Data Structures and Lookup Algorithms Investigation for the IEEE 802.15.4 Security Procedures Implementation

Viktor Melnyk [a,b]

[a] *John Paul II Catholic University of Lublin, Konstantynow str., 1H/313, Lublin, 25-708, Poland*
[b] *Lviv Polytechnic National University, 12, S. Bandery str., Lviv, 79013, Ukraine*

Abstract

This paper investigates data structures and lookup algorithms that can be used in the security procedures to be executed in ZigBee and other IEEE 802.15.4-based Systems-on-Chip. The goal of the study is to find an effective approach to execute the security lookup procedures in resource-constrained embedded device suitable for application in wireless personal area network. Alternative lookup solutions are evaluated and compared by following criteria: speed/time complexity, memory needs, scalability, and hardware realisation efficiency. Lookup procedures functional analysis is performed, and security data characteristics are estimated. Three data structures are overviewed – trees, tries, and hash-tables. Their usage for keeping data for lookups is analysed are algorithms for lookup procedures implementation are proposed. Presented data structures are evaluated based on the mentioned criteria.

Keywords

IEEE 802.15.4 Security, Data Structures, Lookup Algorithms

1. Introduction

In ZigBee, and other IEEE 802.15.4-based Systems on Chip (SoCs) security procedures execution usually splits between programmable microprocessor or microcontroller and a specialized processor. While the last is used to execute most intensive security-related computations, i.e., AES-CCM* encryption and authentication, the first normally performs some not intensive but quite time-consuming legacy (or security) checks, which are executed in a form of lookup procedures. Information, related to these check executions, are placed in dedicated tables stored in a SoC memory and have to be correspondingly structured.

Security checks in the ZigBee and other IEEE 802.15.4-based SoCs have to be performed in a very short time, but also provide compactness and low power consumption, that is extremely important for resource-constrained devices operating in personal area networks (PANs). Since security procedures are based on keys and data stored in various tables, a very fast and effective lookup algorithm is necessary for ensuring that security checks do not have a significant negative impact on the performance and power consumption of the whole system. In this regard, the efficiency of lookup algorithms, over their own construction, depends also on the way data that they operate on is organized. Actually, the data structures used provide the first constraint on the best performance achievable by any algorithm using them. Therefore, prior to choosing a lookup algorithm, it is very important to select the most adequate data structure for the data to be acted upon.

The paper is structured as follows. We start in Sect. 2 with the overview of related works. In Sect. 3 we describe the criteria for the lookup algorithms that we used to evaluate their efficiency. In Sect. 4 we provide the analytical description of the IEEE 802.15.4 security lookup procedures. Security Data Characteristics and Contextual Information are presented in Sect. 5. Further, in Sect. 6, we

IntelITSIS'2021: 2nd International Workshop on Intelligent Information Technologies and Systems of Information Security, March 24–26, 2021, Khmelnytskyi, Ukraine
EMAIL: email1@mail.com (V. Melnyk)
ORCID: 0000-0002-1176-2831 (V. Melnyk)

analyse the keys used in the security lookup procedures in order to understand which of the mentioned data structures serves better our objectives. Moreover, we present trees, tries and hash tables – we do not discuss arrays and lists since they are basic structures easily outperformed by the former three ones – and describe the lookup algorithms that can be used with them. A comparison based on our criteria is given in Sect. 7 and conclusions are drawn at the end of the paper.

2. Related Work

Numerous scientific works are devoted to the implementation of IEEE 802.15.4 transceivers in general, as well as information security systems in them. In particular, in paper [1] the IEEE 802.15.4 software implementation is considered for typical sensor node systems, the available implementations concerning the supported features, flexibility issues and the efficiency of the implementations are discussed. In paper [2] the design and implementation of IEEE 802.15.4 software stack in Embedded Linux kernel is proposed. In papers [3, 4] hardware implementations of the IEEE 802.15.4 protocol in FPGA are proposed, whereas in [5, 6] – implementations of information security subsystem for this standard and of AES-CCM* cryptographic engine. However, to the best of the authors knowledge, in the literature there is no available information that would relate to the study of efficient implementation, hardware or software, of data retrieval operations in the IEEE 802.15.4 security procedures, but which, according to the author's opinion, based on his practical experience in developing security means for IEEE 802.15.4 systems, is quite important.

3. Efficiency Criteria

Below the criteria used for evaluating and comparing alternative lookup solutions are briefly described.

Speed/Time complexity

Speed is the most important metric in the evaluation of the algorithms. Since prior to the implementation we can only have estimates about the speed, we usually use time complexity estimates available from the literature. These estimates are obtained theoretically or empirically, but usually in both ways. This metric cannot provide an absolute duration estimate for a lookup algorithm. It can, however, be used to estimate and compare the efficiency of the algorithms of interest prior to their implementation in universal or specialized processors [7].

Memory needs

Given that the algorithms will run on small, embedded devices, the usable memory is also very limited. The lookup algorithms will use indices or arrange data in a particular way to speed up data access. Therefore, here we aim at minimizing the memory needed to store these indices and rearranged data. In our estimations we assume usage of conventional Random Access Memory (RAM), however, application of alternative memory types are promising in this regard, like the one proposed in [8].

Scalability

The time that can be allocated to lookups is extremely short; therefore, it is important to know how the raising number of entries affects the performance of the algorithms.

Hardware realization

Hardware implementation of the algorithms in question may also be required. Therefore, it is important to know how appropriate each of these alternatives is for this purpose as well.

4. IEEE 802.15.4 Security Lookup Procedures

Although the lookup algorithms and data structures we present in this paper have not been explicitly designed for security data, in order to select the most appropriate ones and to optimize their performance we need to take into account the data's characteristics. Thus, we need to know the type and size of the data used as a key for the lookups, as well as the number of keys (i.e., records) involved in the lookups at the same time. Furthermore, some knowledge about the hardware architecture, like the availability of a cash memory beside the RAM and the size of these memories, is also necessary. While key type and size information are defined in the standard [9] with a good precision, hardware specification may differ from one SoC to another. Therefore, in this latter case we use some assumptions; eventual corrections and adaptations should be made for certain SoC architecture.

There are four security lookup procedures [10] that need to be implemented in the IEEE 802.15.4-based SoC. These are the *key descriptor lookup, incoming security level checking, blacklist checking* and *incoming key usage policy checking* procedures.

4.1. Lookup Procedures Functional Analysis

Each device in the PAN can communicate with one or more another devices. Depending on which keying model [11] is used in the PAN, one or more encryption keys are used by device for communication. The information determining how to provide the security is found in the security-related PIB (PAN Information Base), which contains PIB security material. All keys are contained into KeyDescriptor entries of the *macKeyTable* of the MAC PIB. Besides the key, KeyDescriptor contains key-related information that is used during frame securing/unsecuring process, and auxiliary information that is used for KeyDescriptor identification.

Selection of appropriate key and key-related information performs KeyDescriptor lookup procedure. It compares the pair of input values KeyLookupData and KeyLookupSize with contained into the KeyDescriptor pair of values. Their coincidence means that communicating pair of devices uses exactly this KeyDescriptor (i.e. exactly this key and key-related information). The KeyDescriptor lookup procedure is applied either during securing and unsecuring of the frame. The number of KeyDescriptor entries that *macKeyTable* contains is equal to the number of keys that this device uses. One key may be used by device to communicate with one or more remote devices within the PAN.

During input frame unsecuring device has to determine whether the sending remote device belongs to the devices that use identified key. For this reason, each KeyDescriptor contains the list of devices, for which this key is used. It is contained into the KeyDeviceList entry of the KeyDescriptor. Verification of the device in the KeyDeviceList entry performs blacklist checking procedure (with embedded DeviceDescriptor lookup procedure). Also, during input frame unsecuring, device has to determine whether identified key may be used to unsecure the frame of present type. The frame types which unsecuring with identified key is supported are determined in the KeyUsageList entry of the KeyDescriptor. Verification of the frame type in the KeyUsageList entry performs key usage policy checking procedure.

These three lookup procedures operate in the same MAC PIB security table: *macKeyTable*. During input frame unsecuring device has to check whether the frame to be unsecured corresponds to the minimum security requirements that are applied to this type of MAC frame. For this reason, MAC PIB contains *macSecurityLevelTable* with the set of SecurityLevelDescriptor entries. Each SecurityLevelDescriptor determines minimum security level used for appropriate frame type. If the frame is MAC Command frame then security minimum may differ depending on the command type. Verification of the security minimum requirements in *macSecurityLevelTable* performs incoming security level checking procedure. Each lookup procedure is described in following subsections.

4.2. KeyDescriptor Lookup Procedure

Following actions shall be performed in regard to the *macKeyTable* attribute. The *macKeyTable* attribute represents a table of key descriptor entries, each containing keys and related information required for secured communications. Each key descriptor (KeyDescriptor) entry comprises information which is shown in Table 1. The inputs of this procedure are KeyLookupDataValue and KeyLookupSizeValue. The output of this procedure are KDL_KeyDescriptor and KDL_STATUS values. In order to identify an appropriate KeyDescriptor the KeyIdLookupList element of KeyDescriptor is analyzed. Each KeyIdLookupList element contains a list of KeyIdLookupDescriptor entries used to identify current KeyDescriptor. Consequently, each KeyIdLookupDescriptor comprises information which is shown in Table 2.

