 320-337, 2011.

[3] S. Chang, R. Perlner, W. Burr, M. Turan, J. Kelsey, S. Paul, L. Bassham, *Third-Round Report of the SHA-3 Cryptographic Hash Algorithm Competition*, nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf, 2012.

[4] C. Ganan, J. Munoz, O. Esparza, J. Mata-Diaz, J. Alins, *Toward Revocation Data Handling Efficiency in VANETs*, Communication Technologies for Vehicles, Lecture Notes in Computer Science 7266, pp. 80-90, 2012.

[5] C. Hernandez-Goya, P. Caballero-Gil, J. Molina-Gil, C. Caballero-Gil, *Cooperation Enforcement Schemes in Vehicular Ad-Hoc Networks*, Computer Aided Systems Theory (EUROCAST 2009), Lecture Notes in Computer Science 5717, pp. 429-436, 2009.

[6] M. Jakobsson, S. Wetzel, *Efficient attribute authentication with applications to ad hoc networks, ACM international workshop on Vehicular ad hoc networks*, pp. 38-46, 2004.

[7] R.C. Merkle, *Protocols for public key cryptosystems*. IEEE Symposium on Security and privacy 1109, pp. 122-134, 1980.

ISBN: 978-988-19252-8-2
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)