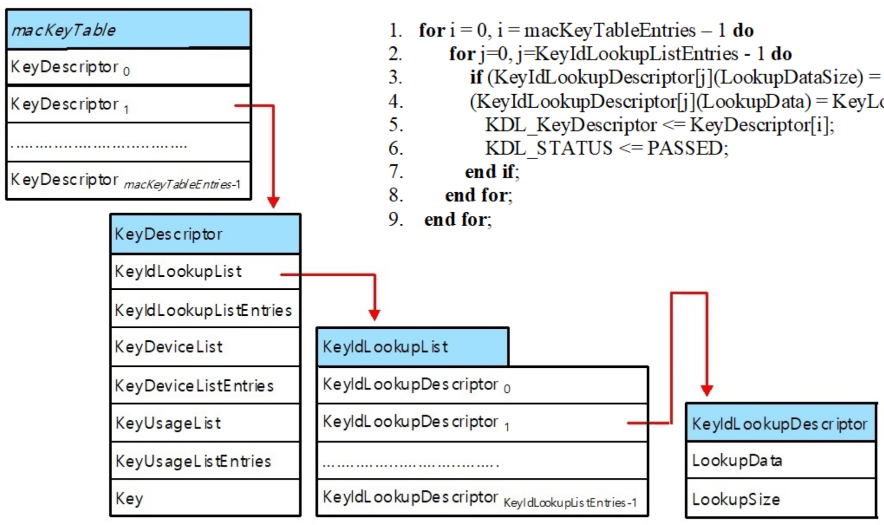
Table 1
Fields and estimated sizes of KeyDescriptor record

Field name	Type	Size (bits)	Description
KeyIdLookupList	List of KeyId-LookupDescriptor etries	8*8+8=72/entry or 4*8+8=40/entry	List of KeyIdLookupListEntries entries used to identify this KeyDescriptor, each containing a LookupData and a LookupSize fields. LookupData (64 or 32 bits) and LookupSize (8 bits) are used as the key for the outgoing frame key retrieval lookup procedure.
KeyIdLookupListEntries	Integer	8	The number of entries in KeyIdLookupList. Implementation specific. 8 bits are enough for 256 entries.
KeyDeviceList	List of Key-DeviceDescriptor entries	8+1+1=10/entry	List of KeyDeviceDescriptor entries indicating which devices are currently using this key, including their blacklist status (see Table 5), each containing a DeviceDescriptorHandle – an implementation specific integer – a UniqueDevice and a Blacklisted indicator, each of boolean type.
KeyDeviceListEntries	Integer	8	The number of entries in KeyDeviceList. Implementation specific.
KeyUsageList	List of KeyUsageDescriptor entries	8+8=16/entry	A list of KeyUsageDescriptor entries indicating frame types this key may be used with, each containing a FrameType and a CommandFrameIdentifier.
KeyUsageListEntries	Integer	8	The number of entries in KeyUsageList.
Key	Set of 16 octets	16*8=128	The actual value of the key.

Table 2
Fields of KeyIdLookupDescriptor entry

Name	Type	Description
LookupData	Set of 5 or 9 octets	Data used to identify the key.
LookupDataSize	Integer	A value of 0x00 indicates a set of 5 octets; a value of 0x01 indicates a set of 9 octets.

A lookup process involving the *macKeyTable*, KeyDescriptor, KeyIdLookupList, and KeyId-LookupDescriptor is illustrated in Figure 1. The algorithm of KeyDescriptor determination is shown by a pseudo code; it implies comparison of input values KeyLookupDataValue and KeyLookupSizeValue with LookupData and LookupSize elements of each KeyIdLookupDescriptor, respectively.



```
1.  for i = 0, i = macKeyTableEntries – 1 do
2.      for j=0, j=KeyIdLookupListEntries - 1 do
3.          if (KeyIdLookupDescriptor[j](LookupDataSize) = KeyLookupSizeValue and
4.          (KeyIdLookupDescriptor[j](LookupData) = KeyLookupDataValue) then
5.              KDL_KeyDescriptor <= KeyDescriptor[i];
6.              KDL_STATUS <= PASSED;
7.          end if;
8.      end for;
9.  end for;
```

Figure 1: KeyDescriptor lookup process

4.3. Incoming Security Level Checking Procedure

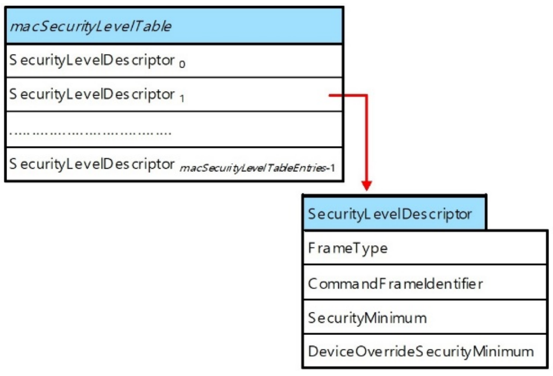
Following actions shall be performed in regard to the *macSecurityLevelTable* attribute of the MAC PIB. This attribute contains a table of SecurityLevelDescriptor entries, each with information about the minimum security level expected depending on incoming frame type and subtype. The elements of the SecurityLevelDescriptor are shown in Table 3.

Table 3
Fields and estimated sizes of SecurityLevelDescriptor record

Field name	Type	Range	Size (bits)	Description
FrameType	Integer	0x00–0x03	3	The Frame Type subfield is 3 bits in length and shall be set to one of the nonreserved values as follows (values 100 – 111 are reserved): “000” – beacon, “001” – data, “010” – acknowledgement, “011” - MAC command. Used as a key in the incoming security level checking procedure together with CommandFrameIdentifier and SecurityMinimum.
CommandFrameIdentifier	Integer	0x00–0x09	8	The CommandFrameIdentifier subfield encodes the command name (defined in the standard [9]).
SecurityMinimum	Integer	0x00–0x07	8	The minimal required/expected security level for incoming MAC frames with the indicated frame type and, if present, command frame type.
DeviceOverrideSecurity-Minimum	Boolean	TRUE or FALSE	1	Indication of whether originating devices, for which the Exempt element (belongs to elements of DeviceDescriptor) is set, may override the minimum security level indicated by the SecurityMinimum element. If TRUE, this indicates that for originating devices with Exempt status, the incoming security level zero is acceptable, in addition to the incoming security levels meeting the minimum expected security level indicated by the SecurityMinimum element.

The inputs of this procedure are SecurityLevelSubfield, FrameTypeSubfield, and, depending on whether the frame is MAC command frame, the first octet of the MAC payload (i.e. Command Frame Identifier subfield). The output of this procedure is status (ISLC_STATUS), which can take PASSED, CONDITIONALLY_PASSED, or FAILED value.

A lookup process in the *macSecurityLevelTable* is shown in Figure 2. It operates as it is shown by a pseudo code. The elements of SecurityLevelDescriptor are enclosed into brackets, like SecurityLevelDescriptor(FrameType) is a FrameType element of SecurityLevelDescriptor of *macSecurityLevelTable* attribute.


macSecurityLevelTable
SecurityLevelDescriptor 0
SecurityLevelDescriptor 1
……………………………
SecurityLevelDescriptor macSecurityLevelTableEntries-1
SecurityLevelDescriptor
FrameType
CommandFrameIdentifier
SecurityMinimum
DeviceOverrideSecurityMinimum

```
for i = 0, i = macSecurityLevelTableEntries -1 do
  if (FrameTypeSubfield /= “011” and
    FrameTypeSubfield = SecurityLevelDescriptor[i](FrameType)) then
    execute sec_level_compare(SecurityLevelValue, SecurityLevelDescriptor[i](SecurityMinimum));
    if sec_level_compare <= FAILED then
      if (SecurityLevelDescriptor[i](DeviceOverrideSecurityMinimum) = true
        and SecurityLevelSubfield = “000”) then
         ISLC_STATUS <= CONDITIONALLY_PASSED;
      else ISLC_STATUS <= FAILED;
      end if:
    else ISLC_STATUS <= PASSED;
  else if (FrameTypeSubfield = “011” and FrameTypeSubfield = SecurityLevelDescriptor[i](FrameType)
        and CommandFrameIdentifierSubfield = SecurityLevelDescriptor[i](CommandFrameIdentifier)) then
    execute sec_level_compare(SecurityLevelValue, SecurityLevelDescriptor[i](SecurityMinimum));
    if sec_level_compare <= FAILED then
      if (SecurityLevelDescriptor[i](DeviceOverrideSecurityMinimum) = true
        and SecurityLevelSubfield = “000”) then
         ISLC_STATUS <= CONDITIONALLY_PASSED;
      else ISLC_STATUS <= FAILED;
      end if:
    else ISLC_STATUS <= PASSED;
  end if;
end for;
```

Figure 2: SecurityLevelDescriptor lookup process

4.4. Blacklist Checking Procedure

Following actions shall be performed in regard to the KeyDescriptorValue, which comprises information as it is shown in Table 1. The procedure inputs are KeyDescriptorValue, DeviceLookupDataValue, and DeviceLookupSizeValue. The procedure outputs are BLC_DeviceDescriptor, BLC_KeyDeviceDescriptor, and BLC_STATUS values.

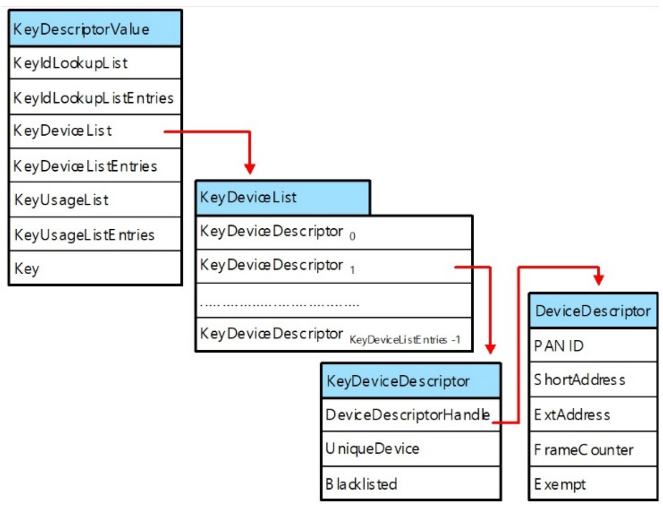
In order to identify an appropriate DeviceDescriptor and KeyDeviceDescriptor the KeyDeviceList element of KeyDescriptorValue is analyzed. Each KeyDeviceList element contains a list of KeyDeviceDescriptor entries indicating which devices are currently using this key, including their blacklist status. Each KeyDeviceDescriptor contains a DeviceDescriptorHandle, which identifies a DeviceDescriptor corresponding to the current device. The DeviceDescriptor comprises information which is shown in Table 4.

Table 4

Fields and estimated sizes of DeviceDescriptor record

Field name	Type	Size (bits)	Description
PAN ID	Device PAN ID	16	PAN identifier of the device in this DeviceDescriptor. May be used as a key in the blacklist checking procedure together with ShortAddress.
ShortAddress	Device short address	16	Short address of the device in this DeviceDescriptor. A value of 0xFFFE indicates that this device is using only its extended address. A value of 0xFFFF indicates that this value is unknown. May be used as a key in the blacklist checking procedure together with PAN ID.
ExtAddress	IEEE address	64	IEEE extended address of the device in this DeviceDescriptor. This element is also used in unsecuring operations on incoming frames. May be used as a key in the blacklist checking procedure.
FrameCounter	Integer	32	The incoming frame counter of the device in this DeviceDescriptor. This value is used to ensure sequential freshness of frames.
Exempt	Boolean	1	Indication of whether the device may override the minimum security level settings defined in Table 3.

A lookup process involving the KeyDescriptorValue, KeyDeviceList, and KeyDeviceDescriptor is illustrated in Figure 3. The pseudo code aside shows the DeviceDescriptor and the KeyDeviceDescriptor lookup process.



```
for i = 0, i = KeyDeviceListEntries – 1, do
   BLC_DeviceDescriptorValue <=
      KeyDeviceDescriptor[i](DeviceDescriptorHandle(DeviceDescriptor));
   if KeyDeviceDescriptor[i](Blacklisted) = false then
      if KeyDeviceDescriptor[i](UniqueElement) = true then
         BLC_KeyDeviceDescriptor <= KeyDeviceDescriptor[i];
         BLC_DeviceDescriptor <= DeviceDescriptorValue;
         BLC_STATUS <= PASSED;
      else if KeyDeviceDescriptor[i](UniqueElement) = false then
 -- DeviceDescriptor lookup
         if ((DeviceLookupSizeValue = 4 and
             DeviceLookupDataValue =
                   BLC_DeviceDescriptorValue(PANID)&&
                   BLC_DeviceDescriptorValue(ShortAddress)) or
            (DeviceLookupSizeValue = 8 and
             DeviceLookupDataValue =
                   BLC_DeviceDescriptorValue(ExtAddress))) then
            BLC_KeyDeviceDescriptor <= KeyDeviceDescriptor[i];
            BLC_DeviceDescriptor <= DeviceDescriptorValue;
            BLC_STATUS <= PASSED;
         else BLC_STATUS <= FAILED;
         end if;
      end if;
   else BLC_STATUS <= FAILED;
   end if;
 end for;
```

Figure 3: DeviceDescriptor and KeyDeviceDescriptor lookup process

4.5. Incoming Key Usage Policy Checking Procedure

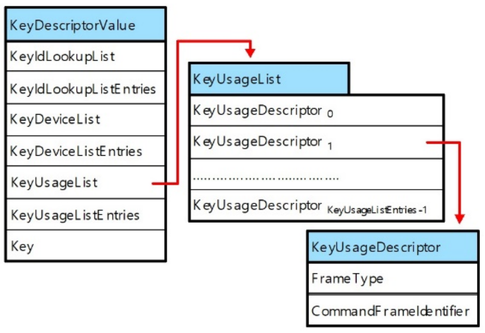
Following actions shall be performed in regard to input KeyDescriptorValue, which comprises information as it is shown in Table 1. The procedure inputs are KeyDescriptorValue, FrameTypeSubfield, and CommandFrameIdentifierSubfield values. The procedure output is IKPC_STATUS value.

In order to check incoming key usage policy a KeyUsageList element of KeyDescriptorValue is analyzed. Each KeyUsageList element contains a list of KeyUsageDescriptor entries indicating which frame types this key may be used with. Consequently, each KeyUsageDescriptor comprises information which is shown in Table 5.

Table 5
Fields and estimated sizes of KeyUsageDescriptor record

Field name	Type	Size (bits)	Description
FrameType	Integer	3	See Table 3. Used as a key in the incoming key usage policy checking procedure together with CommandFrameIdentifier.
CommandFrameIdentifier	Integer	8	8-bit value encoding MAC command name. Used as a key in the incoming key usage policy checking procedure together with FrameType.

A lookup process involving the KeyDescriptorValue, KeyUsageList, and KeyUsageDescriptor is illustrated in Figure 4. A pseudo code aside explains incoming key usage policy checking procedure.



```
for i = 0, i = KeyUsageListEntries – 1, do
  if (FrameTypeSubfield /= 0x03 and
    FrameTypeSubfield = KeyUsageDescriptor[i](FrameType)) then
    IKPC_STATUS <= PASSED;
  else if (FrameTypeSubfield = 0x03 and
    FrameTypeSubfield = KeyUsageDescriptor[i](FrameType)) and
    CommandFrameIdentifierSubfield =
        KeyUsageDescriptor[i](CommandFrameIdentifier)) then
    IKPC_STATUS <= PASSED;
   else IKPC_STATUS <= FAILED;
   end if;
```

Figure 4: KeyUsageDescriptor lookup process

5. Security Data Characteristics and Contextual Information

After looking up the standard [9], we understand that the security data used in the lookup procedures are as presented in Tables 1, 3, 6, and 7. The keys are all bit strings, the longest being an extended device address of 64 bits (8 bytes).

Let us summarize now minimum total amount of memory needed to implement the lookup procedures.

- The minimum total amount of bits needed to store one key descriptor record (given in Table 1), when in each of its lists there is only 1 entry, is 72+8+10+8+16+8+128=250.
- The total amount of bits to store one SecurityLevelDescriptor record (see Table 3) is 20.
- The total amount of bits needed to store one DeviceDescriptor record (see Table 4) is 137.
- The total amount of bits needed to store one KeyUsageDescriptor record (see Table 5) is 11.

As can be seen from the fields enumerated in Table 1, there are various lists involved in the lookup procedures. The number of entries in these lists adds to the memory occupation of the SoC, which is quite limited. Let us assume the size of the SoC's memory (i.e., RAM) is 4 KB. In order to estimate the number of keys that can be present in the security information database of each device (i.e., *macKeyTable*) we make some simple computations for which we assume that one quarter of the RAM (i.e., 1KB =1024 bits) is used on this purpose. Assuming also that the lists in the KeyDescriptor (Table 1) contain only 1 entry each, the amount of memory used for 1 key in all of the lookup procedures is at least 250+20+137+11=418 bits=52.25 bytes – this results from adding up the total number of bits in Tables 1, 3, 6, and 7. Thus, about 19 such keys can be fitted into 1 KB of memory at the same time, without counting the memory used by the index data structure (if one is used). This is a very optimistic calculus; therefore, it may turn out that not even 19 key descriptors can be fitted in 1KB of memory. On the contrary, it may be proved that in practice there is no need to store 19 keys at the same time. Anyway it be, for this research we reduce this number a little bit and fix it to be 16, just to leave some space for the indexing data structure.

Depending on the model of microcontroller unit (MCU) used, a cache memory may or may be not available during the lookup procedures and the execution time of lookup procedures depends heavily on this. Storing the indexing data structure in a fast-access cache memory constitutes a big advantage, because the lookups can be performed with fast cash accesses almost exclusively. Only one access is necessary to the slower RAM to retrieve the data record from the position obtained from the index. If the MCU without a cache memory is considered (e.g., Intel 8051 MCU), each operation on the indexing data structure will consist in a slow memory access.

The above considerations regarding the data that the lookup algorithms are to operate on as well as the mentioned hardware issues should be taken into account in the selection of the data structure used, in order to improve performance.

6. Data Structures and Algorithms

In this section the three main data structures – trees, tries and hash tables – that we consider for the security data indexing, are presented. Moreover, for each data structure we present the necessary algorithms for performing operations on them. These algorithms can be used to insert or delete a node into/from the data structure and to find a desired node in it.

6.1. Trees

A tree is a data structure consisting of nodes, containing data, linked in a particular way. There is a *root* node at the origin or top of the tree structure and it has other nodes, called *child* nodes, linked to it. On turn, each child node has its own children and so on. This way the tree structure is constructed on many *levels* (a level meaning the nodes at the same number of links far from the root node). Each node has a *subtree* underneath, except if it has no children at all. Such nodes are called *leaf* nodes, while all the other nodes are called *internal* nodes. In many cases the useful information is stored in the leaf nodes and the internal nodes only serve for structuring information.

One of the most popular variants of trees is *binary trees*. Binary trees comply with the additional rule that their nodes can have 0, 1 or 2 children only. This characteristic implies that when *traversing* the binary tree (i.e., inspecting/visiting its nodes one-by-one) a binary decision (i.e., 'yes' or 'no'; 'true' or 'false'; '0' or '1'; 'bigger' or 'smaller', etc) has to be made based on some criterion. Such property renders binary trees very adequate for indexing and search activities, since every decision made as going down the levels of the tree structure reduces the possible locations of the searched key to the subtree below the current node. Often, when the tree is used on lookup purposes, data is organized so that all data in the left subtree of the node precedes any data in the right subtree of the same node, again, based on some criterion. Such ordered trees are called *binary search trees* (*BST*). Figure 5 shows an example where the ordered list of numbers 2, 8, 10, 24, 56, 73, 74, 79, 84, 99 is structured into a BST.

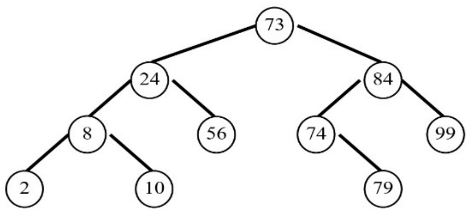


Figure 5: Binary search tree

In Figure 6 we present the pseudo codes of the *Insert*(), *Lookup*() and *Delete*() algorithms that can be used with BSTs. In these algorithms each node is assumed to have a *key* field, containing a pointer (e.g., of 2 bytes, depending on hardware) to the data that uniquely identifies a record among all possible security information data that can be stored in the BST. Moreover, each node has a *left* and *right* field containing pointers to the children of the node. The *currentNode* variable is used to indicate the node under inspection at each time. It represents the point of access to the data structure. *newNode* is the node containing the *key* to insert into the tree. The *ValueOf*(*pointer*) function returns the data pointed by the pointer passed to it as an argument. *Remove*(*pointer*) releases the memory occupied by a node pointed by the function's argument. Next we describe the operation of each of these algorithms (see Figure 6).

The *Insert*() function takes as an argument the node to insert into a tree and a *currentNode* pointer initialized with the root node. The algorithm recursively compares the key to insert with the key pointed by the *currentNode*. If the new key is smaller than the current key (line 4), steps to the left child of *currentNode* (line 5), otherwise continues with the right child (line 7). The insertion point is found when *currentNode* assumes a *null* value (line 2). This is where *newNode* is added to the tree.

The *Lookup*() function operates similarly to the *Insert*() one. It takes as arguments the key to search and the *currentNode* pointer initialized to the root. The function compares recursively the keys pointed by *currentNode*.*key* and if the searched *key* is smaller, the lookup continues at the *left* child of *currentNode* (line 13), otherwise at the *right* child (line 15). When a match is found then the pointer to the security information belonging to *key* is returned. If it happens that *currentNode* takes up the *null* value, it means that the key has not been found in the BST.

Initialization: Each BST node has a *key* field and a left and right child. *key* is a pointer to the record of security information. *newNode* is initialized with the key to insert and its left and right pointers are **null**. *currentNode* is initialized with the root.

```
1. Insert(newNode, currentNode) // Inserts a key into the tree structure
2.      if (currentNode == null) then
3.              currentNode ← newNode;
4.      elseif (ValueOf(currentNode.key) > ValueOf(newNode.key)) then
5.              Insert(newNode, currentNode.left);
6.      else
7.              Insert(newNode, currentNode.right);
8.      endif;
9. end Insert();

10. Lookup(key, currentNode) // Searches a key into the tree structure
11.     if  (currentNode == null) then return "Key not found!";
12.     if (ValueOf(currentNode.key) > key) then
13.             return Lookup(key, currentNode.left);
14.     elseif (ValueOf(currentNode.key) < key) then
15.             return Lookup(key, currentNode.right);
16.     else // keys are equal, key found
17.             return currentNode.key; // returns the pointer to the desired record
18.     endif;
19. end Lookup();

20. Delete(key, currentNode) // Deletes key from the tree
21.     find the node pertaining to key and assign it to currentNode. Now ValueOf(currentNode.key)==key;
22.     if (currentNode.left== null and currentNode.right==null) then
23.             Remove(currentNode);
24.     elseif (currentNode.left== null) then
25.             tmp ← currentNode;
26.             currentNode ← currentNode.right;
27.             Remove(tmp);
28.     elseif (currentNode.right==null) then
29.             tmp ← currentNode;
30.             currentNode ← currentNode.left;
31.             Remove(tmp);
32.     else
33.             tmp←currentNode.left;
34.             while (tmp.right != null) do tmp←tmp.right; endwhile;
```

```
35.            currentNode.key ← tmp.key;
36.            Remove(tmp);
37.    endif;
38. end Delete();
```

Figure 6: Algorithms for the binary search tree approach

Finally, the *Delete*() function removes from the tree the node pointing the security information belonging to its *key* argument. First the location of the node to remove is looked up and then, based on the node's links to its parent and child nodes, the appropriate operations are performed to exclude the node from the tree. There can be four different cases. First, both children are *null* (line 22); second and third, when one of the children is *null* (line 24 and 28); and finally, when none of the nodes is *null* (32). In this latter case, the *left* child will take the eliminated nodes place in the BST.

6.2. Tries

A trie is a tree-shaped data structure in which keys are stored by the position of the nodes in the tree, instead of inside the nodes themselves as in the case of binary trees. If in a trie nodes can have at most 2 children for storing bits then the trie is called a binary *digital search tree* (*BDS-tree*). Thus, BDS-trees are appropriate for structuring bit strings. The bits are stored one-by-one on the links and not inside the nodes. Each bit string is encoded as a series of '0' and '1' links between level 0 of the trie (i.e., the root node) and a leaf node. Thus, each subtree in the trie has the same prefix. For example, in Figure 7a all keys in the right branch have the prefix 1. Therefore, the keys encoded in the right branch are 10001, 1011, and 11. If instead we wanted all of the keys with prefix 000, we would have 00000, 00001 and 0001. This structure is very efficient when it comes to looking up bit string keys. While looking up a key in a BST takes *O(m* n)* time (*n* being the number of levels in the tree and *m* the length (i.e., the number of bits) of the key), the same operation with a trie is just *O(m)* in the worst case. An additional advantage of tries is that comparisons are made on bits, instead of strings, which makes operations cheaper to perform. This significantly increases the search speeds. Tries have lower memory needs as well, since they share prefixes. Trie algorithms, though, can be somewhat more complex than BST ones.

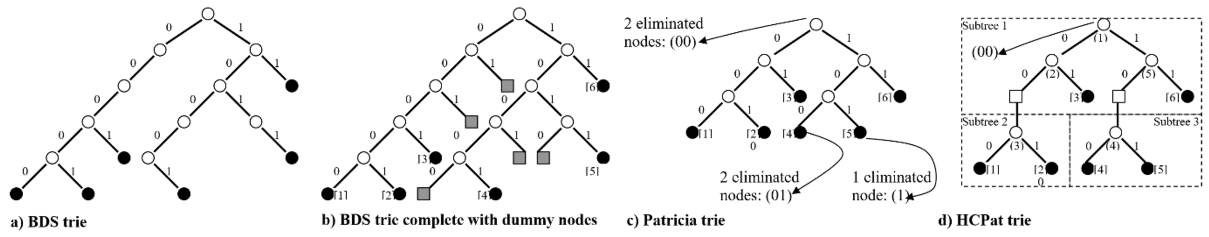


Figure 7: Trie types

It can be noted in Figure 7a that some of the nodes of the trie have one single child only. If there are too many single-child nodes present, the trie can become very deep, which causes performance loss in information retrieval systems [12]. There are two ways to avoid the formation of very deep tries. The first method consists of collecting keys with the same prefix into a group, called *bucket*. When a key has to be looked up, the leaf node with the corresponding prefix is identified and the presence of the searched key is checked in the bucket pointed by the prefix leaf node. If the prefix leaf node is missing, it means that the key has not been found. Note that by using buckets all subtrees that are built of all single-child nodes can be compressed into one node that points to the bucket containing the key. In other words, this way there will be no parent of a leaf node that has one child only.

The second method for reducing the depth of tries is to eliminate the nodes that have only one child. These tries are called *Patricia tries* [13] and discussed in more detail later in this section.

Implementing tries with pointers is inefficient, since the pointers used to store each node in a tree would consume much more memory than the bits of the key themselves. Thus, Jonge et al [14] proposed compressing the binary trie into a compact bit stream, called CB tree.

6.2.1. CB Tree

The basic idea behind CB trees is that the nodes of the trie are organized in a so-called tree map in the form of a bit stream corresponding to the pre-order traversal of the trie. This means that the trie is traversed inspecting the current node first followed by the left and finally the right child, emitting ‘0’ for an internal node and ‘1’ for a leaf. In order for this representation method to be functional, it is necessary for the trie to be *full*, meaning that each internal node must have exactly 2 children. In any full (sub)tree the number of leafs is one more than the number of internal nodes and this property is indispensable for the algorithms that operate on such a tree representation. On this purpose those nodes that have only one child are completed with an extra child, which contains no information. Such empty nodes are called *dummy* nodes. For keeping track of the dummy nodes, a second bit stream is employed, called *leaf map*. In the leaf map for each dummy node a ‘0’ bit is emitted while a ’1’ is used otherwise, again, in pre-order. Finally, the number of ‘1’ bits in the leaf node is used to match the key to a bucket using a bucket table, BTBL. The bucket, on turn, should contain a pointer to the data record (e.g., security information) belonging to the key. In Figure 8a an example is given for transforming the trie in Figure 7b into a CBtrie representation. In Figure 7 the bucket numbers are marked in square brackets beside the non-dummy leaf nodes. The procedure of matching bucket addresses to table positions can be perceived as a *hashing* operation (see section about hashing later in this paper). Indeed, this operation is known as *trie hashing* [15]. Although adding dummy nodes is very important for this compact trie representation, handling such nodes still adds an unwanted burden on the algorithms operating on CB (Compact Binary) digital search tries. This is why in [12] a method that eliminates dummy nodes is proposed. The proposed method is based on Patricia tries.

6.2.2. Patricia Tries

Patricia (Practical Algorithm to Retrieve Information Coded in Alphanumeric) tries [13] were defined to eliminate one-child nodes from tries, so that to reduce their spread and depth. In order to keep track of the eliminated links, their values are stored in the remaining nodes. For example, the trie in Figure 7b can be converted into the Patricia trie in Figure 7c, where a total of 5 nodes have been eliminated from the tree structure. In order to avoid false matches in a search operation, the eliminated links are memorized at their parent nodes. Therefore, during search operation these eliminated nodes have to be taken into consideration.

Patricia tries are very efficient in indexing a small number of long keys, or keys with long shared prefixes and the efficiency of key lookups does not depend on the number of keys, but on their length. Also, since they do not contain one-child nodes or dummy nodes, they have been used to reduce the size of CB tries [12]. CB tries based on Patricia tries are called CPat (Compact Patricia) tries.

6.2.3. CPat Tries

CPat tries combine the compactness of Patricia and CB tries in two different planes. While Patricia tries save memory by reducing the number of nodes in the tree structure, CPat tries take advantage of the array-based representation, which uses a bit stream instead of pointers, for structuring keys. Similar to CB tries, CPat tries also use two bit streams. The tree map is the same as for CB trees, but leaf map is useless in this case because there are no dummy nodes in CPat tries. A node map is used instead, which accounts for the eliminated nodes. Thus, in the node map for each node that contains no eliminated nodes a ‘0’ is emitted. On the other hand, for each eliminated node contained in a node a ‘1’ is emitted and the string of ‘1’s is terminated by a ‘0’. In the example of Figure 8b the Patricia trie from Figure 7c is represented in a compressed format. Note that in the node map only the internal nodes are represented. There is no need to include here the leaf nodes as well, since even nodes containing eliminated nodes match to the same bucket.

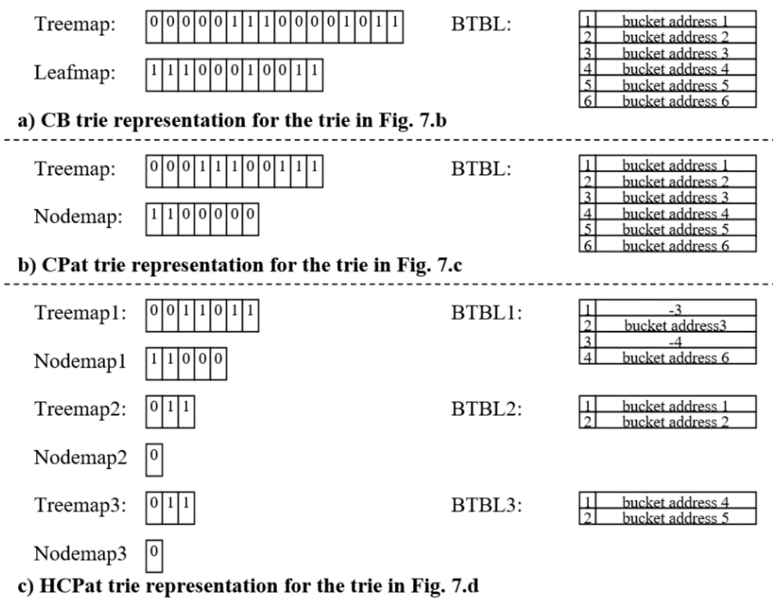


Figure 8: Compact trie representation

This combined representation results in a very reduced memory requirement for CPat tries. However, in cases when the key is situated at the right-bottom side of a Patricia trie, lookup times are somewhat increased because most of the bits in the bit stream representation are to be inspected. Furthermore, in case of insert and delete operations a large number of bits have to be moved. This has a negative impact on the speed of algorithms operating on CPat tries. To overcome this problem, a hierarchical approach, called HCPat (Hierarchical Compact Patricia) tries, has been proposed [16].

6.2.4. HCPat Tries

HCPat tries decompose the CPat trie in many smaller tries and link them together with pointers. In a HCPat trie the depth (i.e., the number of levels) of the tree is fixed and if the data can not fit into the desired number of levels, a new subtrie is created and the root node of the new subtrie is linked to one of the leaves of the old subtrie. This way each subtrie has its own treemap. Therefore, similar to a tree, as the search for a key progresses, some of the subtries are excluded from the search, reducing this way the number of inspected bits. As an example, the HCPat trie composed of 3 subtries in Figure 7d is compressed in Figure 8c. In Figure 7d bucket numbers are marked in square brackets while internal nodes are numbered in round brackets. Negative numbers in the BTBLs (Figure 8c) are used as pointers to the node with the corresponding internal node number (i.e., the absolute value of the pointer). These pointers link a leaf node of one subtrie to the root of another.

HCPat tries provide a good compromise between the performance of Patricia and CPat tries. While their memory requirements are about a third of Patricia tries, they provide a speed of about 40 times faster than CPat tries [16]. This performance renders HCPat tries very attractive for lookup of keys in large key sets.

Each of the presented trie types provide some improvement on the performance of the basic BDS-tree. However, with the performance, the computational complexity of these data structures increases as well. In this paper we aim at finding a data structure that can guarantee high speed with lookups with a relatively small number of keys and on devices with reduced computation resources. Therefore, while HCPat tries can provide a great compromise on speed and memory needs for many tens of thousands of keys, for our application with several tens of keys it may turn out that the additional computational complexity implied by this data structure outweigh the benefits. Furthermore, our application can not benefit from the strong sequential key retrieval property of tries either, since our keys will be retrieved one-by-one. Thus, even though the various types of tries can be very effective in certain application domains, it should be noted that security key lookups for ZigBee and other

IEEE 802.15.4-based SoCs do not really share the assumptions that tries have been designed for. Therefore, the usage of tries for these lookups is not very promising.

In Figure 9 we present high level pseudo codes of the *Insert*(), *Lookup*() and *Delete*() functions for the CPat trie. For these algorithms we use the *nodepos*, *treepos* and *keypos* counter variables to indicate the position of the current inspection in the *nodemap*, *treemap* and key bit strings, respectively. Also, MAX_BUCKET_SIZE is a constant indicating the maximum number of keys that a bucket can hold at a time.

In the *Insert*() function first the position where the new key from the function's argument has to be inserted is searched for. On this purpose the logic described in the *Lookup*() function can be used. When the bucket is identified, there are two cases that can show up. Either the bucket is partially full – in which case the key is simply added to it (line 4) – or it is completely full. In this latter case the bucket has to be split in two new buckets and the contents of the old bucket must be distributed among the two new ones (line 7). The new key has to be then added to the bucket with the corresponding address and all the buckets in the trie renumbered (line 8), such that to maintain the correspondence between the bucket numbers and the number of '1' bits in the node map.

```
Initialization: nodepos $\leftarrow$1; treepos $\leftarrow$ 1; keypos $\leftarrow$1; indices in nodemap, treemap and key;
1. Insert(key) // Inserts a key into a CPat trie;
2.     find insertion point (as in Lookup() );
3.     if (required bucket partially filled) then
4.             add key to bucket;
5.     else // bucket full
6.             split: convert bucket into an internal node with 2 buckets;
7.             distribute the keys, including key, among the 2 new buckets;
8.             renumber all buckets;
9.     endif;
10. Insert(key)//

11. Lookup(key)// Finds a key in a CPat trie;
12.    repeat
13.    if (key[keypos] == 1) then
14.    Skipping left subtree in treemap: advance treepos until nr 1 bits in treemap is one more than nr 0 bits;
15.    Skipping left subtree in nodemap: advance nodepos until nr 0 bits in nodemap passed the same times as the nr of 0 bits skipped in treemap;
16.    else
17.                    treepos++;
18.                    if (treemap[treepos] == 0) then
19.                            keypos++; nodepos++;
20.                            if (nodemap[nodepos] == 1) then
21.                            Advance keypos and nodepos until nodemap[nodepos] == 0
22.                            endif;
23.                    else
24.    count nr 1 bits in treemap from position 1 to treepos and obtain bucket address;
25.    if the key is found in the bucket, return 'Success, return pointer!', otherwise return 'Key not found! Return null.';
26.                    endif;
27.            endif;
28.    until treemap[treepos] != 1;
29. end Lookup();

30. Delete(key) // deletes a key from the trie
31.    find bucket containing key to delete (as in Lookup() );
32.    remove pointer from bucket;
33.    if (nr keys in bucket < MAX_BUCKET_SIZE/2 and nr keys in sibling bucket < MAX_BUCKET_SIZE/2) then
34.            merge bucket with its sibling;
35.    endif;
36. end Delete();
```

Figure 9: Algorithms for the Patricia trie based approach

The *Lookup*() function cycles through the bits of the *key* passed to it as argument until a leaf node is reached in the trie. If the currently inspected bit of the key is '1', the *treepos* and *nodepos* variables are incremented so that to point to the right subtrie. In *treemap* this can be achieved by passing one more ‘1’ bits than ‘0’ bits, since the number of leaf nodes in a full subtrie are one more than internal nodes and this is encoded into the *treemap* by its creation based on the pre-order traversal of the trie (line 14). In the nodemap the number of ‘0’ bits indicate nodes effectively represented in the trie. Therefore, the number of ‘0’ bits passed should be equal to the number of ‘0’ bits passed in the *treemap* (line15). However, if the inspected key bit is ‘1’ (line 16), the next bit in the *treemap* is evaluated. If the *treemap* bit ‘0’, *keypos* and *nodepos* are advanced, accounting for eliminated nodes as well (lines 19-22) and the search continues with a subsequent cycle. If, however, the *treemap* bit is ‘1’ (line 13) then the search arrived to a leaf node and the corresponding bucket number is retrieved by counting the passed ‘1’ bits (leaf nodes) in the *treemap*. The bucket address is obtained from the BTBL (line 24) and the contents examined. If the searched key is found in the bucket then a pointer is returned to the data of interest, otherwise the key is not in the trie and the search is unsuccessful (line 25).

The *Delete*() function removes the *key* passed to it as an argument from the trie. The function starts with identifying the bucket that should contain the *key* (line 31) and removes it. The deletion operation could end at this point, however, this would lead to the formation of buckets that contain too few keys compared to their maximum capacity, MAX_BUCKET_SIZE. This could increase the depth and spread of the trie uselessly. To avoid having almost empty buckets in the trie after the deletion, the number of keys in the bucket from which a key has just been removed and its sibling bucket (i.e., the bucket sharing the same parent node) are counted. If both of these buckets are less than half full (line 33), they are merged (line 34). Merging involves copying the contents of the less full bucket into its sibling bucket and replacing their parent node with this latter. The two old buckets are then removed from the trie. This way an internal node and two buckets are compressed into one single bucket.

Although the above functions were designed for CPat tries, they can be easily adapted for CB or HCPat tries as well, using as reference [12, 14, 16].

6.3. Hash Tables

In many applications the data used to identify a particular record (i.e., the key) can have an extremely wide range of values, even though only a small subset is ever used together. In such situations a function can be defined, that converts the keys so that to fit into an acceptably sized range. This operation produces a table that matches some value from the narrow range to each useful key value in the wide range, resulting in pairs of narrow- and wide-range keys. We can look at this table as an array indexed by the narrow-range key and valued with the wide-range keys, or even valued with pointers to the record that contains the wide-range key. Such tables and arrays are known as hash tables and the function that performs the wide-to-narrow range translation is called a *hash function*. The verb ”*to hash*” is used for the translation operation.

Hash tables provide a very fast way for looking up data. As a long key is fed into the hash function, the corresponding hash table index is obtained and at the same time the searched record (or pointer to its location in the memory) is instantaneously found. Therefore, each lookup operation takes only the evaluation of the hash function. It is an additional value of this data structure that it can maintain this performance even if the number of keys grows very high. Thus, its scalability is unparalleled.

Although the lookup velocity enabled by hash tables is hard to overtake, this data structure can provide such a high performance only in favourable conditions. This is not the case when the number of keys that have to be stored in the hash table is unpredictable. First, since hash tables inherit all of the arrays’ properties, after their dimension is fixed (i.e., the memory had been allocated) it is costly to re-dimension them. Second, if the array is over-dimensioned to let space for growth and fluctuation, very often the memory is used up uselessly. Third, if keys have to be looked up sequentially, extra time-consuming operations have to be performed in order to ensure that hashed and original keys are ordered in the same way. Finally, given that by hashing a wide range of keys is squeezed into a much smaller range, it can often happen that two different long keys are hashed into the same short key. Such collisions can be resolved at a cost. Usually, hash tables full at 70% perform well, but as their *load factor* (i.e., the ratio of the used and total number of hash table positions) grows beyond that point (i.e., 0.07), the performance starts degrading quickly.

There are two widely used methods for resolving collisions: *separate chaining* and *open addressing*. With separate chaining at each position of the hash table, where a collision occurs, a linked list is created and each key that hashes to that position is appended to the list. At lookup this list is searched element-by-element. On the other hand, with open addressing a collision is resolved by searching an empty position for the second key, according to some rule. One of the simplest such rules is to assign the upper adjacent index in the hash table to the key. If that index is also taken, then the next is taken and so on until a free index is found. This rule is known as *linear probing*. According to another rule – known as *quadratic probing* – the new empty index is looked for at a distance from the collision equal to the square of the step number. In other words, with linear probing probes follow as x+1, x+2, x+3, x+4 and so on, while with quadratic probing the rule is x+1, x+4, x+9, x+16, and so on. Quadratic probing is more efficient in avoiding the formation of clusters, which slow down lookups [17]. However, they still can't avoid long sequences from forming, because the long keys that hash to the same position will all follow the same sequence of probing. To avoid this problem, *double hashing* (or *rehashing*) can be used, which hashes the key with a different function for the second time and uses the result as the step size.

In two of the 4 lookup procedures that we are targeting we use either a 16 bit short address or a 64 bit extended address. These addresses are randomly distributed and they do not contain redundant information. Therefore we hash them entirely. The range of these addresses is very wide and only 16 such keys will be used at the same time. On this purpose we use a *hashTable* of HTDIMENSION=23 positions. We use 7 extra positions to reduce the probability of collisions and to ensure a maximum of 70% usage even when the maximum number of keys (i.e., 16) is present in the hash table. Moreover, it is good practice to use a prime number for the hash table dimension to avoid reducing the modulo operation to the selection of the last several bits of the keys [18]. Thus, to squeeze the huge space of keys in the 23-position hash table we use the modulo operator ⨁ as follows:

$$position = key \oplus 23, \qquad (1)$$

where *position* indicates the cell in the hash table that will hold a pointer to the security information record containing *key*. In Figure 10 the value of *position* is obtained with the function *HashFunction*(). In order to prevent clustering and long probe sequences from forming due to repeated collisions, we use double hashing. We use the following function for rehashing:

$$stepSize = 5 - (key \oplus 5). \qquad (2)$$

This type of rehashing function has been proved to work efficiently [19]. The constant can be any prime number smaller than the hash table's dimension. The value of *stepSize* is obtained with the function *DoubleHash*() in Figure 10. With double hashing, if we denote the load factor by $\alpha = 16/23 = 0.7$, the number of probes for unsuccessful lookups will be $1/(1-\alpha) = 3.33$, while for successful ones it will be $1/\alpha \ln\left(1/1(1-\alpha)\right) = 1.7$. This is a worst-case estimate.

In Figure 10 we present the pseudo codes of the *Insert*(), *Lookup*() and *Delete*() algorithms that can be used with hash tables. Moreover, in Figure 10 the pseudo code of two auxiliary functions, namely *ResolveInsertCollision*() and *ResolveLookupCollision*(), is also presented. Beside the functions and constants described above, in these pseudo codes we use the position variable to indicate the *hashTable* index obtained through hashing and *newPosition* for the step size calculated with the *DoubleHashing*() function, in case of collisions. *AddressOf*() and *ValuesOf*() are two functions that return the address of a key and the key pointed by a pointer, respectively.

The *Insert*() function takes as an argument a *key* and it inserts it into the *hashTable*. First the key is hashed (line2) then its insertion into the *hashTable* to the obtained index is attempted. If the index position is empty (line 3), the address of the security information belonging to *key* is assigned to the corresponding *position* of the table. Otherwise a collision happened and *ResolveInsertCollision*() is called to resolve it (line 6). *ResolveInsertCollision*() takes as argument the *position* of the collision and the original *key* and all it does is to repeat probing the *hashTable* for an empty position using *DoubleHashing*() (line 34). When it succeeds, it returns the *newPosition* and this will be used as an index for the security data belonging to *key*.

<u>*Initialization*</u>: Allocate memory for the array *hashTable*[HTDIMENSION], containing pointers initialized with **null**; HTDIMENSION ←23, a constant, is the dimension of the hash table. After the insertion of a new *key*, the corresponding element of this *hashTable* will point to the to the address of the security information belonging to the *key*.

```
1. Insert(key) // Inserts a key into the hashTable
2.      position ← HashFunction(key);
3.      if (hashTable[position] == null) then
4.              hashTable[position] ← AddressOf(securityInfo.key);
5.      else
6.      hashTable[ResolveInsertCollision(position, key)] ← AddressOf(securityInfo.key);
7.      endif;
8. end Insert();

9. Lookup(key)// Returns security info belonging to key if found; returns null otherwise;
10.     position ← HashFunction(key);
11.     if (ValueOf(hashTable[position]).key == key) then
12.             return hashTable[position];// ValueOf() could be returned as well
13.     else
14.             collision ← ResolveLookupCollision(position, key);
15.             if (collision == HTDIMENSION) then
16.                     return "Key not found!";
17.             else
18.                     return  hashTable[collision]; // ValueOf() could be returned as well
19.             endif;
20.     endif;
21. end Lookup();

22. Delete(key) // Deletes key from hashTable
23.     position ← HashFunction(key);
24.     if (ValueOf(hashTable[position]).key == key) then
25.             hashTable[position]← null; // ValueOf() can be used as well
26.     else
27.             collision ← ResolveLookupCollision(position, key);
28.             if (collision != HTDIMENSION) then
29.                     hashTable[position]← null; // ValueOf() can be used as well
30.             endif;
31.     endif;
32. end Delete();

33. ResolveInsertCollision(newPosition, key)
34.     repeat
35.             newPosition ← newPosition + DoubleHash(key);
36.     until hashTable[newPosition] != null;
37.     return newPosition;
38. end ResolveCollision();

39. ResolveLookupCollision(newPosition, key)
40.     repeat
41.             newPosition ← newPosition + DoubleHash(key);
42.     until (ValueOf(hashTable[newPosition]).key != key and
43.             hashTable[newPosition] != null;
44.     if (hashTable[newPosition] == null) then
45.             return HTDIMENSION; // symbolic value returned: value not found;
46.     else
47.             return newPosition;
48.     endif;
49. end ResolveCollision();
```

Figure 10: Algorithms for the hash table approach

The *Insert*() function takes as an argument a key and it inserts it into the *hashTable*. First the *key* is hashed (line2) then its insertion into the *hashTable* to the obtained index is attempted. If the index *position* is empty (line 3), the address of the security information belonging to *key* is assigned to the corresponding *position* of the table. Otherwise a collision happened and *ResolveInsertCollision*() is called to resolve it (line 6). *ResolveInsertCollision*() takes as argument the position of the collision and the original key and all it does is to repeat probing the *hashTable* for an empty position using *DoubleHashing*() (line 34). When it succeeds, it returns the *newPosition* and this will be used as an index for the security data belonging to *key*.

Also the *Lookup*() function begins with hashing the *key* received as an argument, in order to obtain the position of that key in the *hashTable*. If at the obtained position it finds the desired *key* (line 11), *Lookup*() returns the pointer to the appropriate security data (line 12). Otherwise it calls the *ResolveLookupCollision*() function to find the position where the *key* had been stored. *ResolveLookupCollision*() uses the *DoubleHashing*() function to calculate the step size used for probing the *hashTable* (line 41). The probing continues until the desired *key* is found or an empty *hashTable position*, meaning that the *key* is not present in the table, is encountered. (Note that probing happens circularly in the *hashTable*, i.e., if the length of the table is exceeded, the probing continues at its beginning.) To indicate that the *key* has not been found, the HTDIMENSION is returned (line 45). This is interpreted correspondingly in the calling *Lookup*() function (line 15). In case of success the *position* where the *key* is found is returned to the *Lookup*() function (line 47), which, on turn, returns the pointer to the security data belonging to the *key* in question (line 18).

The *Delete*() function works very similarly to *Lookup*(), except that when it finds the position of the desired *key* in the *hashTable*, it empties that position (lines 25 and 29). If desired, an additional instruction can be added to delete also the security data belonging to the *key*.

7. Efficiency Comparison

In order to understand which of the three presented data structures serve better our needs, in this section we evaluate them based on the criteria enumerated in Sect. 2. Our evaluation reflects more of a worst-case scenario, even though we provide some considerations on average values as well. To help the understanding of the explanations on the CPat approach we also included performance estimations on the CB trie.

Below, a brief summary of the five symbols generally used for comparing the rates of growth of functions is provided [20]. On this purpose functions *f(x)* and *g(x)* are used and the growth of *f* is expressed in function of g. In this paper only the O, Ω and Θ symbols were used. $f(x) = o(g(x))$ means that *f* grows slower than *g* does when *x* is very large. $f(x) = O(g(x))$ means that *f* certainly doesn't grow at a faster rate than g. $f(x) = \Theta(g(x))$ means that *f* and *g* are of the same rate of growth, only the multiplicative constants are uncertain. $f(x) = \Omega(g(x))$ means that the calculation of *f* takes *at least* as much as the calculation of g. $f(x) \approx g(x)$ means that not only do *f* and *g* grow at the same rate, but that in fact *f/g* approaches 1 as $x \to \infty$.

To insert, lookup or delete a key into/in/from a *balanced* BST the number of keys that need to be evaluated is one per each level in the worst case and half the number of levels, *h*, on average. Thus, in each of the three cases the **time complexity** is $O(m * h)$, where *m* is the number of bits composing the key and is used here to help comparison with trie-related performance. However, if the tree is extremely unbalanced and takes the shape of a linked list, the number of evaluated nodes for these operations is equal to the number of nodes, *n,* in the worst case. Operation on such a tree have the time complexity of $\Omega(n * m)$. Node evaluations are performed on bit strings and not on bits as in the case of tries. This also adds to the time complexity of the operations.

When looking up a CB trie, the worst case is when the rightmost bit of the tree map is searched for. This requires scanning of the whole tree and hence its complexity is $O(2^h)$, with h being the maximum depth of the complete trie. Since in the case of CPat tries the dummy nodes are excluded from the tree map, the same operation requires Dx100 times less operation, where D ∈ [0,1] is the ratio of dummy leaves over leaf nodes in the CB trie [12]. Thus, the lookup time complexity with CPat tries is $O(D*2^h)$. The situation is similar with the insert and delete operations as well. The worst-case scenario is when the leftmost bit of the tree map has to be acted upon, since then all of the bits in the tree map have to be shifted. The time complexity in this case is $O(2^h - h)$ for the CB trie and $O(D*(2^h - h))$ for the CPat trie. Note that the complexities of CB and CPat algorithms differ only by the constant D. Thus, the growth of the CPat algorithm execution times with respect to the growth of the CB algorithm times is Θ(D). Also note that node evaluations in tries mean operations on bits, which are much simpler than operations on long bit string keys. Therefore, if we want to compare the performance of tree- and trie-based solutions, the number of nodes in trees must be multiplied with the length of the keys. However, while it is not straightforward how to compare the time complexities of trie- and tree-based algorithms, it can be clearly seen that hash tables-based algorithms promise the highest speeds, with reduced memory requirements as well. Since speed is the most important parameter of our security information lookup application, it is clear that the hash table-based solution is the most attractive one for us. With this latter approach, any node can be looked up, inserted or deleted by simply calculating the value of the hash function. This complexity is $O(1)$ and it does not depend on the number of keys, but it can degrade if collisions happen. However, if the hash table is dimensioned so that it never gets loaded over 70% of its capacity, the performance loss is not significant.

The **memory needs** of BSTs can be calculated as follows. Each BST node is made of 3 pointers of 2 bytes each (i.e., key, left child, right child). This requires 48 bits to store each node. Thus, to store the whole tree 48**k* bits are necessary, where *k* is the number of keys in the BST. For the CB and CPat tries the total number of bits required is given by the size of tree map, leaf or node map and BTBL. In a CB representation the tree map is of $2^{h+1}-1$ bits, leaf map is 2^h and BTBL is *K**16 bits long, where *h* is the maximum depth of the complete trie and *K* is the maximum number of keys that can be stored in the BTBL. This gives a total of $2^{h+1}-1+2^h+16k$ bits. On the other hand, in a CPat representation the tree map is $(1-D)*2^{h+1}-1$, node map is $2^{h+1}-1$and BTBL is again *k**16 bits long, totalling $(1-D)*2^{h+1}+2^{h-1}+16k-1$ bits. Such size complexities can result in quite high memory requirements. This is due to the fact that with tries not only a 16-bit pointer is stored for each key, but the nodes themselves are stored bit-by-bit. Therefore, 64 bit keys would occupy 4 times more memory here than in a tree. The memory needed to store a key in a hash table is 16 bits, since it represents a pointer. Thus, the total amount of memory needed to store the entire hash table depends on the maximum number of keys, *K*, that it can store at the same time and is 16**K*.

For example, for storing 16 keys, 64 bits each, setting D=0.3, with the above data structures the following amounts of memory would be necessary. With BST trees and hash tables 48*16=768 bits and $16*23=368$ bits would suffice, respectively. With tries the maximum depth of the trie is required for the calculation. Assuming that about 30% of all leaf nodes in the CB trie are dummies, then 16 keys (meaning 16 leaf nodes) imply about 8 dummies. Further we assume that we have 23 internal nodes in the trie. This means that the size of the tree map will be 24+23=47 bits, while the leaf map for the CB trie will be of 24 bits and the node map for the CPat trie will be of 23 bits. The BTBL in both cases will be of 16*16=256 bits. Thus the total memory used for the CB and CPat tries will be 47+24+256=327 and (1-0.3)*(47+23+256)=228.2 bits, respectively. This is a very good performance, however, keep in mind that these calculations highly depend on the actual values of the keys, since this has a major impact on the sparseness and depth of the tries.

The **scalability** of a BST trees depends on their depth, which depends on the number of nodes in the tree (many nodes result in many levels), but also on the shape of the tree (an unbalanced tree can have significantly more levels than a balanced one, with the same number of nodes). Since balancing the tree has its own cost as well and because operations are performed on bit strings and not bits, on a scale from 1 to 5 we assign the grade 3 to the scalability of BSTs, meaning a medium scalability.

Trie-based solutions are expected to provide higher scalability than tree-based ones for two reasons. First, tries operate on bits and not bit strings as trees do. Second, in tries common prefixes are represented only once, while in trees they are repeated each time they occur. On the other hand, trie-based algorithms are significantly more complicated than those used with trees. Altogether, on a scale from 1 to 5 we assign the grade 4 to the scalability of tries, meaning good scalability.

Given that time complexity does not depend on the number of keys in the case of hash tables and also because memory needs are low, the scalability of hash table-based algorithms is considered the highest among the data structures presented.

Hash tables are favourite also from the **hardware implementation** point of view, because algorithms are enough simple and the main characteristic operation that has to be performed is the computing of hash function values. Many hash functions have been specially designed to be effective for hardware implementation. Tree based algorithms are also simple, however they require fast handling of bit strings representing keys. Although this may not be too difficult for a resource-contained device either, it may somewhat slow down lookup times. This is why on a scale from 1 to 5 we assign trees the grade 4, meaning that the tree-based solution is good for hardware implementation.

Finally, despite the fact that tries operate on bits instead of bit strings, we consider them the most inappropriate for hardware implementation among the presented approaches, since the algorithms that operate on them are quite complicated for a resource-limited embedded device. This is why on a scale from 1 to 5 we assign them the grade 2, meaning that tries should possibly be avoided when it comes to implementing them in hardware.

Table 6 summarizes the above considerations on the efficiency of lookup solutions discussed in this paper. Note though, that they should be compared with care because the primitives of the various data structures are not equivalent. Thus, the number of levels in a tree should not be compared with the number of levels in a trie. However, from the perspective delimited by the information provided in this paper, the formulas of Table 6 can still provide solid grounds for understanding which of these solutions can serve better the needs of the security information lookup application.

Table 6

Various data structures efficiency summary for lookup procedures execution

Criteria	BST Tree	CB Trie	CPat	Hash Table
Lookup Complexity	$O(m*h)$	$O(2^h)$	$O(D*2^h)$	$O(1)$
Insert/Delete Complexity	$O(m*h)$	$O(2^h-h)$	$O(D*(2^h-h))$	$O(1)$
Memory needs for index (bits)	48*k	$2^{h+1}-1+2^h+16k$	$(1-D)*2^{h+1}-1+2^{h-1}+16k$	16k
Scalability	Medium	Good	Good	Very good
Hardware implementation	Good	Not good	Not good	Very good

8. Conclusions

In this paper an analysis on the possibilities for implementing lookup algorithms for ZigBee and other IEEE 802.15.4-based SoCs security lookup procedures is presented. An initial description about the security data as well as the security lookup procedures and the criteria used for evaluating lookup algorithms are described at the beginning of the paper. Based on this information the actual needs of the mentioned security application were loosely delimited and the search for lookup algorithms has been conducted from this perspective.

The search for lookup algorithms leads to the recognition that, first of all, it is important to find an adequate data structure for indexing security information. Once the data structure has been selected, lookup algorithms can be identified more easily, since they are bound to the data structures. Thus, three main data structures have been identified: binary search trees, tries and hash tables. For each of these data structures a general presentation is provided, followed by the pseudo code of insert, lookup and delete algorithms and, finally, their efficiency is estimated based on the selected criteria. The analysis shows that a hash table-based solution can highly outperform the other two data structures, especially on the most important metric, the lookup speed. Furthermore, the hash table-based approach has low memory needs as well, and it does not require complicated algorithms either. On the other hand, tree-based approaches are simple, but they compare many bit string keys until they find the desired one, which slows them down. Tries aim at improving on this by requiring bit comparisons only. However, they achieve this with additional lines of algorithm code, which renders them less attractive for implementation. In conclusion, based on the analysis in this document we suggest that the described hash table-based approach be implemented and the tree and trie based approaches should be considered for implementation only if execution time permits.

9. References

[1] T. Basmer, H. Schomann and S. Peter, Implementation analysis of the IEEE 802.15.4 MAC for wireless sensor networks, in: Proceedings of 2011 International Conference on Selected Topics in Mobile and Wireless Networking (iCOST), Shanghai, China, 2011, pp. 7-12. doi: 10.1109/iCOST.2011.6085840.

[2] L. Feng, H. Chen, T. Li, J. Chiou and C. Shen, Design and Implementation of an IEEE 802.15.4 Protocol Stack in Embedded Linux Kernel, in: Proceedings of 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops, Perth, WA, Australia, 2010, pp. 251-256. doi: 10.1109/WAINA.2010.72.

[3] G. Cornetta, A. Touhafi, D. J. Santos, J. M. Vázquez, Field-programmable gate array implementation of an all-digital IEEE 802.15.4-compliant transceiver, International Journal of Electronics, 97:12, (2010) 1361-1378, doi: 10.1080/00207217.2010.488909

[4] N. S Bhat, Design and Implementation of IEEE 802.15.4 Mac Protocol on FPGA, in: IJCA Proceedings on Innovative Conference on Embedded Systems, Mobile Communication and Computing (ICEMC2), 2011, pp. 1-5.

[5] P. Hämäläinen, M. Hännikäinen, T. Hämäläinen, Efficient hardware implementation of security processing for IEEE 802.15.4 wireless networks, In: Proceedings of the 48th IEEE Int. Midwest Symp. on Circuits and Systems (MWSCAS 2005), Cincinnati, OH, USA, 2005, pp. 484–487.

[6] M. L. Chávez, F. R. Henríquez, E. L. Trejo, AES-CCM implementations for the IEEE 802.15.4 devices, in: IFAC Proceedings Volumes, Volume 40, Issue 22, 2007. doi:10.3182/20071107-3-FR-3907.00030.

[7] M. Cherkaskyy and A. Melnyk, Complexity of specialized processors, The Experience of Designing and Application of CAD Systems in Microelectronics, 2003. CADSM 2003. In: Proceedings of the 7th International Conference., Slavske, Ukraine, 2003, pp. 209-210, doi: 10.1109/CADSM.2003.1255033.

[8] A. Melnyk. “Computer Memory with Parallel Conflict-Free Sorting Network-Based Ordered Data Access.” Recent Patents on Computer Science, volume 8, Issue1, 2015. pp. 67–77. doi: 10.2174/2213275907666141021234845.

[9] IEEE Standard 802.15.4, “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)”, Sept 2006.

[10] Viktor Melnyk. “Implementation Options of Key Retrieval Procedures for the IEEE 802.15.4 Wireless Personal Area Networks Security Subsystem.” Advances in Cyber-Physical Systems, 4, 1 (2019) 42–54.

[11] N. Sastry and D. Wagner, “Security considerations for IEEE 802.15.4 networks”, In: Proceedings of the 2004 ACM workshop on Wireless security, Philadelphia, PA, USA, 2004, pp. 32–42, doi: 10.1145/1023646.1023654.

[12] M. Shishibori, M. Okuno, Kazuaki Ando and Jun-ichi Aoe, An Efficient Compression Method for Patricia Tries, in: IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, Orlando, FL, USA, 1997, pp. 415-420 vol.1, doi: 10.1109/ICSMC.1997.625785.

[13] Donald R. Morrison, PATRICIA – Practical Algorithm To Retrieve Information Coded in Alphanumeric, Journal of the ACM, Vol 15, No. 4, 1968, pp. 514-334.

[14] Wiebren De Jonge, Andrew S. Tanenbaum and Reind P. Van De Riet. “Two Access Methods Using Compact Binary Trees.” IEEE Transactions on Software Engineering, vol. 13, no. 7 (Jul 1987), pp. 799-810, doi: 10.1109/TSE.1987.233491.

[15] W.Litwin, Trie Hashing, In: Proceedings of the ACM-SIGMOD International Conference on Management of Data, Ann Arbor, MI, 1981, pp.19-29.

[16] M. Jung, M. Shishibori, A. Tanaka and Jun-ichi Aoe, "A Dynamic Construction Algorithm for the Compact Patricia Trie Using the Hierarchical Structure." Information Processing & Management, Vol.38, No.2, 2002, pp.221-236, doi: 10.1016/S0306-4573(01)00031-0.

[17] M. Płaza, S. Deniziak, M. Płaza, R. Belka, P. Pięta, Analysis of parallel computational models for clustering, in: Proceedings SPIE 10808, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018, 108081O (1 October 2018). doi: 10.1117/12.2500795.

[18] Juris Viksna, Construction and Analysis of Efficient Algorithms, Course Handouts, 2020. URL: http://susurs.mii.lu.lv/juris/courses/alg2020.html.

[19] M. Waite, R. Lafore, Data Structures & Algorithms in Java, Waite Group Press, Corte Madera, CA, 1998.

[20] H. S. Wilf, Algorithms and Complexity, A K Peters/CRC Press, 2nd edition, 2002.